Safeguarding Controller Software from Single Event Upset via Conditional Variable Verification

Ganesha
International Institute of Information Technology, Bangalore
Bangalore, India
ganesha@iiitb.ac.in

Abstract

We present a method based on program analysis and formal verification to identify Conditionally Relevant Variables (CRVs)—variables that, if affected by a Single Event Upset (SEU), may lead to a safety property violation in control software. Traditional slicing and conditioned slicing distinguish relevant variables by dependence, but they over-approximate and produce false positives. Our approach defines CRVs directly with respect to a safety property, which can be a point assertion or a short temporal condition, and detects them by injecting exactly one SEU per variable and checking the property with a model checker. Experiments show that CRVs form a smaller and more precise set than slicing-relevant variables. This enables selective hardening, compiler-guided SEU resilience, and cost-efficient controller design.

Keywords: Conditional Relevance of a Variable, Single Event Upset, Hardening, Model Checking, Formal Verification, Program Slicing

1 Introduction

1.1 Single Event Upsets

Control systems are often deployed on microcontrollers or system-on-chip (SoC) devices that operate in harsh environments. Such systems are vulnerable to a random phenomenon called single event upset (SEU) [1], where a single bit flip in hardware can disrupt the computation and cause failures. In safety-critical domains such as automotive, aerospace, healthcare, and industrial automation, even one fault can lead to unacceptable consequences.

APLAS SRC 2025, Mon 27 - Thu 30 October 2025, (The International Institute of Information Technology Bangalore)

1.2 Hardening

A common technique against SEUs is hardening [2], where parts of the silicon are made resilient to bit flips. Hardening the entire chip is expensive, so practical designs protect only a subset. This creates the need to know which variables in software must run on hardened regions. Identifying such "critical" variables in advance allows the compiler to place them accordingly, achieving a balance between safety and cost.

1.3 Our Contribution

We address this by introducing the notion of Conditionally Relevant Variables (CRVs), variables whose corruption by a single SEU can change whether a safety property is satisfied. Variables not meeting this criterion are treated as non-CRVs and need not be hardened. While static slicing identifies variables relevant by dependence, it tends to over-approximate. Conditioned slicing refines this by considering predicates, but still ignores the actual impact of a fault on the property. Our method closes this gap by combining slicing with formal verification: we inject a single SEU before each use of a chosen variable and use a model checker to see if the safety property (assertion and a temporal condition such as "output stays in range for k cycles") changes outcome.

This paper makes the following contributions:

1. Introduce and formally define *conditional relevance* of a variable.
2. Propose an instrumentation and verification method to detect CRVs.
3. Experimentally show that CRVs form a smaller and more precise set than slicing-relevant variables, reducing false positives.

We believe this method can reduce controller chip cost by giving useful inputs to the compiler. The outcome is a tighter set of critical variables, providing direct input to compilers and enabling selective hardening with reduced chip cost.

2 Related Work

Program slicing [3, 4] extracts semantically relevant parts of a program but often over-approximates. Variants such as predictive slicing [5] and field-sensitive slicing [6] improve precision, yet under strict safety conditions they still mark variables that never affect the property. Conditioned slicing

```
int f(int x, int y) {
  int output = 4, count = 0;
  bool alarm = false;
  while(count++ < 7) {
    if(x > 10) output = (y == 1) ? 2 : 1;
    else { output++; alarm = true; }
  }
  printf("%b", alarm);
  return output;
}
```

Figure 1. Motivating example

[7] further refines relevance by considering predicates, but it remains dependence-based rather than fault-impact-based. Our work differs by checking the effect of an actual SEU on a given safety property.

Model checkers such as CBMC [8–10] support bounded verification and are used here to confirm whether a single injected SEU changes the property outcome. Hardware-level countermeasures (ECC, refresh, hardened cells) [11] and software techniques like SWIFT [12] reduce SEU impact, but they apply globally and do not identify variable-level criticality.

Unlike self-composition methods [13, 14], which duplicate the program to compare executions, our instrumentation introduces exactly one SEU per run, preserves control flow, and observes its direct effect on the safety property. This enables more precise detection of conditionally relevant variables in embedded control software.

3 Motivating Example

Figure 1 shows a program where the safety property is $output \leq 10$ at return. This is violated when $(x > 10)$ is false: the loop executes seven times, increasing `output` to 11. Variable `y` is marked relevant by static slicing, as `output` depends on it in one branch. However, `y` does not influence whether the property is violated. Even conditioned slicing (considering predicates) would still include `y`. Our method identifies that `y` is not a CRV with respect to the safety property. This example illustrates how slicing over-approximates, while CRV detection pinpoints only those variables whose SEU can actually flip the outcome.

4 Terms and Concepts

4.1 Dependencies, Slicing and Observations

In programs, dependencies arise via data or control flow. Program slicing extracts variables that influence an output point [4], but it over-approximates under strict safety conditions. For instance, in Figure 1, slicing reports y as relevant even though it never affects the violation of the property $output \leq 10$. Our method uses slicing only to bound the search space

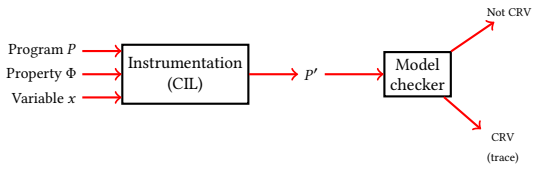

Figure 2. Overall approach

and then applies formal verification to eliminate such false positives.

4.2 The safety property Φ

A safety property specifies acceptable behaviour at control cycle boundaries. Unlike strict correctness, transient SEUs may be tolerated if they do not persist. For example, a temperature control system may allow the output o to deviate briefly but not remain outside (r_{min}, r_{max}) for k consecutive cycles. This can be expressed as:

$$\Phi(O) = \neg\Big(\bigwedge_{i=0}^{k-1}(O_i < r_{min} \vee O_i > r_{max})\Big).$$

Such temporal conditions capture the practical notion of safety.

4.3 Conditionally Relevant Variable (CRV)

A variable is *relevant* if it influences outputs; slicing can find these. A variable is a *Conditionally Relevant Variable (CRV)* if a single SEU on it can flip the truth of a safety property in some execution. Formally, let $P(\mathcal{I})$ and $P'(\mathcal{I})$ be the outputs of runs with and without an SEU. Then variable x is a CRV if for some input $\mathcal{I}$,

$$\Phi(P(\mathcal{I})) \neq \Phi(P'(\mathcal{I})).$$

This covers two situations:

- **Violation due to fault:** the property holds without the SEU but fails when the SEU occurs.
- **Masked violation:** the property fails without the SEU but appears to hold when the SEU occurs.

Both cases are undesirable: the first leads to unsafe behaviour, while the second hides real faults from being detected. Variables that fall in either category are treated as CRVs; all others are non-CRVs.

5 Our Approach

5.1 Overall Architecture

Figure 2 shows our workflow. A program P, safety property Φ, and variable x (under test) are inputs. The program is instrumented to P', which enforces that exactly one SEU may occur on x. The model checker then verifies whether Φ always holds. If yes, x is not a CRV. Otherwise it produces a counterexample trace showing how a fault on x leads to violation of Φ.

5.2 Instrumentation

We instrument the program P to create P', which enforces the *only-one-SEU* rule on a chosen variable x. The idea is simple: before every use of x, we insert a helper call `mimic_seu_effect(&x)`, which nondeterministically flips a single bit of x at most once during execution. Alongside the normal computation, we maintain a buffer O of recent outputs and evaluate the safety property $\Phi(O)$ at the end of each control cycle.

Finally, we assert that the property outcomes with and without SEU (ϕ and ϕ') do not diverge:

$$\texttt{assert}(\neg(\phi \oplus \phi')).$$

If the model checker finds a counterexample to this assertion, then x is classified as a CRV; otherwise it is a non-CRV.

5.3 Mimicking SEU Effects

SEUs are mimicked by inserting a helper call before every use of the variable under investigation. Our tool, built on CIL [15], inserts `mimic_seu_bitflip(&x)`, which flips one randomly chosen bit of x using XOR logic. A global `flag` ensures that at most one SEU occurs per run, enforcing the *only-one-SEU* semantics. Instrumentation preserves the original program behaviour except for this injected fault, allowing the model checker to explore both safe and unsafe outcomes of the safety property.

6 Experimental Results

6.1 Experimental Setup

The goal is to validate and demonstrate the effectiveness of the proposed algorithm on a set of C controller programs. We use Frama-C (v30.0) [16] for slicing to *bound* the variable set and keep only code relevant to the chosen safety property at the output point (just before writing to the actuator/plant). Properties are either a point assertion on the controller output or a short temporal condition (e.g., staying within a range for k consecutive cycles).

Instrumentation is performed automatically using a CIL-based tool [15], which inserts `mimic_seu_bitflip(&x)` before each use of the variable under investigation and enforces an *only-one-SEU-per-run* rule via a flag. The instrumented program preserves the original control flow; no program duplication is used. We then invoke CBMC (v5.12) [8–10] for bounded model checking. The model checker compares executions with and without the injected SEU. If both agree on the safety property outcome, the variable is not a CRV. If the SEU flips the outcome—making a previously safe run unsafe, or masking an unsafe run—the variable is classified as a CRV.

Program Category	#Cases	Observation
No additional CRVs	8	Matches slicing; no difference detected
More CRVs than slicing	9	Our method flagged extra CRVs missed by slicing
Time-based safety CRVs	7	Detected only when temporal conditions were used
Buffer-triggered path blowup	7	CBMC faced path/state explosion
Clamping-based Logic	8	Saturation kept outputs safe; all inputs non-CRV

Table 1. Summary of CRV analysis across 39 controller programs

6.2 Case Studies and Evaluation

We applied our algorithm to 39 C controller programs, covering common closed-loop designs such as temperature control, cruise control, and motor regulation. Properties ranged from simple range checks to temporal conditions over output histories. While the programs were hand-written, they capture typical embedded controller patterns such as clamping, resets, and timing guards.

Key findings are summarised below:

- **No extra CRVs (8 cases):** slicing alone was sufficient; our check confirmed no additional variables mattered.
- **More CRVs than slicing (9 cases):** static slicing missed some variables whose SEU changed the safety property; our method flagged these correctly.
- **Time-based properties (7 cases):** buffer logic (e.g., output $\leq$ 15 for 4 cycles) revealed CRVs visible only under temporal conditions.
- **State explosion (7 cases):** output buffers caused CBMC to face path/state blowup—an expected limitation of bounded checking.
- **Clamping logic (8 cases):** saturation ensured outputs always stayed safe; our analysis marked all inputs as non-CRVs, avoiding false positives from slicing.

Overall, our method is both stricter and more precise than slicing, and complements conditioned slicing by directly checking the impact of SEUs on the safety property. This enables more accurate guidance for compiler optimisations and selective hardening.

7 Conclusion and Future Work

We presented a method to identify *conditionally relevant variables* (CRVs)—those variables whose single-event upset can change the outcome of a safety property in control software. Unlike slicing or conditioned slicing, which are dependence-based and often over-approximate, our approach checks the direct effect of an injected fault on the property. Experiments on 39 controllers showed that CRVs are often fewer than slicing-relevant variables, especially under clamping or temporal safety conditions. This enables selective hardening and compiler guidance at lower cost.

References

[1] A. Jilani, D. Sharma, and R. Naaz, "Single Event Upset," in *National Conference on Industry 4.0 (NCI 4.0)*, 2020. [Online]. Available: https://www.researchgate.net/publication/351515765_Single_Event_Upset

[2] Science Alert, "Overview of Radiation Hardening Techniques for IC Design." [Online]. Available: https://scialert.net/fulltext/?doi=itj.2010.1068.1080

[3] M. Weiser, "Program Slicing," in *Proc. 5th Int. Conf. Software Engineering*, 1981, pp. 439–449.

[4] F. Tip, "A Survey of Program Slicing Techniques," *J. Programming Languages*, 1995.

[5] A. Yadavally et al., "Predictive Program Slicing via Execution Knowledge-Guided Learning," in *Proc. ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA)*, 2024.

[6] C. Galindo et al., "Field-Sensitive Program Slicing," *J. Systems and Software*, 2024.

[7] S. Danicic, C. Fox, M. Harman, R. Hierons, J. Howroyd, and M. Ward, "Static Program Slicing Algorithms for Conditioned Slicing," *Journal of Software Maintenance and Evolution*, vol. 17, no. 5, pp. 345–377, 2005.

[8] D. Kroening and M. Tautschnig, "CBMC–C Bounded Model Checker," in *Proc. 20th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2014, pp. 389–391.

[9] F. R. Monteiro, L. C. Cordeiro, and D. A. Nicole, "Model Checking C++ Programs," *Software Testing, Verification & Reliability*, vol. 32, no. 3, p. e1793, 2022.

[10] B. Fischer et al., "CBMC-SSM: Bounded Model Checking of C Programs with Symbolic Shadow Memory," in *Proc. Int. Conf. Automated Software Engineering*, 2023.

[11] Y. Lu et al., "A Self-Adaptive SEU Mitigation Scheme for Embedded Systems in Extreme Radiation Environments," *IEEE Syst. J.*, vol. 16, no. 1, 2022.

[12] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software Implemented Fault Tolerance," in *Proc. Int. Symp. Code Generation and Optimization (CGO)*, 2005, doi:10.1109/CGO.2005.34.

[13] G. Barthe, P. D'Argenio, and T. Rezk, "Secure Information Flow by Self-Composition," in *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, 2004, pp. 100–114.

[14] T. Terauchi and A. Aiken, "Secure Information Flow as a Safety Problem," in *Proc. Int. Symp. Static Analysis (SAS)*, 2005, LNCS vol. 3672, Springer, pp. 352–367. Available: http://theory.stanford.edu/~aiken/publications/papers/sas05b.pdf

[15] CIL Project, "Infrastructure for C Program Analysis and Transformation, Version 1.7.3." [Online]. Available: https://cil-project.github.io/cil/doc/html/cil/

[16] Frama-C, "Slicing Plugin." [Online]. Available: https://frama-c.com/fc-plugins/slicing.html