Safeguarding Controller Variables from SEUs using Static Analysis

Ganesha
IIIT-Bangalore
Samsung R&D - Bangalore
India
ganesha@iiitb.ac.in

Sujit Kumar Chakrabarti
IIIT-Bangalore
India
sujitkc@iiitb.ac.in

ABSTRACT

Control systems deployed in safety-critical applications face the risk of Single Event Upsets (SEUs), wherein rare, random phenomena can induce errors, potentially leading to catastrophic consequences. This paper proposes a novel approach to enhance the reliability of control algorithms by identifying conditionally relevant variables (CRVs) through a combination of static analysis and formal verification techniques. Once the CRV has been located, the compiler is instructed to produce the code that lives in the silicon chip's hardening region, remaining Non-Critical part is moved to non-hardened part. Hardening causes the cost of the chip to go up; hence, it is in our interest to harden only a portion of the chip that balances the risk of SEUs with the rise in manufacturing costs of the chip. We know that discovering the actual set of CRVs is undecidable. Slicing gives us a sound but loose upper bound of CRVs. Program analysis and verification is used to detect irrelevant variables that are missed by static slicing leading to sound and more precise estimates.

KEYWORDS

Program Analysis, CRV, SEU, Hardening, Model Checking, Formal Verification

ACM Reference Format:
Ganesha and Sujit Kumar Chakrabarti. 2024. Safeguarding Controller Variables from SEUs using Static Analysis. In *17th Innovations in Software Engineering Conference (ISEC 2024), February 22–24, 2024, Bangalore, India.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3641399.3641471

1 INTRODUCTION

The de-facto way of deploying control systems is through computing chips. The control algorithm is implemented as a software running on a micro-controller or (System on a chip) SoC device. Often, such embedded controllers function in harsh environmental conditions. In such cases, the computing components get exposed to a rare random phenomenon called single event upsets (SEU). When SEU happens, a bit in the chip would flip its value. This could lead the on-going computation to fall off track leading to errors – often with catastrophic results in safety critical applications, e.g. automotive, industrial automation, aerospace, oil and gas, mining and healthcare etc.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISEC 2024, February 22–24, 2024, Bangalore, India
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1767-3/24/02
https://doi.org/10.1145/3641399.3641471

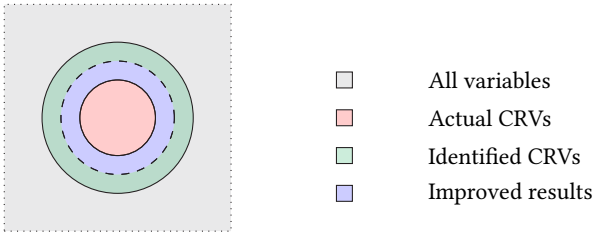

Figure 1: Bounds

A common technique used in the industry to safeguard electronic components from SEUs is called hardening [5]. The hardened part of the silicon can withstand the causes of SEUs, and bit-flips are avoided. The process of hardening the chip results in an increase in its cost. Therefore, it is advantageous for us to selectively harden a specific section of the chip that effectively manages the trade-off between the risk of Single Event Upsets (SEUs) and the corresponding rise in manufacturing expense.

In this paper, we outline an approach for dividing the source code of the control algorithm into two parts: one that should be placed in the hardened part of the silicon, and the other that can be left unguarded by being placed in the unhardened part. For this, we define the idea of conditional relevance of a variable. We use a combination of static analysis (program slicing) and formal verification (software model checking) to solve this problem. Note that discovering the actual set of conditionally relevant variables (CRVs) is undecidable. Slicing gives us a sound but loose upper bound of CRVs. Formal verification helps us tighten this bound without compromising soundness. This is shown schematically in Figure 1.

2 MOTIVATING EXAMPLE

In Figure 2, we show a fragment of code. The program computes the value of the output variable 'output'. Suppose that there is correctness condition imposed on the system that says that output should be less than or equal to 10. We can see that it is possible to violate this condition in the given code. The program iterates through the outer while loop exactly 7 times. If the branch condition (x > 10) is false, output would be incremented by 1 in each iteration. In 7 iteration, its value would be 11, which would violate the safety condition. But, an interesting aspect of this situation is that the above computation never uses the value of y. This is not to say that output is not dependent on y; in fact, it is dependent on y. A traditional static slicing would reveal this to us. But what this technique would fail to discover is that output doesn't depend on y's

```
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

value when it violates the safety condition. This is to say that y is not relevant as far as the safety condition output <= 10 is concerned. We resort to formal verification to identify such cases. The key takeaway from the example is the identification of conditionally relevant variables (CRVs) concerning a given safety property. In this instance, even though y is used in the computation of output, its value does not influence the violation of the safety condition. Therefore, y can be considered as not critical for the safety of the program, and this understanding guides our approach to selective hardening.

3 STATE OF THE ART

Single event upset (SEU) refers to a change in state of a semiconductor memory or processor caused by a high-energy particle, such as a cosmic ray. These particles can cause a bit flip, which means a binary 0 is changed to a binary 1 or vice versa [8], [7]. To mitigate the risk of SEUs, designers can implement techniques such as error-correcting codes (ECC) or redundancy. ECC adds extra bits to each data word that allow the detection and correction of single-bit errors. Redundancy involves duplicating critical components or systems so that if one fails due to an SEU, a backup can take over the function. The selective radio hardening is used with circuit components corresponding to bits identified as susceptible to errors, such as transient errors [9], [4]. All of the proposed strategies to minimize SEUs are hardware-based. Our method is distinct since the proposed solution is at the controller algorithm, which is a software component.

While the general technique may have been explored in previous research [1] [3] [6], our work introduces several novel aspects specific to the domain of control algorithms. Domain-Specific Challenges: Control algorithms, especially those deployed in safety-critical systems, present unique challenges that may not be fully addressed by generic techniques. Our paper delves into the details of control algorithms, highlighting specific challenges related to conditionally relevant variables (CRVs) and their impact on system safety. The exploration of these challenges is a key differentiator from existing literature.

4 PROPOSED SOLUTION

4.1 Backgroud

In a given program, when one part, say A, affects the computation happening at some other part, say B, we say that there is a dependency between A and B, or B is dependent on A. We speak of two types of dependencies: data dependency and control dependency. When the computation at A gets used as an input for the computation at B, we say B is data dependent on A. When whether the code at B gets executed or not is determined by the computation at A, B is said to be control dependent on A. In order to perform an analysis of the CRVs, it is necessary for us to have knowledge of the data flow throughout the program as well as their dependencies. The combination of the data dependency, the control dependence, and the changing dependencies is what constitutes these dependencies. The variable and location pair say, (v, l) is dependent on (x, lx) and (x, lx) is dependent on (y, ly) then (v, l) is dependent on (y, ly). Dependencies between data and control, data and data, control and data, and control and control are all possible. Note that we are not explicit about the type of dependency.

A program slice (or slice for short) in a program is the part of the program which is related in some causal way to the program location of interest. For example, a backward slice gives us those parts of the program which may, in some or the other run of the program, have an influence on the value of a variable of interest at a program point of interest. This pair of variable (v) of interest and program point of interest (l) for the pair (v, l) and are known as the slicing criterion.

Program slicing refers to a broad class of techniques in program analysis with the primary intent to discover slices. Program slicing can be static or dynamic, backward or forward, among a host of other variants. Slicing has been successfully and widely used to solve many software engineering problems. Program dependence graph (PDG) is the data structure that is used as the first step towards computing a slice. In fact, once the PDG is available, computing slice is almost trivial. A PDG explicates is formed by super-imposing the dependency information on top of the control flow graph of a program. We begin by applying program slicing to the program of interest; as a result, we will receive a list of CRVs that contains some false positives. As a result, this is the very first stage in our strategy. In the following stage, we will tighten the bound on CRVs by making use of formal verification techniques.

4.2 Observations

Static slicing yields sound results, i.e. it will not miss identifying any CRV. However, it will often yield imprecise results. For example, in the given example, it will not be able to detect the fact that y is not a CRV for output >= 10 at the return statement. In this work, we use program slicing and formal verification to identify the conditionally relevant parts of the program which should be safeguarded. As a result, 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. This will make it possible to harden a smaller part of the silicon without compromises on safety, leading to cost saving without loss of safety guarantee.

```
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 3: Instrumented Code

4.3 Program Instrumentation

A variety of formal verification techniques can be employed to detect the relevant variables. For this, we first need to instrument the given code. This allows us to reduce the problem of arbitrary property checking to that of reachability. In Figure 2, we are interested in detecting if y is a conditionally relevant variable w.r.t. the 'return output' statement (the answer is 'No') using formal verification. For this, we insert instrumentation code at strategic points in the original program (In Figure 2) to give us the instrumented code as shown in Figure 3. On this instrumented code, the simplified verification question now becomes "Is the 'error' statement reachable?" Although, this kind of reachability questions are undecidable in general, this has been addressed earlier in literature using software model checking. 'How to automatically instrument the code to simplify an arbitrary property checking question to a reachability problem?' is one of the central questions we intend to address through this work. The instrumentation in Figure 3 specifically targets the variable 'y' because the focus of the analysis is on determining whether 'y' is conditionally relevant with respect to the safety property. Based on manual analysis of the code 'y' is instrumented. In general all variables need to be investigated.

The algorithm primarily focuses on individual variables to assess their conditional relevance based on safety conditions. However, if there are dependencies between input variables, the algorithm should take these dependencies into account during the static analysis and formal verification steps. Algorithm need to be designed to scalable and efficient, even when dealing with predicates involving multiple variables. The static analysis and formal verification steps need to be optimized to handle complex conditions without causing a significant increase in the number of instrumentation passes.

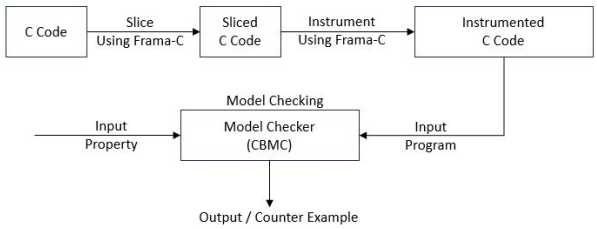

Figure 4: Flow Diagram

4.4 Program Verification

As mentioned earlier, unfortunately, computing precise static slices is undecidable; so is identifying conditionally (ir)relevant variables. Using formal verification on top of static slicing to identify conditionally (ir)relevant variables. We Instrument the code and solve reachability problem using appropriate formal verification technique(s).

5 EXPERIMENTAL RESULTS

5.1 Experimental Setup

The test-bed configuration for our proposed solution is elucidated by a flow diagram seen in Figure 4. In the conducted experiment, the C programming snippet depicted in Figure 2 was taken into consideration. The C code sample is sliced using the framework provided by Frama-C. Frama-C facilitates [2] the integration of new plug-ins into its platform, hence assisting developers in this process. The software framework described herein facilitates the creation of static analysis tools for C programs. This platform also offers support for program slicing. Frama-C provides support for generating Abstract Syntax Trees (ASTs), Program Dependence Graphs (PDGs), and traversal functions for the purpose of code instrumentation.

The output variable is used as a slicing criterion, and the C code is sliced based on that value. When output was greater than 10, this failed to determine that y is not a CRV. The following phase involves instrumenting the sliced code at various strategic points. For a simple understanding these strategic points are flag variable that are set to true or false based on manual investigation of the code. The flag is set to true when y is CRV based on safety property. Additionally, a safety property check has been incorporated, accompanied by an assert statement that is suitable for the CBMC model checker to verify the aforementioned property. The CBMC tool accepts a Boolean condition as input, and subsequently use the model checker to ascertain the truth value of the condition for every possible execution of the program. CBMC considers both control and data dependencies when determining the relevance of variables with respect to the slicing criteria. Now the simple question to the model checker is 'output > 10 && flag == true' condition is true for any runs of the program execution, the answer is "No". This proves that y is not a CRV when program violate the safety condition.

5.2 Design of the Algorithm

The main goal of the code instrumentation algorithm is to decide, when the program reaches faulty state, what are the program statements need to be included or removed. The code instrumentation

in this paper is described based on adding program statements at strategic points. This is done manually. Final goal is to write an algorithm which preforms instrumentation automatically. Some of the steps which are taken into consideration while adding flags are described below.

Here, 'y' is the investigating variable and 'output' is a property variable. In this context, the term 'investigating variable' refers to the variable 'y,' which is currently under examination to determine its conditional relevance in the given iteration of the algorithm. The investigation aims to ascertain whether 'y' is conditionally relevant or not based on safety property considerations. The algorithm iteratively analyzes each input variable individually to determine its conditional relevance. For each variable, the algorithm adds instrumentation in the code at strategic points, such as introducing flag variables, to mark conditions that contribute to the conditional relevance assessment. This process is performed systematically for each input variable, allowing the algorithm to identify conditionally relevant parts of the program accurately.

Instrumentation involves the strategic placement of flags within the program code to mark the points where a variable, relevant to the safety criterion, undergoes modification. Specifically, we introduce a 'flag' variable that is set to 'true' or 'false' based on the conditional relevance of the investigated variable. This instrumentation enables us to identify and track the influence of variables on safety conditions during program execution. To address the question of automation, our algorithm need to systematically analyzes the control and data dependencies in the program, to identify the set of conditionally relevant variables (CRVs). Subsequently, the algorithm automatically places flags at points where these CRVs affect the program's execution paths. The goal is to streamline the process of instrumentation, reducing the need for manual intervention.

The term "modification" in the algorithm refers specifically to instances where the value of a variable is changed in a manner that may impact the safety property of interest. We acknowledge that every definition in the program involves some form of modification, but the algorithm focuses on identifying modifications that are conditionally relevant concerning the safety property. The algorithm considers modifications that alter the value of a variable based on conditional dependencies. In the examples provided (output=2 at Line 8 and output=1 at Line 11 in Fig. 2), these modifications are conditionally relevant because they occur within branches where the safety property may be violated.

The purpose of the algorithm evaluation plan is to showcase the efficacy of the technique in various control algorithms and safety scenarios. The evaluation will measure the extent to which CBMC reduces the occurrence of false positives in compared to standard static analysis. Although there may not be standardised benchmark suites specifically tailored for this particular environment, we will make efforts to select representative program's. The review approach we have developed is aimed to be comprehensive, covering a wide range of scenarios. We will incorporate a variety of control algorithms, safety criteria, and benchmarks to ensure the technique's robustness and efficiency under different situations.

6 CONCLUSION

Some variables are important w.r.t. some criterion; these need to be safeguarded against SEUs. Static slicing is sound, but too conservative. Program analysis and verification to detect irrelevant variables that are missed by static slicing leading to sound but more precise estimates. Algorithm has been designed and tried on a motivating example. A prototype has been implemented using LLVM framework and has been used to detect conditionally relevant variables.

7 FUTURE PLAN

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

- Identification of larger case studies.
- Implementation of a non-trivial proof-of-concept prototype.
- Experimental evaluation.

We believe that issues that can be detected using an experimental fault injection simulation can also be detected using our static analysis based approach. Further, while we have restricted our attention to variables in terms of their conditional relevance, the idea can be extended to the verification of conditional relevance of program locations too. We would like to clarify that, as of the current version, we have presented an outline of the correctness argument and the key aspects that need to be addressed in the proof. Our next steps include providing a more formal and detailed proof of correctness, explicitly addressing the verification challenges and establishing the soundness of our technique. We will incorporate mathematical formulations and additional explanatory content to substantiate the correctness of our proposed approach.

REFERENCES

[1] B. Chimdyalwar, P. Darke, A. Chavda, S. Vaghani, and A. Chauhan. 2015. Eliminating static analysis false positives using loop abstraction and bounded model checking. , 573-576 pages.
[2] Frama-C 2023. *Frama-C Plugin Development Guide*. Frama-C. https://frama-c.com/html/documentation.html.
[3] A. Kaiser H. Post, C. Sinz and T. Gorges. 2008. Reducing False Positives by Combining Abstract Interpretation and Bounded Model Checking. *23rd IEEE/ACM International Conference on Automated Software Engineering* 15 (April 2008), 2–5.
[4] David John. 2014. Method and Apparatus for Soft Error Mitigation in Computers. https://patents.google.com/patent/US20150234693A1/en Patent No.US20150234693A1, Filed Feb 25th., 2001, Issued January. 12th., 206.
[5] Y. Monne. 2005. Hardening techniques against transient faults for asynchronous circuits.
[6] T. Muske, M. Datar, A.and Khanzode, and K. Madhukar. 2013. Efficient elimination of false positives using bounded model checking. *ISSRE* 15 (April 2013), 2–5.
[7] E. Normand. 1996. Single-event effects in avionics. *IEEE Transactions on Nuclear Science*. 43 (April 1996), 461 – 474.
[8] E. Normand. 1996. Single event upset at ground level. *IEEE Transactions on Nuclear Science*. 43 (Dec. 1996), 2742 – 2750.
[9] Sujan Pandey. 2015. Data error susceptible bit identification. https://patents.google.com/patent/US20150074631A1/en Patent No. US20150074631A1, Filed November 25th., 2013, Issued December. 15th., 2015.