electronics

MDPI

Article

F_Radish: Enhancing Silent Data Corruption Detection for Aerospace-Based Computing

Na Yang [1,2,*] and Yun Wang [1,2,*]

1 School of Computer Science and Engineering, Southeast University, Nanjing 210000, China
2 Key Laboratory of Computer Network and Information Integration, Ministry of Education, Southeast University, Nanjing 210000, China
* Correspondence: nyangsd@seu.edu.cn (N.Y.); ywang_cse@seu.edu.cn (Y.W.)

Abstract: Radiation-induced soft errors degrade the reliability of aerospace-based computing. Silent data corruption (SDC) is the most dangerous and insidious type of soft error result. To detect SDC, program invariant assertions are used to harden programs. However, there exist redundant assertions in hardened programs, which impairs the detection efficiency. Benign errors are another type of soft error result. An assertion may detect benign errors, incurring unnecessary recovery overhead. The detection degree of an assertion represents the detection capability, and an assertion with a high detection degree can detect severe errors. To improve the detection efficiency and detection degree while reducing the benign detection ratio, F_Radish is proposed in the present work to screen redundant assertions in a novel way. At a program point, the detection degree and benign detection ratio are considered to evaluate the importance of the assertions in the program point. As a result, only the most important assertion remains in the program point. Moreover, the redundancy degree is considered to screen redundant assertions for neighbouring program points. Experimental results show that in comparison with the Radish approach, the detection efficiency of F_Radish is about two times greater. Moreover, F_Radish reduces the benign detection ratio and improves the detection degree. It can avoid more unnecessary recovery overheads and detect more serious SDC than can Radish.

Keywords: soft error; single event upset; silent data corruption; reliability; redundant assertion

check for updates

Citation: Yang, N.; Wang, Y. F_Radish: Enhancing Silent Data Corruption Detection for Aerospace-Based Computing. *Electronics* **2021**, *10*, 61. https://doi.org/10.3390/electronics10010061

Received: 31 October 2020
Accepted: 24 December 2020
Published: 31 December 2020

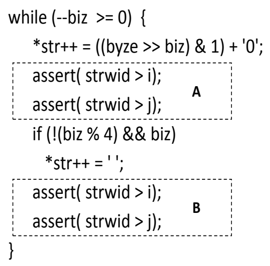
1. Introduction

Soft errors are transient errors caused by single event effects that occur in microelectronics when highly energetic particles, such as protons, electrons, and neutrons, strike sensitive regions of a microelectronic circuit [1]. Neutron-induced soft errors were observed in airborne computers in 1993 [2]. Today, with the aggressive shrinking of nodes in microelectronic devices and the reduction in supply voltages, the energy threshold for causing soft errors has decreased rapidly, resulting in the increase of soft errors and causing them to become a chief reliability threat [3]. Soft errors have become a critical reliability concern for applications, and their resulting forms can be categorised as crash, hang, benign, and silent data corruption (SDC) errors. Crash and hang are explicit errors that cause programs to respectively stop execution and to run non-stop, and they can be easily captured by those explicit behaviours. Being benign means that an error is masked during the program execution and does not have an effect on the output of the program [4]. SDC means that an error does not incur explicit behaviours; however, an incorrect result is produced after the program finishes. SDC is very difficult to detect and can therefore have severe consequences [5–7].

Soft error detection is the first and crucial step of soft error protection, and is conducted at both the software and hardware levels. Hardware-based approaches usually change the original processor architecture or attach special-purpose hardware modules to the processor. However, they require substantial development efforts, and the hardware modules are barely portable. Moreover, as the soft error rates increase, hardware may not provide adequate protection [8]. In contrast, software-based approaches require no hardware and provide high portability and a short development time, and are therefore promising.

An SDC is hazardous because it corrupts the execution result of a program without any explicit behaviour. The instruction duplication mechanism [9–11], prediction-based mechanism [12,13], and assertion-based mechanism [14,15] are three types of mechanisms for the detection of SDC at the software level. Via the instruction duplication mechanism, instructions are duplicated and their results are compared to detect SDC. However, this method is expensive because it involves the replication of a large portion of a program [14]. The prediction-based mechanism generates a predicted value at runtime via a prediction method [12], and the predicted and actual observed values are then compared to detect SDC. This mechanism is applicable to iterative HPC applications or parallel applications [13,16], and its process is complex. The assertion-based detection mechanism detects SDC at the program level by inserting assertions into programs [14], and its overhead is low. Moreover, its process is relatively simple, and it generally does not target specific types of applications.

An assertion is a statement with a predicate (A Boolean-valued function, a true-false expression). If it is found to be false at runtime, an assertion failure occurs. The predicate also represents an invariant that is true during normal program execution. At present, the extracted invariants can be classified as logic-based invariants with multiple variables and bounded-range-based invariants with a single variable, and the former outperforms the latter. An SDC detection approach called Radish was proposed in the authors' previous research [14]. In Radish, a program invariant describes the relationship of the variables that appear in the program point, such as global variables, and the local variables and function parameters of the function in which the program point is located. Figure 1 provides an example of program invariant assertions in "bitstring", which is a program in the Mibench benchmark suite that prints a bit pattern of bytes formatted as a string. In the figure, "*strwid*" represents the total width of the string, and $strwid > i$ and $strwid > j$ are two program invariants at program point A that respectively describe the relationship between *strwid* and i and that between *strwid* and j. They are satisfied whenever the program is executed normally, but are seldom satisfied if a soft error affects the value of *strwid* or i or j. Radish performs well and is outstanding in that it extracts logic-based invariants with multiple variables. It also screens invariant assertions to improve the detection efficiency. However, the following challenges have been identified.



```
while (--biz  >= 0) {
    *str++ = ((byze >> biz) & 1) + '0';
    assert( strwid > i);
    assert( strwid > j);
    if (!(biz % 4) && biz)
      *str++ = ' ';
    assert( strwid > i);
    assert( strwid > j);
}
```

Figure 1. An example of assertions.

(1) Redundant assertions impair the detection efficiency

Because multiple variables appear in the program point and multiple relationships are considered, multiple assertions are produced in the program point. The detection efficiency is the ratio between the SDC coverage and detection overhead. To improve the detection efficiency, Radish reduces the detection overhead by assertion screening based on the features of assertions. However, there may still exist multiple assertions at a program point, as assertions may behave with the same features. For example, in Figure 1, the program point A still has multiple assertions. Additionally, neighbouring program points may have the same or similar assertions because they are likely to have several of the same variables. For example, A and B are neighbouring program points that have the same assertions. However, Radish does not screen redundant assertions in neighbouring program points. In summary, multiple assertions in a program point and the same or similar assertions in neighbouring program points will incur more detection overhead and further impair the detection efficiency. Therefore, it is necessary to further filter out these redundant assertions.

(2) The detection degree and benign detection ratio are not considered in the process of assertion selection.

The result of a program may consist of multiple outputs that have different importance to the result and will cause various damage to the result when they are corrupted. For example, a program performs operations to get a result. The result represents a radian and consists of two outputs, namely the integral and fractional parts of the radian. The integral part is more important than the fractional part because an incorrect integral part will cause more damage to the result. Different SDCs corrupt different variables and program outputs, causing various damage to the result of the program. The detection degree of an assertion is considered as the damage of the SDC detected by the assertion to the result of the program. For example, suppose that there are two assertions, $assert(x > 10)$ and $assert(y > 0)$, where x and y, respectively, represent the integral and fractional parts of the radian. The detection degree of the former assertion is higher than that of the latter assertion, as the SDC detected by the former assertion causes more damage to the result of the program than that detected by the latter assertion. It is therefore significant to take the detection degree into consideration and to favour the assertions with high detection degree. Additionally, assertions may detect benign errors, and a higher benign detection ratio means more unnecessary recovery overhead. Therefore, disfavoring the assertions with high benign detection ratios is essential. However, Radish does not consider the detection degree or benign detection ratio in its process of assertion selection.

In this paper, F_Radish is proposed to detect SDC with a high detection efficiency, a high detection degree, and a low benign detection ratio. The main contributions of this research are summarised as follows.

(1) The detection degree and benign detection ratio of an assertion are considered during the process of assertion screening. For a program point, the importance of each of its assertions is evaluated based on the detection degree and benign detection ratio. As a result, only the most important assertion remains in the program point.
(2) Redundant assertions in neighbouring program points are handled. The redundancy degree of an assertion with respect to its neighbouring assertion is calculated. If the redundancy degree exceeds a specified threshold, the gain and loss of deleting the assertion are evaluated. When there is a profit, the assertion is deleted.
(3) An evaluation of F_Radish is conducted. Compared to Radish, the SDC detection efficiency of F_Radish is about two times greater. Moreover, the percentage increase of the detection degree is 10%. In addition, F_Radish reduces the benign detection ratio from 27.8% to 19.2%.

The remainder of this paper is organised as follows. Section 2 briefly reviews related work. The overview of the proposed F_Radish approach is provided in Section 3. Section 4 presents the process of F_Radish, and explains how redundant assertions in program points and neighbouring program points are screened. An experimental analysis is reported in Section 5. Finally, Section 6 draws the conclusions and discusses future work.

2. Related Work

There is a substantial amount of literature on the detection of soft errors at the software level. SWIFT [9] is an instruction duplication approach that duplicates instructions and inserts comparison instructions at detection points. During program execution, if there is a divergence between the original and duplicated instructions, an error is detected. S-SWIFT-R [17] is a flexible version of SWIFT that selects different register subsets from the microprocessor register file to be duplicated. NEMESIS [10] also detects soft errors at the compiler level. To reduce the detection overhead, it checks the results, rather than the operands, of instructions. The research by Rehman et al. [18] found that different functions are not equally susceptible to soft errors due to their varying data flow and control flow properties. To avoid excessive protection, only the most reliability-wise important instructions, which are evaluated by the masking probability, vulnerability and redundancy overhead, are protected. SDSC [11] is a novel data flow error detection technique that protects the blocks in the longest path of the control flow graph via instruction duplication, and inserts comparison instructions only in the critical blocks that have two or more incoming edges in the longest path of the control flow graph. While various efforts have been made to refine the duplication space, the overhead of the instruction duplication mechanism remains quite high.

Targeted at iterative HPC applications, the research work of [12,16] detected SDC by comparing the observed value at runtime with the predicted value obtained via methods such as the acceleration-based predictor (ABP), linear curve fitting (LCF), and quadratic curve fitting (QCF). Mutlu et al. [19] developed a machine learning-based predictor to generate ground-truth results in the presence of errors. The predictor aims at accelerating SDC detectors and targeting iterative solvers. LADR [20] was proposed to protect applications from SDC at the application level by identifying and monitoring manually selected sink variables. Sirius [13] is a technique for the detection of SDC using the temporal and spatial locality of physical variables in parallel applications. It constructs a neural network model for each spatiotemporal variable, and the model is used to check if the actual observed value of the variable falls within a bounded range around the predicted value. However, the processes of prediction-based detection approaches are complex, and the selection of the protected variable is not fully automated.

In the assertion-based mechanism, an assertion is used as a detector. If the assertion fails at runtime, it indicates that an error has occurred. iSWAT [15] and FaultScreening [21] adopt bounded-range-based invariant assertions. iSWAT applies soft-level symptoms at the firmware level to detect permanent faults and invariant assertions at the program level to detect transient faults. However, its invariants are produced based on the range of valid values of a single variable. FaultScreening narrows down the valid value space to improve fault coverage by dynamically dividing the range of valid values into resizable segments. Although it refines the valid value space, its invariants are also generated based on single variable values. LPD [22] generates assertions by identifying a few program properties. However, the process of deriving assertions is manual, which incurs a high demand of application-specific knowledge. Those assertion-based approaches do not conduct the work of screening assertions. In addition to bounded-range-based invariants with a single variable, logic-based invariants with multiple variables are another form of invariants, and are generated by considering the relationships among multiple variables. In general, logic-based invariants with multiple variables outperform bounded-range-based invariants with a single variable. For example, consider two variables in a program, p and q. During program execution, the sum of p and q is 10, and p and q are both less than 10. In this case, the assertions generated by single-variable-based methods are in the form of $assert(p < 10)$ and $assert(q < 10)$. However, the assertion generated by multiple-variables-based methods is in the form of $assert(p + q = 10)$. $assert(p + q = 10)$ outperforms $assert(p < 10)$ and $assert(q < 10)$, as its detection overhead is less than the total detection overhead of $assert(p < 10)$ and $assert(q < 10)$. Additionally, it can detect more SDC.

Radish [14] automatically extracts logic-based invariant assertions with multiple variables to detect SDC. It includes three phases, namely the preprocessing, detection, and selection phases. In the preprocessing phase, Radish identifies critical program points and extracts their execution profiles. The critical program points are connector and branch instructions that work against data flow propagation and control flow propagation, respectively. The execution profiles refer to data trace files of variables and their values that manifest at the critical program points, and they are obtained by Kvasir [23]. In the detection phase, at the critical program points, the values of the variables in the data trace files are utilised to generate invariants by checking whether the values satisfy any relationship considered by Radish, such as unary and binary relationships. The relationships that are satisfied are potential invariants. In the selection phase, the potential invariants are selected by heuristics and the final invariant assertions are generated. The process of invariant selection is also that of assertion selection. Radish is simple and the variables that are protected are identified automatically. It achieves a high detection efficiency and screens assertions. Radish_D [14] protects the code sections that are not covered by Radish via the instruction duplication mechanism. In this manner, Radish_D detects SDC at the program level via assertions and SDC at the instruction level via instruction duplication. At the program level, the work of Radish_D is the same as that of Radish, Radish_D is Radish. At present, in terms of the existing assertion-based detection approaches at the program level, Radish is outstanding in the detection of SDC, and it screens assertions. However, two factors have been identified as requiring improvement. First, although Radish screens assertions, redundant assertions still exist and require further screening to improve the detection efficiency. Second, the detection degree and benign detection ratio need to be considered during the process of assertion screening to detect severe SDC and reduce unnecessary recovery overhead.

3. Overview of the F_Radish Approach

F_Radish contains two stages, namely the screening of assertions for every program point and the screening of assertions for neighbouring program points. Note that program points in the two stages refer to those that have assertions. The overview of F_Radish is presented in Figure 2.

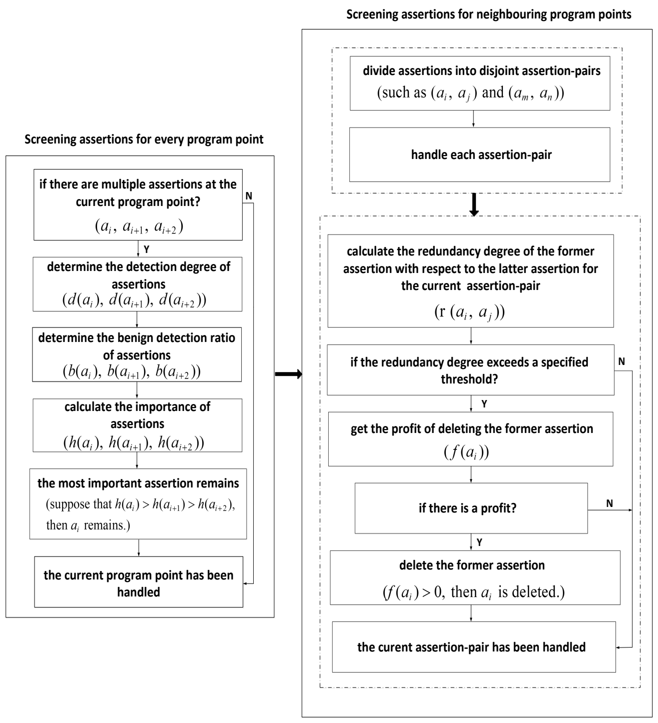

Screening assertions for every program point
if there are multiple assertions at the current program point?
(a_i, a_{i+1}, a_{i+2})
N
Y
determine the detection degree of assertions
$(d(a_i), d(a_{i+1}), d(a_{i+2}))$
determine the benign detection ratio of assertions
$(b(a_i), b(a_{i+1}), b(a_{i+2}))$
calculate the importance of assertions
$(h(a_i), h(a_{i+1}), h(a_{i+2}))$
the most important assertion remains
(suppose that $h(a_i) > h(a_{i+1}) > h(a_{i+2})$, then a_i remains.)
the current program point has been handled
Screening assertions for neighbouring program points
divide assertions into disjoint assertion-pairs
(such as (a_i, a_j) and (a_m, a_n))
handle each assertion-pair
calculate the redundancy degree of the former assertion with respect to the latter assertion for the current assertion-pair
$(\mathrm{r}\,(a_i, a_j))$
if the redundancy degree exceeds a specified threshold?
get the profit of deleting the former assertion
$(f(a_i))$
if there is a profit?
delete the former assertion
($f(a_i) > 0$, then a_i is deleted.)
the curent assertion-pair has been handled

Figure 2. The overview of the F_Radish approach.

Screening assertions for every program point: This stage handles every program point of the hardened program. If a program point does not have multiple assertions, it is skipped and considered to have been handled, and the next program point is then handled. Otherwise, the following steps are executed for the program point. Assume that the program point has three assertions, namely a_i, a_{i+1}, and a_{i+2}. The detection degree and benign detection ratio of a_i, a_{i+1}, and a_{i+2} are first determined, and are then utilised to assess the importance of a_i, a_{i+1}, and a_{i+2}. Finally, only the most important assertion remains. Under the assumption that the importance of a_i is the greatest, a_i remains, and a_{i+1} and a_{i+2} are deleted. This stage ends when all program points have been handled.

Screening assertions for neighbouring program points: After the first stage, there is only one assertion in every program point. In the second stage, assertions are screened for neighbouring program points. As preparation, the assertions in the program are divided into multiple disjoint assertion-pairs based on their execution order and the functions that they belong to. For example, suppose that a_i, a_j, a_m, and a_n are four assertions in the program after the first stage. They are executed sequentially and belong to the same function. In this case, two assertion-pairs, (a_i, a_j) and (a_m, a_n), are generated. After preparation, each assertion-pair is handled. The assertions of an assertion-pair are screened by determining whether its former assertion can be deleted. To be specific, the redundancy degree of the former assertion with respect to the latter assertion is first calculated. If the redundancy degree does not exceed a specified threshold, the former assertion is not deleted. Otherwise, the profit of deleting the former assertion is calculated. If there is a profit, the former assertion is deleted or else it is not deleted. This stage ends when all assertion-pairs have been handled.

An example is subsequently provided to explain the general process of F_Radish. Assume that there is a program with 1000 program points, and 100 program points have one or more assertions. In this case, F_Radish will screen assertions for each of the 100 program points during the first stage. After this stage, only one assertion remains in each of the 100 program points, and the total number of assertions remaining in the program is therefore 100. In the second stage, F_Radish screens assertions for neighbouring program points. First, the 100 assertions are divided into disjoint assertion-pairs. Assume that 50 disjoint assertion-pairs are generated based on the execution order of the 100 assertions and the functions to which the 100 assertions belong. Then, each of the 50 assertion-pairs is handled. For each assertion-pair, whether its former assertion can be deleted is evaluated and determined. If the former assertion can be deleted, the former assertion is deleted; otherwise, it is not deleted. As a result, at least 50 assertions remain in the program.

4. The Stages of F_Radish

The two stages of F_Radish are detailed in Sections 4.1 and 4.2, respectively. The notations that are frequently used in this section are presented in Table 1.

4.1. Screening Assertions for Every Program Point

(1) Determining the benign detection ratio of assertions

The benign detection ratio of a_i is the probability that a benign error that corrupts V_{a_i} can be detected by a_i. To determine the benign detection ratio of a_i, the instructions that operate V_{a_i} at a_i are first obtained, and the backward slice set of these instructions is then generated. Next, fault injection is conducted on the backward slice set. Finally, the benign detection ratio of a_i is obtained by analysing the result of fault injection, and is represented by Equation (1).

$$b(a_i) = \frac{n_1(a_i)}{n_2(a_i)} \tag{1}$$

The backward slice set of an instruction contains the instructions that will influence the values of the instruction. To obtain the backward slice set, the dynamic dependence graph is first constructed. It is a directed acyclic graph that is defined as $G = (V, E)$, where V is the set of instruction nodes and E is the set of edges. If instruction i_2 reads a value produced by instruction i_1, then the edge $i_1 \rightarrow i_2$ is produced. After obtaining the dependence graph, reverse path-searching is performed to obtain the backward slice set. Figure 3 presents an example of a program code that calculates the sum of the integers from 0 to $size - 1$ and returns the sum. Figure 4 exhibits the dynamic dependency graph with a $size$ of 2. In Figure 4, the nodes are placed based on the variable that is written. For example, the nodes in the first column all write k. Table 2 provides the corresponding instructions and nodes, where PR and PW are the positions that are read and written by the instruction, respectively. Node id refers to the node in Figure 4. Take node 7 as an example, it represents $add\ dword\ ptr\ [esp + 0x24],\ eax$, which reads eax and $[esp + 0x24]$ and writes $[esp + 0x24]$. The father nodes of node 7 are node 6, which writes eax, and node 1, which writes $[esp + 0x24]$. The child node of node 7 is node 13, which reads $[esp + 0x24]$. Via a reverse path search, the backward slice set of node 7 is 1, 6, and 2.

Table 1. Notations.

Symbol	Description
O	The output set of the program, $O = \{o_1, o_2, \cdots, o_c\}$.
$w(o_j)$	The weight of o_j, $1 \leq j \leq c$.
V_{a_i}	The variable set of a_i, $V_{a_i} = \{V_{a_i}^1, V_{a_i}^2, \cdots, V_{a_i}^m\}$.
$fs(V_{a_i}^l)$	The forward slice set of $V_{a_i}^l$ at a_i, $1 \leq l \leq m$.
$s(fs(V_{a_i}^l))$	The size of $fs(V_{a_i}^l)$.
$fs(V_{a_i}^l, k)$	The k-th element of $fs(V_{a_i}^l), 1 \leq k \leq s(fs(V_{a_i}^l))$.
P	The initial hardened program.
FP	The filtered program of P after the first stage of F_Radish.
SP	The filtered program of FP after the two stages of F_Radish.
ps	The set of program points of P.
p	The p-th program point in ps. It is also called program point p for convenience.
$A(p)$	The set of assertions at p.
$a_{p,q}$	The q-th assertion at p.
$V_{a_{p,q}}$	The variable set of $a_{p,q}$.
$V_{a_{p,q}}^l$	The l-th element of $V_{a_{p,q}}$.
$fs(V_{a_{p,q}}^l)$	The forward slice set of $V_{a_{p,q}}^l$ at $a_{p,q}$.
$u(V_{a_{p,q}})$	The set of instructions that operate one or more variables in $V_{a_{p,q}}$ at $a_{p,q}$.
$bs(u(V_{a_{p,q}}))$	The backward slice set of the instructions in $u(V_{a_{p,q}})$.
$t_1(a_i)$	The number of fault injections that incur SDC and invalidate a_i.
$t_2(a_i, a_j)$	The number of fault injections that not only result in SDC but also invalidate a_i and a_j.
θ	The threshold of redundancy degree, $0 \leq \theta \leq 1$.
ap	The set of assertion-pairs of FP.
$bi(V_{a_i}^l)$	The backward slice set of instructions that operate $V_{a_i}^l$ at a_i.
$n_1(a_i)$	The number of fault injections that are injected on the backward slice set of the instructions that operate V_{a_i} at a_i, and not only result in benign error but also are detected by a_i.
$n_2(a_i)$	The number of fault injections that are injected on the backward slice set of the instructions that operate V_{a_i} at a_i and result in benign error.
α	The weight of the detection degree.
β	The weight of the benign detection ratio.
max_d	The maximum detection degree.
max_b	The maximum benign detection ratio.
$d(a_i, a_j)$	The instructions between assertions a_i and a_j.

Table 1. *Cont.*

Symbol	Description
x	The instruction number of the first instruction in $d(a_i, a_j)$.
$n(a_i, a_j)$	The number of the instructions in $d(a_i, a_j)$.
$sr(a_i)$	The SDC detection ratio of a_i.
$dt(i_k)$	The execution times of instruction i_k.
y	The instruction number of the first instruction of a_i.
z	The instruction number of the last instruction of a_i.

```
int sum (int size){
  k=0;
  for (int i=0;i<size;i++)
  {
    k=k+i;
  }
  return k;
}
```

Figure 3. A program code.

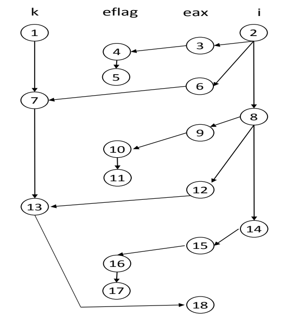


Figure 4. The dynamic dependency graph with a *size* of 2.

Table 2. Instructions and nodes.

Source Code	Instruction	PR	PW	Node ID
k = 0	mov dword ptr [esp+0x24], 0x0	-	[esp + 0x24]	1
i = 0	mov dword ptr [esp + 0x20], 0x0	-	[esp + 0x20]	2
	mov eax, dword ptr [esp + 0x20]	[esp + 0x20]	eax	3, 9, 15
i < size	cmp eax, dword ptr [esp – 0x8]	eax, [esp + 0x1c]	eflag	4, 10, 16
	jl 0x80486da	eflag	eip	5, 11, 17
k = k + i	mov eax, dword ptr [esp + 0x20]	[esp + 0x20]	eax	6, 12
	add dword ptr [esp + 0x24], eax	eax, [esp + 0x24]	[esp + 0x24]	7, 13
i++	add dword ptr [esp + 0x20], 0x1	[esp + 0x20]	[esp + 0x20]	8, 14
return k	mov eax, dword ptr [esp + 0x24]	[esp + 0x24]	eax	18

(2) Determining the detection degree of assertions

The detection degree of an assertion is considered as the damage of the SDC detected by the assertion to the result of the program, and reflects the detection capability. When the SDC detected by an assertion causes more damage to the result of the program, its detection degree is higher. In this section, an assertion called a_i is taken as an example to present how to determine its detection degree.

Assume that the result of the program is an output set called O that consists of c outputs with different weights. The greater the weight of an output, the more damage to the result of the program the output will cause when it is corrupted. V_{a_i} is the variable set of a_i. Because the SDC detected by a_i damages O by corrupting V_{a_i}, the detection degree of a_i can approximately be considered as the damage of corrupted V_{a_i} to O. For the l-th element of V_{a_i} at a_i, namely $V_{a_i}^l$, the total weights of the output variables in the forward slice set of $V_{a_i}^l$ are considered as the damage of corrupted $V_{a_i}^l$ to O. Then, the damage of corrupted $V_{a_i}^l$ to O can be represented by Equation (2), where the value of $e(o_j, fs(V_{a_i}^l, k))$ is set to 1 when $fs(V_{a_i}^l, k)$ is o_j, otherwise, it is set to 0. Because soft errors are rare relative to the execution time of typical programs, it is assumed that at most one fault occurs during one program execution. This assumption is in line with previous work in [14,24,25]. Under this assumption, the detection degree of a_i is considered as the averaged z across all variables of a_i, and is expressed by Equation (3).

$$z(V_{a_i}^l) = \sum_{k=1}^{s(fs(V_{a_i}^l))} (\sum_{j=1}^{c} (e(o_j, fs(V_{a_i}^l, k)) \times w(o_j)) \tag{2}$$

$$d(a_i) = \frac{\sum_{l=1}^{m} z(V_{a_i}^l)}{m} \tag{3}$$

The forward slice of a variable in a program statement is the set of the variables in which the values will be influenced by that variable during program execution. The forward slice is obtained in a similar way as the acquisition of the backward slice. In particular, the program statement is first compiled to instructions as preparation, and forward path-searching is applied. Herein, $assert(x > 10)$ and $assert(y > 0)$ described in Section 1 are taken as an example to further describe the detection degree. The two assertions are called a_1 and a_2. The output set of the program is O, $O = \{x, y\}$. Because x represents the integral part of the radian and is more important than y, $w(x)$ and $w(y)$ can be set to 0.7 and 0.3, respectively. For a_1 and a_2, assume that $fs(x) = \{x, y, z\}$ and $fs(y) = \{y\}$ after analysing the forward slices, where z is a variable in the program and does not represent an output. In this case, the damage of corrupted x in a_1 to O is 1, which is the sum of $w(x)$ and $w(y)$ because $fs(x)$ contains x and y. Because the SDC detected by a_1 damage O by corrupting x, the detection degree of a_1 is 1, namely, $d(a_1) = 1$. In a similar way, $d(a_2) = 0.3$.

(3) Calculating the importance of assertions

Because an assertion with a high detection degree and a low benign detection ratio is preferred, the importance of a_i is represented by Equation (4), where α and β, respectively, represent the weights of the detection degree and benign detection ratio. They are set according to preference, and their sum is 1. If more attention is paid to the detection of severe errors than the avoidance of unnecessary recovery overhead, α can be set as having a larger value than β.

$$h(a_i) = \alpha \times \frac{d(a_i)}{max\_d} - \beta \times \frac{b(a_i)}{max\_b} \tag{4}$$

After determining the importance of all the assertions of a program point, only the most important assertion remains at this program point, and other assertions are deleted. The process of screening assertions for every program point is presented in Algorithm 1. A detailed explanation is subsequently provided.

If a program point has only one assertion, it is skipped, and the next program point is handled (Line 4–5), otherwise, the following steps are executed for it. For every assertion in the program point, the damage of its corrupted variables to the output of the program is first evaluated (Line 11–15), and its detection degree can then be obtained (Line 16). After this, the backward slices of the instructions, which operate one or more variables of the assertion at the assertion, are generated, and fault injections are conducted on them (Line 22–24). Further, the result of fault injections is analysed to determine the benign detection ratio of the assertion (Line 25–26). After determining the detection degree and benign detection ratio of each assertion in the program point, the importance of each assertion is calculated. Finally, only the most important assertion remains, and the other assertions are deleted (Line 33–43). In particular, t aims to find the maximum value of h, and it records the current largest value of h of the assertions that have been traversed during the traversal process, which is conducted by a for loop (Line 34–42). It is initialised to a small value that is less than or equal to the minimum value of h. In Equation (4), all variables are all not less than 0. The minimum value of h is obtained when the first term is the minimum value and the second term is the maximum value. The minimum value of h is obtained when α is equal to 0 or $d(a_i)$ is equal to 0, and its minimum value is 0. The maximum value of the second term is obtained when β is equal to 1 and $b(a_i)$ is equal to max_b, and its maximum value is 1. Therefore, t is initialised to -1.

4.2. Screening Assertions for Neighbouring Program Points

For preparation, the assertions that remain after the first stage are divided into disjoint assertion-pairs based on the functions that they belong to and their execution order. More specifically, for every function, its assertions are first sorted according to their execution order, and the sorted assertions are then divided into disjoint assertion-pairs. Note that if there is an odd number of assertions in the function, the last-executed assertion will not be considered in this stage, as there is no extra assertion in the function with which it can compose an assertion-pair. Next, an example is provided to demonstrate how to generate assertion-pairs for FP, which is the filtered program after the first stage. Suppose that there are seven assertions and two functions (f_1 and f_2) in FP. In f_1, a_i, a_m, a_j, a_n, and a_k are executed in the order of a_i, a_j, a_m, a_n, a_k. In f_2, a_x and a_y are executed in the order of a_x, a_y. In this case, two assertion-pairs will be obtained in $f1$, namely (a_i, a_j) and (a_m, a_n). Similarly, (a_x, a_y) is obtained in f_2. As a result, three assertion-pairs are obtained for FP. After obtaining all assertion-pairs, each assertion-pair is operated to determine whether its former assertion can be deleted. In the following, (a_i, a_j) is taken as an example to explain how to determine whether a_i can be deleted.

(1) Calculating the redundancy degree of a_i with respect to a_j

$r(a_i, a_j)$ represents the redundancy degree of a_i with respect to a_j, and refers to the probability that the SDC detected by a_i can also be detected by a_j. An SDC invalidates a_i by corrupting the variable of a_i. For the variable called $V^l_{a_i}$ in a_i, the probability that the SDC incurred by the corrupted $V^l_{a_i}$ and detected by a_i can also be detected by a_j is determined by injecting faults into the backward slice set of the instructions that operate $V^l_{a_i}$ at a_i, and by analysing the result of fault injections. This is represented by Equation (5). Then, $r(a_i, a_j)$ is the averaged p across all variables of a_i, which is expressed by Equation (6).

$$p(V^l_{a_i}, a_j) = \frac{t_2(a_i, a_j)}{t_1(a_i)} \tag{5}$$

$$r(a_i, a_j) = \frac{\sum_{l=1}^{m} p(V^l_{a_i}, a_j)}{m} \tag{6}$$

If $r(a_i, a_j)$ exceeds a specified threshold, the profit of deleting a_i is further evaluated to determine whether a_i can be deleted, otherwise, a_i can not be deleted. Intuitively, $r(a_j, a_i)$ can be calculated, and the deletion of a_j can be considered. However, in general, $r(a_j, a_i)$ is less than $r(a_i, a_j)$, as there exists some errors that not only occur between a_i and a_j, but are also detected by a_j, cannot be detected when a_j is deleted. This results in a greater impairment to SDC coverage. Therefore, the deletion of a_i is considered.

Algorithm 1 Screening assertions for every program point.

```
Input: $P$
Output: $FP$
 1: get $ps$
 2: for $p = 1 \rightarrow size(ps)$ do
 3:     get $A(p)$
 4:     if $size(A(p)) == 1$ then
 5:         continue;
 6:     else
 7:         $max\_d = 0, max\_b = 0$
 8:         for $q = 1 \rightarrow size(A(p))$ do
 9:             get $a_{p,q}$ and $V_{a_{p,q}}$
10:             $d(a_{p,q}) = 0, sum\_z = 0$
11:             for $l = 1 \rightarrow size(V_{a_{p,q}})$ do
12:                 generate $fs(V^l_{a_{p,q}})$
13:                 get $z(V^l_{a_{p,q}})$ by (2)
14:                 $sum\_z = sum\_z + z(V^l_{a_{p,q}})$
15:             end for
16:             get $d(a_{p,q})$ by (3)
17:             if $d(a_{p,q}) > max\_d$ then
18:                 $max\_d = d(a_{p,q})$
19:             else
20:                 continue
21:             end if
22:             acquire $u(V_{a_{p,q}})$
23:             generate $bs(u(V_{a_{p,q}}))$
24:             conduct fault injections on $bs(u(V_{a_{p,q}}))$
25:             count $n_1(a_{p,q})$ and $n_2(a_{p,q})$
26:             get $b(a_{p,q})$ by (1)
27:             if $b(a_{p,q}) > max\_b$ then
28:                 $max\_b = b(a_{p,q})$
29:             else
30:                 continue
31:             end if
32:         end for
33:         $t = -1, m = 0$
34:         for $q = 1 \rightarrow size(A(p))$ do
35:             get $h(a_{p,q})$ by (4)
36:             if $h(a_{p,q}) > t$ then
37:                 $t = h(a_{p,q})$
38:                 $m = q$
39:             else
40:                 continue
41:             end if
42:         end for
43:         delete $a_{p,q}(q \neq m)$
44:     end if
45: end for
```

(2) Evaluating the profit of deleting a_i

The threshold of the redundancy degree is denoted by θ (in general, $\theta \geq 0.9$). Deleting a_i leads to two losses. On the one hand, $(1-\theta) \times 100$ percent of the SDC detected by a_i will not be detected, thereby impairing the SDC coverage of the hardened program. On the other hand, $\theta \times 100$ percent of the SDC detected by a_i will be detected by a_j with a delay, thereby incurring a delayed detection loss. In this paper, only the delayed detection loss is considered because $(1-\theta) \times 100$ percent is a small value.

Figure 5 illustrates the delayed detection loss of deleting a_i. In the figure, c represents a checkpoint, and $i_k (1 \leq k \leq 600)$ is an instruction. It should be noted that instructions are used in Figure 5 for the convenience of the following calculations. For an SDC called s, if it is detected by a_i, the program will roll back to c and continue to execute. Furthermore, the program will also roll back to c when s is detected by a_j because checkpoints are usually sparser than assertions. In comparison with the detection of s by a_i, the instructions between a_i and a_j, namely $d(a_i, a_j)$, are additionally executed when s is detected by a_j.

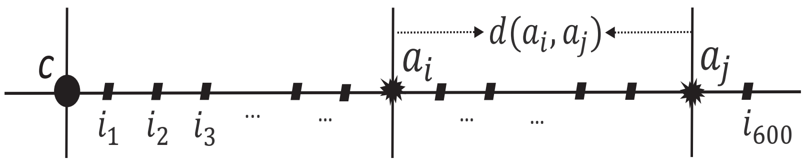


Figure 5. An example of deleting an assertion.

The total extra execution times of the instructions in $d(a_i, a_j)$ is considered as the loss of deleting a_i, as expressed by Equation (7). To determine $sr(a_i)$, fault injection is first conducted on the backward slice set of the instructions that operate V_{a_i} at a_i. Then, the result of fault injection is analysed. Finally, $sr(a_i)$ is considered as the ratio of the number of fault injections that not only incur SDC, but are also detected by a_i, and the number of fault injections that result in SDC.

$$l(a_i) = \sum_{k=x}^{x+n(a_i,a_j)-1} dt(i_k) \times sr(a_i) \tag{7}$$

Deleting a_i decreases the detection overhead. The total execution times of the instructions mapped by a_i is considered as the gain of deleting a_i, and is represented by Equation (8). Further, the profit of deleting a_i can be expressed by Equation (9). From Equation (9), it can be seen that there is a profit of deleting a_i when $f(a_i) > 0$. In this case, a_i is deleted.

$$g(a_i) = \sum_{k=y}^{z} dt(i_k) \tag{8}$$

$$f(a_i) = g(a_i) - l(a_i) \tag{9}$$

The process of screening assertions for neighbouring program points is presented in Algorithm 2, and a detailed explanation is provided as follows. First, assertion-pairs are obtained (Line 1). Then, for every assertion-pair, whether its former assertion can be deleted is evaluated. As the first step of the evaluation, for every variable in the former assertion, the probability that the SDC incurred by the corrupted value of the variable and detected by the former assertion can also be detected by the latter assertion is determined (Line 5–11). As the second step, the redundancy degree of the former assertion with respect to the latter assertion is calculated by averaging the probability across all variables in the former assertion (Line 12). In the subsequent steps, if the redundancy degree is less than a specified threshold, the former assertion cannot be deleted (Line 13–14), and the next assertion pair is handled, otherwise, the loss and gain of deleting the former assertion are further assessed to determine whether the former assertion can be deleted.

To determine the loss of deleting the former assertion, the SDC detection ratio of the former assertion is first obtained by fault injection (Line 16). The total execution times of the instructions between the former and latter assertions is then determined (Line 17–22). Finally, the loss of deleting the former assertion is represented by the product of the SDC detection ratio and the total execution times (Line 23). The gain of deleting the former assertion is determined by counting the total execution times of the instructions mapped by the former assertion (Line 24–29). After obtaining the loss and gain of deleting the former assertion, the profit is calculated (Line 30). If the profit is greater than 0, the former assertion is deleted, otherwise, the former assertion cannot be deleted (Line 31–35). This stage ends after all assertion-pairs have been handled.

Algorithm 2 Screening assertions for neighbouring program points.

```
Input: $FP$
Output: $SP$
 1: get $ap$ of $FP$
 2: for $s = 1 \to size(ap)$ do
 3:     $a_i = ap_s.former, a_j = ap_s.latter$
 4:     $z = 0, m = size(V_{a_i})$
 5:     for $l = 1 \to m$ do
 6:         generate $bi(V_{a_i}^l)$
 7:         conduct fault injections on $bi(V_{a_i}^l)$.
 8:         count $t_2(a_i, a_j)$ and $t_1(a_i)$
 9:         get $p(V_{a_i}^l, a_j)$ by (5)
10:         $z = z + p(V_{a_i}^l, a_j)$
11:     end for
12:     get $r(a_i, a_j)$ by (6)
13:     if $r(a_i, a_j) < \theta$ then
14:         continue
15:     else
16:         get $sr(a_i)$ by fault injection
17:         $t = 0$
18:         acquire $n(a_i, a_j)$ and $x$
19:         for $k = x \to x + n(a_i, a_j) - 1$ do
20:             get $dt(i_k)$
21:             $t = t + dt(i_k)$
22:         end for
23:         calculate $l(a_i)$ by (7)
24:         $g(a_i) = 0$
25:         get $y$ and $z$
26:         for $k = y \to z$ do
27:             acquire $dt(i_k)$
28:             $g(a_i) = g(a_i) + dt(i_k)$
29:         end for
30:         get $f(a_i)$ by (9)
31:         if $f(a_i) > 0$ then
32:             delete $a_i$
33:         else
34:             continue
35:         end if
36:     end if
37: end for
```

5. Experimental Analysis

The experimental setup is first presented in Section 5.1. Then, the experimental evaluation is provided in Section 5.2 to demonstrate the effectiveness of F_Radish. In the experimental evaluation, F_Radish is first compared with Radish in terms of the SDC coverage, detection overhead, detection efficiency, benign detection ratio, and detection degree. Then, different contributions of the two stages of F_Radish are evaluated.

5.1. Experimental Setup

(1) Fault injection.

A fault injection experiment was conducted to evaluate F_Radish. The fault injection experiment was first performed on the original program. The program hardened by Radish and the program hardened by F_Radish were subsequently targeted. Note that the program hardened by F_Radish refers to the program that was first hardened by Radish and then subjected to assertion filtering by the two stages of F_Radish. Moreover, to evaluate the different contributions of the two stages of F_Radish, faults were also injected to the program that was first hardened by Radish and its assertions were then filtered only by the first stage of F_Radish. Faults were injected into the operand of the instructions of programs. They were not injected into the opcode of the instructions, as this would result in illegal opcode exceptions rather than SDC or benign errors [26]. A single bit flip was considered, as this is widely considered in the study of soft errors [6,24] and Radish. In our fault injection campaign, fault injection was conducted by altering one bit in the register or memory cell from 0 to 1 or from 1 to 0. The result of fault injection was then compared with that of fault-free. If there was a divergence, it was considered as SDC, and if the two results were the same, it was considered as a benign error. The platform for experimental validation was a Dell Workstation with an i7 processor running Ubuntu 10.04. Pin is a dynamic binary instrumentation framework [27] that was used to create a dynamic instrument tool for carrying out the fault injection campaign.

(2) Benchmarks.

The programs used for experimental evaluation were sourced from Mibench and Siemens benchmark suites. These programs were bitstrng (which prints bit pattern of bytes formatted to string), rad2deg (which converts between radians and degrees), isqrt (which is a base-two analogue of the square root algorithm), and replace (which computes statistics over input data). To evaluate the detection degree, these programs were properly modified. For example, the input of rad2deg was two figures that represent a radian and a degree, respectively. The result of rad2deg were four figures, which were the integral and fractional parts of the degree converted from the radian, and the integral and the fractional parts of the radian converted from the degree. The weights of the integral parts were greater than those of the fractional parts.

(3) Evaluation Metrics.

Five metrics were evaluated, namely the SDC coverage, detection overhead, detection efficiency, benign detection ratio, and detection degree. The SDC coverage is the percentage of SDC that is detected by the hardened program, and also refers to the SDC detection ratio. The detection overhead is the cost of SDC detection, which is evaluated by taking the total execution times of the instructions in the original program as a baseline, and is represented by the difference between the total execution times of the instructions of the hardened program and that of the original program, divided by that of the original program. Assertion screening decreases both the SDC coverage and detection overhead. To ensure a fair comparison in terms of the SDC coverage and detection overhead, the detection efficiency was evaluated. It is the ratio between the SDC coverage and detection overhead, and has been used in previous research [14,25]. The benign detection ratio is the percentage of benign errors that are detected by the hardened program. The detection degree is the averaged damage of the SDC, which is detected by the hardened program, to the result of the program. The damage of an SDC is quantified by the total weights of the corrupted outputs of the program that are corrupted by the SDC.

5.2. Experimental Evaluation

(1) SDC coverage, detection overhead and detection efficiency

Figure 6a compares F_Radish with Radish in term of the SDC coverage. From Figure 6a, it can be seen that the average SDC coverage of Radish across the four programs was 76.9%. In contrast, the result of F_Radish was 57%. 19.9% SDCs detected by the program hardened by Radish were not detected by the program hardened by F_Radish. This is because the program hardened by F_Radish had fewer assertions than the program hardened by Radish after the two assertion-screening stages of F_Radish. Figure 6b presents the results of the detection overhead, from which it is evident that F_Radish reduced the detection overhead. The average detection overhead of Radish was 54%, while that of F_Radish was 21%, which is 33% less than that of Radish. The reason for this is that, in comparison with the program hardened by Radish, the program hardened by F_Radish executed fewer instructions, as it had fewer assertions. Figure 6a,b reveal that although F_Radish impaired the SDC coverage, it reduced the detection overhead.

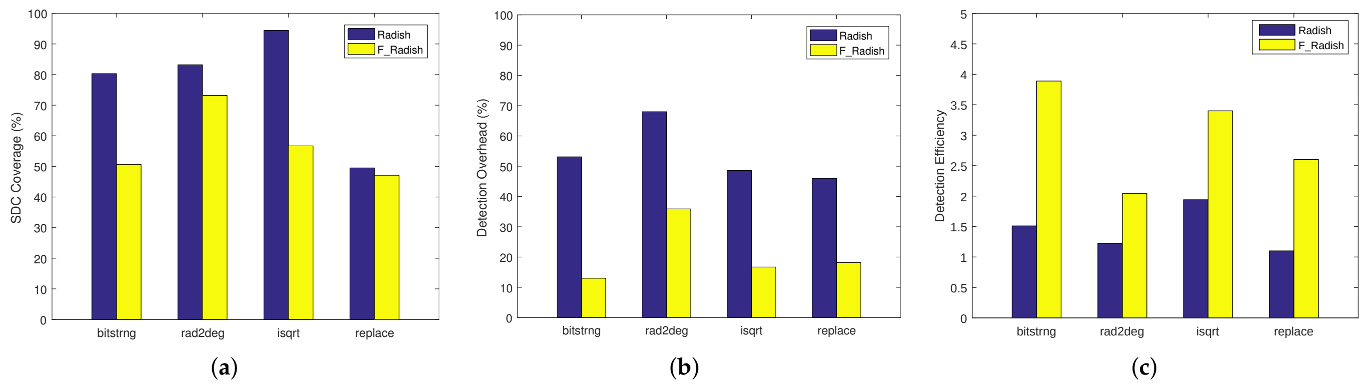


Figure 6. The performance of F_Radish and Radish on (**a**) silent data corruption (SDC) coverage, (**b**) detection overhead, (**c**) detection efficiency.

To ensure a fair comparison between Radish and F_Radish in terms of the SDC coverage and detection overhead, the metric of the SDC detection efficiency was evaluated, as shown in Figure 6c. The average detection efficiency of Radish was 1.4, while that of F_Radish result was 2.7, which is about two times greater than that of Radish. This is because, although Radish had a high SDC coverage because it had more assertions, its detection overhead was also very high. In contrast, F_Radish obtained a proper SDC coverage with low overhead by screening redundant assertions, thereby increasing its detection efficiency. Consider rad2deg as an example, the SDC coverage and detection overhead of Radish were 83.2% and 68%, respectively, while those of F_Radish were, respectively, 73.2% and 35.9%. The detection efficiencies of Radish and F_Radish on rad2deg were 1.2 and 2, respectively, meaning that F_Radish made a better trade-off between the SDC coverage and detection overhead.

There may exist some programs that have special requirements for the SDC coverage and detection overhead. For example, a program may pay more attention to the SDC coverage than to the detection overhead, or vice versa. To meet different requirements, the creation of an adjustable F_Radish is important, and this will be addressed in future work. With regard to this future work, to satisfy the requirement for a higher SDC coverage, more assertions remain. For example, during the first stage of F_Radish, two or more of the most important assertions remain at a program point rather than one. To satisfy the requirement for a lower detection overhead, more redundant assertions can be deleted. For example, during the second stage of F_Radish, n $(n > 2)$ assertions are considered as a group, instead of considering two assertions as an assertion-pair. In this case, the detection overhead is reduced by determining whether the previous $n-1$ assertions can be deleted.

(2) Benign detection ratio

Figure 7 compares the benign detection ratio of F_Radish with that of Radish. As can be seen, the benign detection ratios of F_Radish for the four programs were all lower than those of Radish. On average, the benign detection ratio of Radish was 27.8%, while that of F_Radish was 19.2%, thereby exhibiting a decrease of 8.6%. This means that F_Radish detected fewer benign errors than Radish, and could avoid more unnecessary recovery overhead.

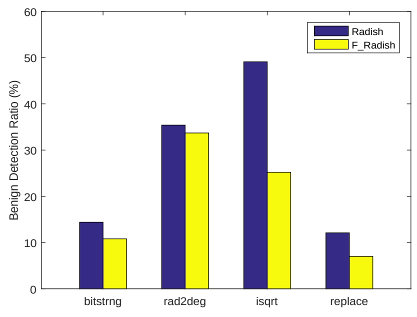


Figure 7. The benign detection ratios of F_Radish and Radish.

(3) Detection degree

The detection degrees of F_Radish and Radish were evaluated, and the results are presented in Figure 8. The average detection degree of Radish was 0.4, whereas that of F_Radish was 0.44. F_Radish achieved a higher detection degree by considering the detection degrees of assertions in the first assertion-screening stage. The percentage increase of the detection degree is the difference between the detection degrees of F_Radish and Radish divided by the detection degree of Radish, and was equal to 10%. These results demonstrate that F_Radish outperformed Radish in term of the detection degree, and it detected more serious SDC than did Radish.

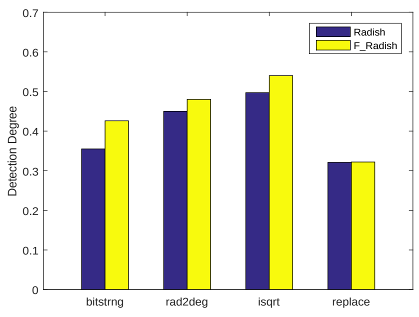


Figure 8. The detection degrees of F_Radish and Radish.

(4) Different contributions of the two stages of F_Radish

The influences of the two stages of F_Radish on the SDC coverage, detection overhead, detection efficiency, benign detection ratio, and detection degree were evaluated to reveal the different contributions of the two stages to the enhancement of SDC detection. The influence of each stage on every metric is represented by the decrement or increment of the metric produced by it. For example, the influence of the first stage on the benign detection ratio is the decrement of the benign detection ratio produced by the first stage, namely the difference between the benign detection ratios of P and FP. As another example, the influence of the second stage on the detection efficiency is the increment of the detection efficiency produced by the second stage, namely the difference between the detection efficiencies of SP and FP.

The influences of the two stages on the SDC coverage, detection overhead, and detection efficiency were first evaluated, as presented in Figure 9. As shown in Figure 9a,b, for stage one, the averaged decrements of the SDC coverage and detection overhead were 17.1% and 29%, respectively. For stage two, these values were respectively 2.8% and 4%. The two stages therefore had different influences on the SDC coverage and detection overhead. The increment of the detection efficiency is presented in Figure 9c, which reveals that the averaged increments of the detection efficiency for stage one and stage two were 1 and 0.3, respectively. This means that stage one made greater contributions to the improvement of the detection efficiency than did stage two. Although stage two made fewer contributions, it increased the detection efficiency by 0.3. Via the two stages, F_Radish enhanced SDC detection in terms of the detection efficiency. It is worth noting that the reason why the contribution of stage two was less than that of stage one is that there were fewer redundant assertions in neighbouring program points than in program points.

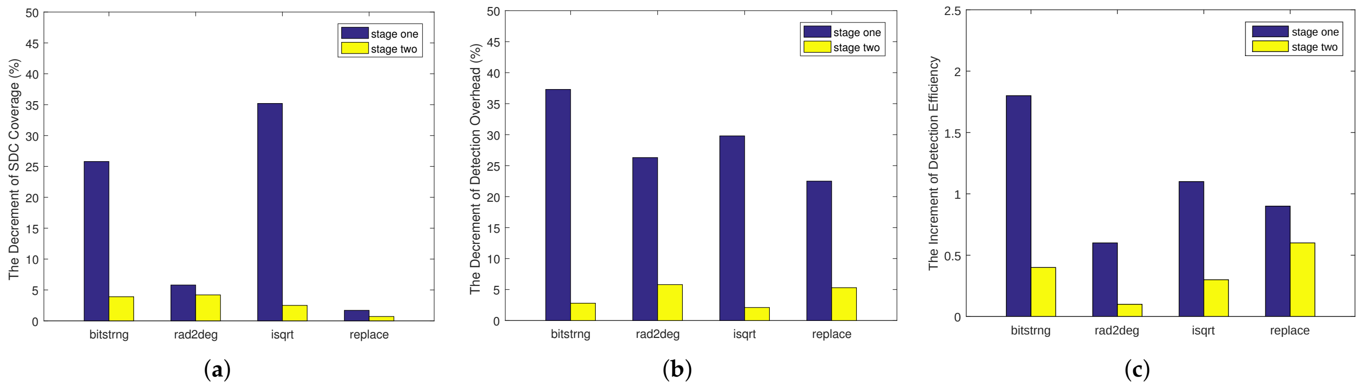


Figure 9. The decrement or increment of (**a**) SDC coverage, (**b**) detection overhead, (**c**) detection efficiency.

The influences of the two stages on the benign detection ratio and detection degree were then evaluated, as shown in Figures 10 and 11. As exhibited in Figure 10, the averaged decrements of the benign detection ratio for stage one and stage two were 8.1% and 0.5%, respectively. As presented in Figure 11, the averaged increments of the detection degree for stage one and stage two were 0.04 and −0.0008, respectively. In summary, the first stage made contributions to decreasing the benign detection ratio and improving the detection degree, while the second stage made little contributions. The reason for this is that, in comparison with stage one, the second stage did not consider the benign detection ratio or detection degree during assertion screening.

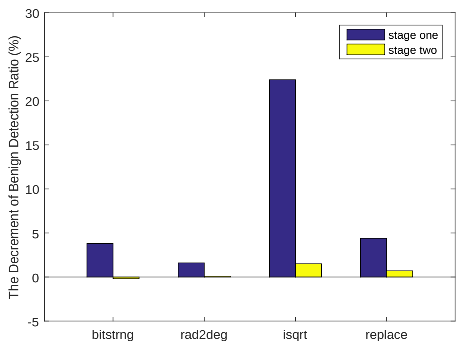


Figure 10. The decrement of benign detection ratio.

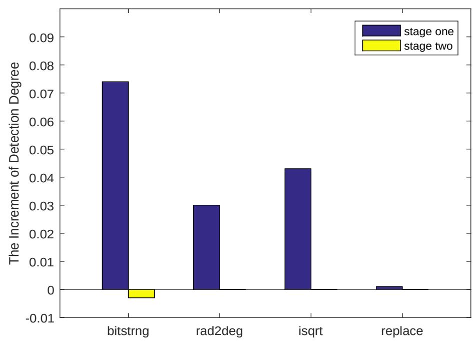


Figure 11. The increment of detection degree.

6. Conclusions and Future Work

This paper proposed the F_Radish approach, which is an enhancement of SDC detection that screens redundancy assertions to improve the detection efficiency and detection degree while reducing the benign detection ratio.

There are two stages of F_Radish, namely, the screening of assertions for each program point and the screening of assertions for neighbouring program points. In the first stage, if a program point has only one assertion, it is skipped. Otherwise, the benign detection ratio and detection degree are applied to evaluate the importance of each assertion in the program point. As a result, only the most important assertion remains in the program point. In the second stage, assertion-pairs are first generated. Then, for each assertion-pair, the redundancy degree of the former assertion with respect to the latter assertion and the profit of deleting the former assertion are calculated. If the redundancy degree exceeds a specified threshold and there is a profit of deleting the former assertion, the former assertion is deleted. Otherwise, it is not deleted.

Experiments were conducted to validate the effectiveness of F_Radish, and the results demonstrated that F_Radish improved the detection efficiency and detection degree and reduced the benign detection ratio. The influences of the two stages of F_Radish were also analysed to demonstrate the different contributions of the two stages to the enhancement of SDC detection. The experimental results indicated that both stages made contributions to the improvement of the detection efficiency, and the first stage made greater contributions to the reduction of the benign detection ratio and the improvement of the detection degree.

In future work, a deeper exploration will be conducted to further improve the detection degree, and a checkpoint recovery mechanism will be applied to quantify the unnecessary recovery overhead. The second stage of F_Radish will also be refined to improve its contributions, and multiple bit flips will be considered. Another facet of future research will be to make F_Radish adjustable.

Author Contributions: Conceptualization, N.Y. and Y.W.; investigation, N.Y. and Y.W.; methodology, N.Y. and Y.W.; software, N.Y.; supervision, Y.W.; validation, N.Y.; writing—original draft preparation, N.Y.; writing—review and editing, Y.W.; funding acquisition, Y.W. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Hi-Tech Project, China with grant No. 315055101, and Collaborative Innovation Center of Novel Software Technology and Industrialization.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Restrictions apply to the availability of these data. Data sharing is not applicable to this article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. James, B.; Tran, L.T.; Bolst, D.; Peracchi, S.; Davis, J.A.; Prokopovich, D.A.; Guatelli, S.; Petasecca, M.; Lerch, M.; Povoli, M.; et al. SOI Thin Microdosimeters for High LET Single-Event Upset Studies in Fe, O, Xe, and Cocktail Ion Beam Fields. *IEEE Trans. Nucl. Sci.* **2020**, *67*, 146–153. [CrossRef]
2. Olsen, J.; Becher, P.E.; Fynbo, P.B.; Raaby, P.; Schultz, J. Neutron-induced Single Event Upsets in Static RAMS Observed at 10 km Flight Altitude. *IEEE Trans. Nucl. Sci.* **1993**, *40*, 74–77. [CrossRef]
3. Martínez, J.A.; Maestro, J.A.; Reviriego, P. Evaluating the Impact of the Instruction Set on Microprocessor Reliability to Soft Errors. *IEEE Trans. Device Mater. Reliab.* **2018**, *18*, 70–79. [CrossRef]
4. Yang, N.; Wang, Y. Identify Silent Data Corruption Vulnerable Instructions using SVM. *IEEE Access* **2019**, *2019*, 40210–40219. [CrossRef]
5. Yang, N.; Wang, Y. Predicting the Silent Data Corruption Vulnerability of Instructions in Programs. In Proceedings of the IEEE International Conference on Parallel and Distributed Systems, Tianjin, China, 4–6 December 2019; pp. 862–869.
6. Li, G.P.; Pattabiraman, K.; Hari, S.K.S.; Sullivan, M.; Tsai, T. Modeling Soft-Error Propagation in Programs. In Proceedings of the IEEE International Conference Dependable Systems and Networks, Luxembourg, 25–28 June 2018; pp. 27–38.
7. Snir, M.; Wisniewski, R.W.; Abraham, J.A. Addressing Failures in Exascale Computing. *Int. J. High Perform. Comput. Appl.* **2014**, *28*, 129–173. [CrossRef]
8. Calhoun, J.; Snir, M.; Olson, L.N.; Gropp, W.D. Toward a More Complete Understanding of SDC Propagation. In Proceedings of the International Symposium High-Performance Parallel and Distributed Computing, Washington, DC, USA, 26–30 June 2017; pp. 131–142.
9. Reis, G.A.; Chang, J.; Vachharajani, N.; Rangan, R.; August, D.I. SWIFT: Software Implemented Fault Tolerance. In Proceedings of the International Symposium Code Generation and Optimization, New York, NY, USA, 20–23 March 2005; pp. 243–254.
10. Didehban, M.; Shrivastava, A.; Lokam, S.R.D. NEMESIS: A Software Approach for Computing in Presence of Soft Errors. In Proceedings of the IEEE/ACM Int. Conf. Computer-Aided Design, Irvine, CA, USA, 13–16 November 2017; pp. 297–304.
11. Thati, V.B.; Vankeirsbilck, J.; Boydens, J.; Pissort, D. Selective Duplication and Selective Comparison for Data flow Error Detection. In Proceedings of the International Conference System Reliability and Safety, Rome, Italy, 20–22 November 2019; pp. 10–15.
12. Berrocal, E.; Gomez, L.B.; Di, S.; Lan, Z.L.; Cappello, F. Lightweight Silent Data Corruption Detection Based on Runtime Data Analysis for HPC Applications. In Proceedings of the International Symposium High-performance Parallel & Distributed Computing, Portland, OR, USA, 15–19 June 2015; pp. 1–4.
13. Thomas, T.E.; Bhattad, A.J.; Mitra, S.; Bagchi, S. Sirius: Neural Network based Probabilistic Assertions for Detecting Silent Data Corruption in Parallel Programs. In Proceedings of the IEEE International Symposium Reliable Distributed Systems, Budapest, Hungary, 26–29 September 2016; pp. 41–50.
14. Ma, J.; Yu, D.Y.; Wang, Y.; Cai, Z.B.; Zhang, Q.X.; Hu,C. Detecting Silent Data Corruptions in Aerospace-based Computing using Program Invariants. *Int. J. Aerosp. Eng.* **2016**, *2016*, 1–10. [CrossRef]
15. Sahoo, S.K.; Li, M.L.; Ramachandran, P.; Adve, S.V.; Sdve, V.S.; Zhou, Y.Y. Using Likely Program Invariants to Detect Hardware Errors. In Proceedings of the IEEE International Conference Dependable Systems and Networks with FTCS and DCC, Anchorage, AK, USA, 24–27 June 2008; pp. 70–79.
16. Di, S.; Cappello, F. Adaptive Impact-Driven Detection of Silent Data Corruption for HPC Applications. *IEEE Trans. Parallel Distrib. Syst.* **2016**, 27, 2809–2823. [CrossRef]
17. Restrepo-Calle, F.; MartÍnez-Álvarez, A.; Asensi, S.C.; Morenilla, A.J. Selective SWIFT-R: A Flexible Software-Based Technique for Soft Error Mitigation in Low-Cost Embedded Systems. *J. Electron. Test.* **2013**, *29*, 825–838. [CrossRef]
18. Rehman, S.; Shafique, M.; Aceituno, P.V.; Kriebel, F.; Chen, J.J.; Henkel, J. Leveraging Variable Function Resilience for Selective Software Reliability on Unreliable Hardware. In Proceedings of the IEEE International Conference Design, Automation & Test in Europe Conference & Exhibition, Grenoble, France, 18–22 March 2013; pp. 1759–1764.
19. Mutlu, B.O; Kestor, G.; Cristal, A.; Unsal, O.; Krishnamoorthy, S. Ground-Truth Prediction to Accelerate Soft-Error Impact Analysis for Iterative Methods. In Proceedings of the IEEE International Conference High Performance Computing, Data, and Analytics, Hyderabad, India, 17–20 December 2019; pp. 333–344.
20. Chen, C.; Eisenhauer, G.; Wolf, M.; Pande, S. LADR: Low-cost Application-level Detector for Reducing Silent Output Corruptions. In Proceedings of the International Symposium High-Performance Parallel and Distributed Computing, Tempe, AZ, USA, 11–15 June 2018; pp. 156–167.
21. Racunas, P.; Constantinides, K.; Manne, S.; Mukherjee, S.S. Perturbation-based Fault Screening. In Proceedings of the IEEE International Symposium High Performance Computer Architectur, Scottsdale, AZ, USA, 10–14 February 2007; pp. 169–180.
22. Hari, S.K.S.; Adve, S.V.; Naeimi, H. Low-cost Program-level Detectors for Reducing Silent Data Corruptions. In Proceedings of the IEEE International Conference Dependable Systems and Networks, Boston, MA, USA, 25–28 June 2012; pp. 1–12.
23. Ernst, M.D.; Perkins, J.H.; Guo, P.J.; McCamant, S.; Pacheco, C.; Tschantz, M.S.; Xiao, C. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* **2007**, *69*, 35–45. [CrossRef]
24. Fang, B.; Lu, Q.; Pattabiraman, K.; Ripeanu, M.; Gurumurthi, S. ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-layer Resilience Analysis. In Proceedings of the IEEE International Conference Dependable Systems and Networks, Toulouse, France, 28 June–1 July 2016; pp. 168–179.

25. Lu, Q.; Pattabiraman, K.; Gupta, M.S.; Rivers, J.A. SDCTune: A Model for Predicting the SDC Proneness of an Application for Configurable Protection. In Proceedings of the International Conference Compilers, Architecture and Synthesis for Embedded Systems, Greens, India, 12–17 October 2014; pp. 1–10.
26. Ma, J.; Wang, Y.; Zhou, L.; Hu, C.; Wang, H. SDCInfer: Inference of Silent Data Corruption Causing Instructions. In Proceedings of the International Conference Software Engineering and Service Science, Beijing, China, 23–25 September 2015; pp. 228–232.
27. Ma, J.; Wang, Y. Identification of Critical Variables for Soft Error Detection. In Proceedings of the IEEE International Conference Human Centered Computing, Colombo, Sri Lanka, 7–9 January 2016; pp. 310–321.