Safeguarding Controller Software from Single Event Upset via Conditional Variable Verification

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

***Abstract*—We present a method based on program analysis and formal verification to identify conditionally relevant variables (CRVs) – a novel class of variables that, if affected by an SEU, may lead to safety violation in control software under single event upsets (SEUs). Traditional static analysis distinguishes between relevant and irrelevant variables but fails to consider conditions specific to control logic, leading to false positives. Our approach defines and detects CRVs using formal verification to eliminate such false positives. Experiments show that CRVs are fewer than what traditional slicing detects, and our algorithm precisely isolates them. This enables selective hardening, compiler-guided SEU resilience, and improved cost-efficiency in safety-critical controller design.**

***Index Terms*—CRV (Conditional Relevance of a Variable), SEU (Single Event Upset), Hardening, Model Checking, Formal Verification, Program Slicing**

I. Introduction

A. Single Event Upsets

Control systems are implemented as control algorithms running on microcontroller or *system-on-chip* (SoC) devices. Such embedded controllers often function in harsh environmental conditions, exposing computing components to a rare random phenomenon called *single event upset* (SEU) [1]. When an SEU happens, a bit in the chip flips, possibly causing computation to fall off track and leading to errors – often with catastrophic consequences in safety critical applications like automotive, industrial automation, aerosphere, oil and gas, mining, and healthcare.

B. Hardening

A common industry technique to safeguard electronic components from SEUs is *hardening* [2]. The hardened silicon can withstand SEUs and avoid bit-flips. However, it increases chip cost. Hence, hardening only a portion of the chip can balance SEU risk with manufacturing cost. This requires that code be generated accordingly — critical parts of the code must run on the hardened chip region, while 'non-critical' computation can run on the non-hardened region. In this paper, we present a method to statically identify 'critical' variables whose SEU-induced faults could throw computation off-track and cause unacceptable errors.

C. Our Contribution

We present an approach to divide variables in a control algorithm into: *conditionally relevant variables* (CRVs) – to be placed in the hardened silicon, and *conditionally irrelevant* (non-CRVs) – safe in the unhardened part. Discovering the precise set of CRVs is undecidable. Static slicing techniques yield a sound but over-approximate CRV set. Our algorithm uses model checking to tighten this bound without losing soundness.

This paper makes the following contributions:

1) Introduce and formally define *conditional relevance* of a variable.
2) Combine static analysis (slicing) and formal verification (model checking) to identify CRVs/non-CRVs.
3) Experimentally show:
    a) Control software does have conditionally irrelevant variables.
    b) Formal verification over slicing helps detect more non-CRVs, avoiding false positives.

We believe this method can reduce controller chip cost by giving useful inputs to the compiler. Figure 1 illustrates selective hardening: a program is analysed (using our proposed algorithm) to identify SEU-critical parts, passed to the compiler, which ensures critical variables go to the hardened chip region.

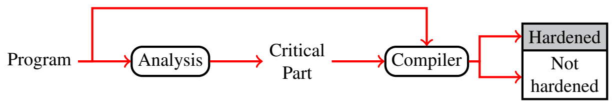


Fig. 1. Hardening

II. Related Work

Program slicing [3], [4] is a widely used program analysis technique to extract semantically relevant parts of a program but often yields over-approximate results. Enhancements like predictive slicing [5] and field-sensitive slicing [6] improve precision but may still fail under strict safety conditions. Tools such as CBMC [7], [9] and ESBMC [8] are used for software verification, supporting features like symbolic

```
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;
}
```

Fig. 2. Motivating Example

shadow memory and bounded model checking. Our method leverages such model checkers for accurate detection of CRVs. SEU mitigation is often handled through hardware solutions like ECC and refresh-based hardening [10], or software fault-tolerance techniques like SWIFT [11]. These do not offer variable-level precision for CRV detection. Our work combines static analysis with formal verification to address this gap.

Unlike self-composition approaches like [14], [15], which reduce information flow checks to safety verification by comparing dual program copies, our work introduces fault at a single point and analyses its direct effect on the safety property—without duplicating program state.. We enforce *only-one-SEU* semantics per run via instrumentation, preserving the control flow and enabling precise detection of safety violations in timing-sensitive embedded systems.

III. Motivating Example

In Figure 2, we show a program fragment that returns $output$. Suppose the safety property is $output \leq 10$ at return. This can be violated: the while loop runs 7 times, and if $(x > 10)$ is false, $output$ is incremented by 1 in each iteration. Hence, $output = 11$ after 7 iterations, violating the property. Interestingly, this computation never uses y, though $output$ *is* dependent on y (as static slicing would reveal). But y does not affect $output$ *when* the safety property is violated. So, y is irrelevant to the safety condition $output \leq 10$. We use formal verification to identify such cases. Though simple, Figure 2 captures typical control logic where a safety violation occurs only under specific paths, showing how slicing often marks more variables than actually needed.

IV. Terms and Concepts

A. Dependencies, Slicing and Observations

In any program, dependencies arise via data (data dependency) or control (control dependency). Program slicing identifies parts of a program that influence a variable at a point of interest (slicing criterion). While slicing (static, dynamic, backward, forward) is sound and widely used [4], it may over-approximate and retain irrelevant variables under specific safety conditions.

Static slicing is sound, it will not miss any CRV, but is often imprecise. For example, in Figure 2, it will not detect that y is not a CRV for $output > 10$ safety condition. We use program slicing and formal verification to identify CRVs. This helps the compiler to place CRVs in hardened region during registers allocation. These bounds are tighter than traditional static analysis, enabling selective hardening and cost savings without compromising safety.

B. The safety property Φ

A safety property defines the safety of computation and is usually weaker than strict correctness. In closed-loop feedback control, many SEU-induced glitches do not persist and are self-corrected due to the plant's dynamic nature, which often behaves like a low-pass filter removing high-frequency transients.

For instance, in a temperature control system, if the output variable o (heating rate) is affected by an SEU, the plant temperature θ will not change quickly due to high time constants. So, the next o (computed from $\theta_{ref} - \theta$) is barely influenced by the prior SEU. The system can 'forget' such one-time upsets. Only when such exceptions persist, they cause unacceptable changes. The safety property captures acceptable plant behaviour.

Formally, the safety property checks if the output at the *output point* (end of control cycle) is 'safe', derived from plant physical properties. Currently, this comes from domain knowledge.

For example, it may be stipulated that the computed control output o must not be outside a certain range, say (r_{min}, r_{max}) for more than 5 consecutive times. Suppose O is a buffer which is used to store the last 5 values of the output variable, with O_4 being the last computed value and O_0 being the fifth most recent value of o. Then,

$$\Phi(O) = \neg(\bigwedge_{i=0}^{4}(O_i < r_{min} \vee O_i > r_{max}))$$

C. Conditionally Relevant Variable (CRV)

We now present the idea of conditionally relevant variables by first giving an informal explanation and then presenting a formal definition.

Relevant variables are those on which the output of the computation is dependent. Static analysis, e.g. program slicing, can identify relevant variables. On the other hand, a *conditionally* relevant variable (CRV) is one which affects the satisfaction of a given safety property Φ. Conversely, a *conditionally irrelevant variable* (non-CRV) is one whose value does not influence the satisfaction of Φ. Therefore, a variable may be relevant, but conditionally irrelevant as per the safety property. More specifically, a CRV is one whose value – if changed randomly during program execution – would lead to a change in the value of Φ in at least one execution.

Now, we present a formal definition. Let V be the set of all program variables. Let $I(\subseteq V) = \{x_1, x_2, .., x_n\}$ be the set of input variables, i.e., values set via environment (plant, user, sensors). Let $\mathbf{I}$ denote the set of all possible input vectors, where each vector represents a full run with environment inputs for all input variables. Let $\mathcal{I} \in \mathbf{I}$ be a specific input vector.

Let $\pi(\mathcal{I})$ be the trace of the original program with input $\mathcal{I}$, i.e., the sequence of program locations visited during execution. Let $P(\mathcal{I})$ be the output points in this trace. Suppose $x \in V$ suffers an SEU at some point $l \in \pi(\mathcal{I})$, causing a modified trace $\pi'(\mathcal{I})$ and output set $P'(\mathcal{I})$. The prefix of $\mathcal{I}'$ (with bit flip) up to l is the same as that of $\mathcal{I}$. Formally,

$$\begin{aligned} CRV(x) = \exists\, \mathcal{I} \in \mathbf{I} \text{ such that} & \\ & (\forall\, p \in P(\mathcal{I}),\ \Phi_p = \text{true} \wedge \\ & \quad \exists\, p' \in P'(\mathcal{I}),\ \Phi_{p'} = \text{false}) \\ & \vee \\ & (\forall\, p' \in P'(\mathcal{I}),\ \Phi_{p'} = \text{true} \wedge \\ & \quad \exists\, p \in P(\mathcal{I}),\ \Phi_p = \text{false}) \end{aligned} \tag{1}$$

In other words,

1) CRV requires safety violation for some input $\mathcal{I}$ when x is upset ($\Phi'_p = false$), while $\Phi_p = true$ without upset.
2) Or, $\Phi'_p = true$ while $\Phi_p = false$, i.e. fault is masked due to SEU on x.

When an SEU affects x, the first case causes fault; the second masks fault. Both are undesirable. Second case is rarer, but possible. All $x \in V$ for which $CRV(x) = true$ are CRVs; others are non-CRVs.

V. Our Approach

A. Overall Architecture

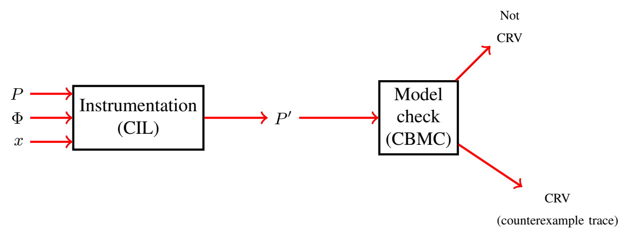


Fig. 3. Overall approach

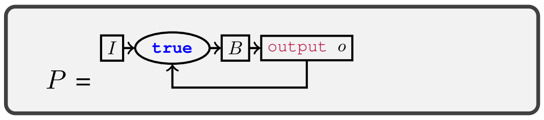
Figure 3 shows our approach. P is the control program, Φ is the safety property, and x is the *variable under investigation*, i.e., we test if x is a CRV. We first instrument P to obtain P', which is passed to a model checker. The model checker either proves x is not a CRV or returns a counterexample trace through P' showing how an SEU on x leads to a violation of Φ.

B. Instrumentation

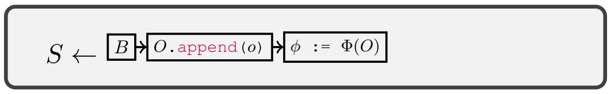
Our overall approach instruments the control procedure P for verification. The instrumented version is P'. Algorithm 1 shows the structure of P: I is the initialisation part, and B is the body of the main loop computing the output o. The sequence S is formed by appending `O.append(o)` and ϕ `:=` $\Phi(O)$ to B. To construct P', we merge the original and modified versions of P by replacing all variables in P with new ones: $I' \leftarrow I[V'/V]$ and $S' \leftarrow S[V'/V]$, where V is the original set and V' is the new set. S' is further modified by inserting `mimic_seu_effect(&x)` before each use of $x \in V$, mimicking SEUs. O is a buffer that stores the last n values of o; `O.append(o)` appends o, and drops the first element if O already has n elements. The variable x under investigation may suffer SEU anytime, so we insert `mimic_seu_effect(&x)` before every use of x (see Section **??**).

Algorithm 1 Instrumentation algorithm

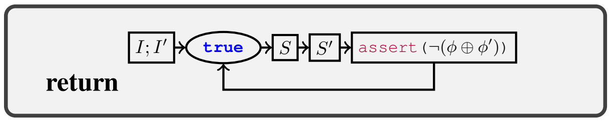
```
function INSTRUMENT($P$)
        ▷ $P$ is a control algorithm with the following structure:
```



```
        ▷ where $B$ is the body of an arbitrary control computation.
```



```
    $I' \leftarrow I[V'/V]$
    $S' \leftarrow S[V'/V]$
```



C. Mimicking SEU Effects

To mimic SEU effects, we instrument the program statically by injecting a bit flip on the variable under investigation before each of its uses. We have developed a tool, built on C Intermediate Language (CIL) [12], that performs this automatically by inserting calls to the helper function `mimic_seu_bitflip`. SEUs can occur at any point in the program, but we ensure only one SEU is introduced per analysis using a `flag` variable. The method for mimicking SEU effects is detailed in Figure 4. A single-bit flip is introduced at a random bit position using XOR logic. Instrumentation preserves the original program semantics, except for the calls to `mimic_seu_effect`, which are inserted before every use of `x` (the variable under investigation). These instrumentations are intended for static verification, allowing the model checker to explore both satisfying and violating scenarios of the safety property.

VI. Experimental Results

A. Experimental Setup

The goal of our experiment is to validate and demonstrate the effectiveness of the proposed algorithm. We use a set of controller programs written in C. Program slicing is performed

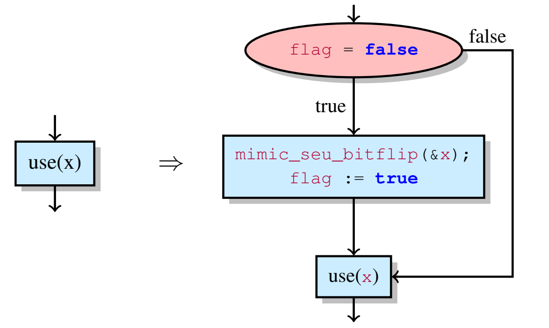


Fig. 4. Method for Mimicking SEU Effects

using Frama-C version 30.0 [13], and model checking is done using CBMC version 5.12. For each case study, a slicing criterion is chosen based on a safety property defined on the controller output at the output point (just before writing to the actuator/plant). The input program is first sliced to retain only those statements relevant to the safety property. This sliced code is then instrumented by mimicking SEU effects on the variable under investigation, producing the instrumented version. The instrumented program and the safety property are then given to CBMC. It performs bounded model checking to check property satisfaction and provides a counterexample trace if the property is violated.

B. Case Studies and Evaluation

We validated our algorithm across 39 controller programs, all written in C by us. These include typical closed-loop designs and general embedded control logic used in industrial controller scenarios. The safety properties vary from simple range checks to timing-based conditions and clamping enforced on the output.

Our evaluation revealed five key categories: (1) cases where slicing alone was sufficient and our method detected no additional CRVs; (2) programs where our method discovered additional CRVs missed by slicing; (3) controllers with time-based properties (e.g., output $\leq$ 15 for 4ms), handled using output buffers; (4) cases where buffer logic introduced path/state explosion in CBMC; and (5) Clamping logic – where output is always forced within safe limits – ensured no single input variable led to safety violation, hence no CRVs were detected.

Program Category	#Cases	Observation
No additional CRVs	8	Matches slicing; no extra non-CRVs detected
More CRVs than slicing	9	Static slicing missed these CRVs
Time-based safety CRVs	7	Output history influences safety
Buffer-triggered path blowup	7	CBMC faced space/state explosion
Clamping-based Logic	8	Output is bounded, so no input causes violation; no CRVs detected

TABLE I
SUMMARY OF CRV ANALYSIS ACROSS 39 CONTROLLER PROGRAMS

In some programs, even though multiple input variables affected the internal control logic, the final output was always passed through saturation or bounding logic. This ensured that any single-bit upset did not lead to a safety violation, and our algorithm correctly marked all such inputs as non-CRVs. In several programs, our method identified additional non-CRVs that slicing failed to eliminate. These cases show that our CRV detection is more accurate, especially when the program uses bounding logic that limits the output within a safe range.

C. Additional Observations: Buffer-Based Safety Conditions

Several controllers used temporal properties where safety violations depend on sustained output patterns, not instantaneous values. To support these, we implemented a buffer to hold recent output values across loop iterations. For example, in one case, the safety condition required that `output` $\leq$ 15 for 4ms continuously. Our algorithm was able to trace these patterns and identify the correct CRV set. We used a buffer to track actual output values over time, enabling checks for properties like monotonicity or average-value constraints. CBMC was able to trace faults via assertion checks, though with increased state space in such scenarios.

We note that time-based safety conditions can expose additional non-CRVs, but only when the controller's output depends on values from previous iterations. If the safety check uses only the current output, all relevant variables continue to remain CRVs.

VII. Conclusion

In this paper, we introduced the concept of conditionally relevant variables (CRVs)–variables whose values must not be affected during execution to maintain safety in control programs. This is determined based on whether a given safety property continues to hold after a fault is injected. Our method combines static slicing and formal verification to identify CRVs more precisely w.r.t. traditional static analysis. We have shown that the set of CRVs is often smaller than the set of all relevant variables, enabling selective protection without compromising correctness. This enables selective chip hardening, reducing design cost without compromising safety.

A. Future Work

Currently, our method supports single-procedure programs; we are extending it to multi-procedure programs using interprocedural analysis. We also plan to use other model checking methods apart from CBMC (which is sound but within given bounds), to increase coverage and soundness. A key future direction is to automate the derivation of safety properties from system-level specifications. Finally, we are incorporating our approach into a compiler that uses CRV information to guide SEU-aware register allocation during compilation.

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] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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