elektron ámbar

Original Article

Revista elektron, Vol. 7, No. 1, pp. 28-33 (2023)
https://doi.org/10.37537/rev.elektron.7.1.178.2023

Mitigación de errores de software producidos por la radiación del ambiente espacial

Mitigation of software errors produced by radiation from the space environment

Germán Castro

Laboratorio de Sistemas Embebidos
Facultad de Ingeniería - UBA
Buenos Aires, Argentina
gcastro@fi.uba.ar

Recibido: 09/04/23; Aceptado: 01/05/23

***Resumen*—En este trabajo se presenta el desarrollo e implementación de una técnica de mitigación de errores de software con el objetivo de proteger funciones a ser utilizadas en misiones espaciales. Se analiza la técnica *Preemptive Control Signature* para mitigar los errores del tipo *Single Event Upset* sobre la arquitectura ARMv7E-M y poder así extender el tiempo de disponibilidad de los dispositivos espaciales aumentando la tolerancia a dichos errores. La técnica implementada logró detectar el 79.5 % de los errores.**

Palabras clave: CFI; CMOS; ISA; PECOS; SEU.

***Abstract*—This paper presents the development and implementation of a software error mitigation technique with the aim of protecting functions to be used in space missions. The *Preemptive Control Signature* technique is analyzed to mitigate *Single Event Upset* type errors on the ARMv7E-M architecture and thus be able to extend the availability time of spatial devices by increasing tolerance to such errors. Implemented technique detected 79.5 % of the errors.**

Keywords: CFI; CMOS; ISA; PECOS; SEU.

I. Introduction

Dado lo agresivo del ambiente espacial y el elevado costo de los lanzamientos y de las misiones espaciales, los componentes electrónicos utilizados son sometidos a un largo y costoso proceso de diseño y calificación. Como consecuencia de ello los dispositivos, tales como los CMOS, adolecen de un pronunciado retraso tecnológico.

Al diseñar aplicaciones en entornos de radiación hostiles, como el espacial, se deben considerar los efectos que puede producir el impacto de partículas cargadas, tales como los iones pesados o los protones, sobre los dispositivos electrónicos. Mientras estas partículas cargadas atraviesan un dispositivo CMOS (ver figura 1), pueden alterar los estados lógicos o cualquier elemento de memoria estática al depositar (perder) energía e inducir carga (pares electrón-laguna) a lo largo de su camino, generando perturbaciones. La mayoría de los pares inducidos se recombinan inmediatamente, mientras que una pequeña fracción puede ser recolectada por el campo eléctrico del dispositivo. Estas perturbaciones, al alterar celdas lógicas, pueden inducir errores en el software (SE) que, dependiendo la aplicación, pueden generar fallas críticas en el sistema.

Para aprovechar el uso de tecnologías modernas en la industria espacial, surgió una nueva iniciativa dentro del uso

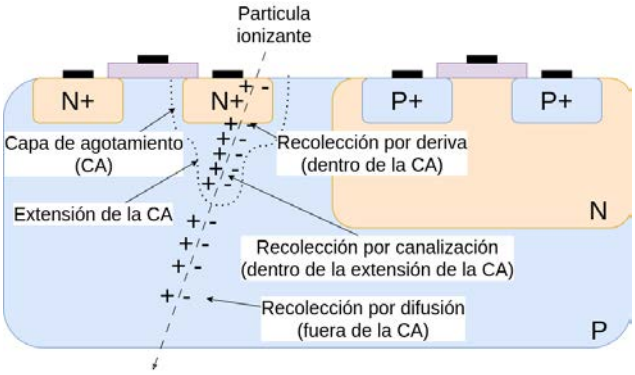

Figura 1: Partícula ionizante atravesando un dispositivo CMOS.

comercial del espacio denominada *New Space*, impulsada por emprendimientos privados [1]. La iniciativa *New Space* tiene como objetivo replantear radicalmente la metodología tradicional de desarrollo de proyectos espaciales. Dentro de las áreas de estudio a replantear se encuentran las técnicas alternativas de evaluación y mitigación de vulnerabilidades de software producidas por la radiación del ambiente espacial [2]. El presente trabajo intenta contribuir a dicha área de estudio al permitir medir y garantizar la capacidad de mitigación de SE en los sistemas de vuelo de los satélites, así como también evaluar el rendimiento de las herramientas de inyección de SE ya existentes.

I-A. Errores de flujo de control

Este documento se concentra en CFEs, que son errores que causan la divergencia de la secuencia de valores del PC respecto de su secuencia durante la ejecución libre de errores de la aplicación [3]. Los CFEs pueden generar la corrupción de datos, fallas de programa o propagación de errores [4]. Estudios indican que un tercio de los errores producidos en el código de una aplicación de uso de datos no intensivo conducen a CFEs [5]. Se propone una técnica de verificación preventiva y generación de firmas de flujo de control denominada PECOS. Que la verificación sea preventiva significa que la técnica de detección es activada antes que un error cause la ejecución de un camino de flujo de control incorrecto [6].

II. Diseño e implementación

PECOS monitorea el camino de control tomado en tiempo de ejecución por una aplicación y la compara con el conjunto de caminos de control esperados para validar el comportamiento de la aplicación. El esquema puede manejar situaciones en las que los caminos de control se determinan estática o dinámicamente (i.e., en tiempo de ejecución).

La aplicación se representa por su CFG que a su vez se compone de nodos (o bloques básicos en el sentido tradicional de un compilador). Cada nodo comienza con un BFI y termina con una CFI que es utilizada como un disparador para el PAB. Un PAB es insertado en cada nodo justo antes de su CFI. El PAB contiene:

1. Un conjunto de direcciones de destino válidas (i.e., firmas de referencia o "copias de oro") a las que la aplicación puede saltar, las cuales son determinadas en tiempo de compilación o en tiempo de ejecución.
2. Código para determinar la dirección de destino en tiempo de ejecución.

La determinación de la dirección de destino en tiempo de ejecución y su comparación contra las direcciones válidas se realiza antes de que el salto a la dirección destino se ejecute. En caso de error, el PAB lanza una excepción de división por cero, la cual es atendida por el PSH. El PSH verifica si el problema fue causado por un CFE y, de ser así, ejecuta una acción de recuperación, e.g., terminar la ejecución del hilo defectuoso.

En este punto, es importante destacar la diferencia entre una CFI incorrecta y una CFI ilegal. Se propone como ejemplo un escenario en el cual un salto condicional tiene dos posibles destinos o caminos de ejecución, P1 o P2, y que en tiempo de ejecución se decide que el camino correcto a tomar es P1. Si la ejecución toma el camino P2, entonces se ejecutó una CFI incorrecta pero legal. Si la ejecución toma un camino aleatorio P3, que no está dentro de los caminos posibles, entonces se ejecutó una CFI ilegal (y claramente incorrecta). PECOS puede detectar todas las CFIs ilegales y un subconjunto de CFIs incorrectas pero legales.

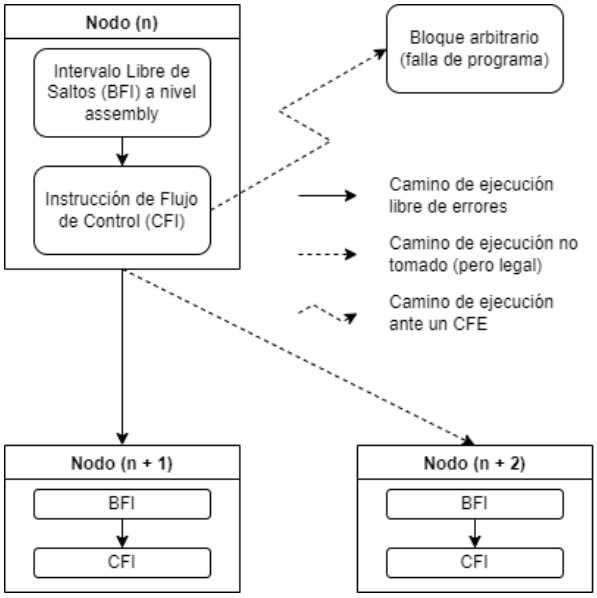

Figura 2: Caminos de ejecución sin PECOS.

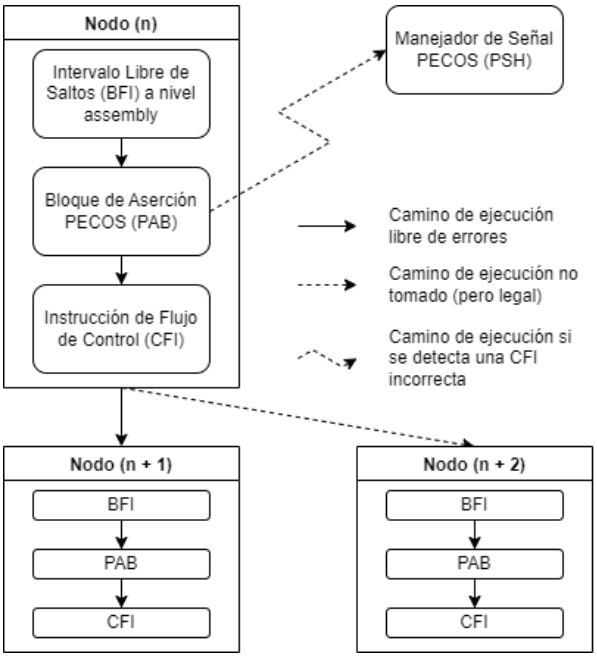

Figura 3: Caminos de ejecución con PECOS.

II-A. *Tipos de errores protegidos*

PECOS detecta CFEs, errores que pueden ocurrir por distintos tipos de fallas. A continuación se enumeran los tres modelos de errores que pueden ser manejados por PECOS con un ejemplo representativo para cada uno.

1. SEU (SBU y MBU). Estos son errores que producen un *bit flip* en la memoria. Puede ser tanto en la memoria principal como en la memoria cache. Un SEU puede ocasionar también un SEFI, generando un *latch-up* en el dispositivo CMOS. E.g.: una rafaga de errores que corrompa una instrucción *call* en la memoria cache L1.
2. Transmisión de errores durante la comunicación entre dos niveles de distinta jerarquía de memoria, e.g.: un error en la linea de direcciones cuando una CFI se está trayendo desde la memoria al procesador.
3. Errores en registros, e.g.: corrupción del valor de un registro que determina la dirección de destino de una instrucción de salto incondicional.

PECOS no puede capturar un CFE que resulte de la corrupción de una no-CFI en una CFI, ya que los PABs son insertados únicamente antes de las CFIs. De todas formas, se observa que las ISAs, tales como ARM Thumb 2, están construidas de tal manera que la distancia de Hamming [7] de los *opcodes* pertenecientes a la categoría no-CFI respecto a la categoría CFI es bastante larga. Es razonable entonces esperar que errores que causen la transformación de una no-CFI en una CFI sean muy raros en la práctica [8].

II-B. *Construcción de PAB*

Cada CFI es precedida por un PAB. El PAB no debe estar compuesto por ninguna CFI adicional ya que es la CFI la que se intenta proteger, de lo contrario se estaría en presencia de un bucle de protección a CFIs sin sentido.

El objetivo del PAB es verificar que la dirección de destino X_d de la CFI pertenece a un conjunto de direcciones

de destino validas X_{sn}, lo que es igual a decir que:

$$\exists n \in N : X_d = X_{sn} \tag{1}$$

Para lograrlo, el PAB calcula la dirección de destino verificada X como indica la ecuación 2 de alto nivel. Las tareas que debe realizar el PAB en tiempo de ejecución son:

1. Determinar la dirección de destino X_d de la CFI en tiempo de ejecución.
2. Determinar la dirección de destino valida X_{sn} de la CFI en tiempo de ejecución. El numero de direcciones de destino validas pueden ser uno (saltos incondicionales), dos (saltos condicionales), o muchas (llamadas a, y retornos de funciones).
3. Comparar las dos direcciones determinadas anteriormente. La comparación se implementa de tal forma que la detección de una CFI ilegal causará una excepción de división por cero del procesador al intentar operar con la expresión booleana $isValid$, indicando un error.

$$X = \frac{X_d}{isValid} : isValid = ![(X_d - X_{s1}) \dots (X_d - X_{sn})] \tag{2}$$

II-C. Determinación de dirección de destino

La dirección de destino de la CFI es aquella que se especifica como el operando de la instrucción. PECOS se instrumentó para operar con VMAs, lo que quiere decir que el operando de la CFI no es de carácter constante, sino simbólico. Esto significa que, desde el punto de vista del código ensamblador, la CFI utilizará como operando una etiqueta que haga referencia a otra sección de código. Esta etiqueta es interpretada por el compilador y codificada con el resto de la CFI según las reglas de la arquitectura embebida especificada. El bloque de código 1 muestra cómo se puede extraer dicha codificación directamente desde la memoria de programa, el único requisito es conocer donde se encuentra ubicada la CFI, en este caso mediante la etiqueta ".CFI1".

Listing 1: Extracción de CFI en la memoria del programa.

```
@Load runtime CFI's VMA.
ldr     r0, =.CFI1
ldr     r1, [r0]
rev16   r1, r1
rev     r1, r1
```

II-D. Determinación de dirección de destino válida

Es necesario que el PAB reconstruya de principio a fin la codificación válida de la CFI que intenta proteger. En esencia debe repetir parte del trabajo que el compilador hace por única vez sobre un programa. Los requisitos para lograrlo son conocer: el nemónico de la CFI, el símbolo de su operando y los algoritmos que realiza la arquitectura embebida para codificar la instrucción completa. Si bien el concepto y los requisitos son iguales para cada CFI, el procedimiento puede variar ligeramente para cada una dependiendo de si, por ejemplo, se encuentra bajo ejecución condicional o la arquitectura presenta varias codificaciones para la misma CFI. Es por esto que PECOS puede tener un trato especial para cada CFI.

El bloque de código 2 muestra cómo se puede reconstruir una codificación T4 válida de una CFI *Branch*.

Listing 2: Reconstrucción de CFI válida.

```
@Load CFI target address
ldr r2, =.L3
add     r0, r0, #4
@Build 32 bit immediate (imm32)
sub     r2, r2, r0
mov     r3, #0
@Load imm11 from imm32.
mov     r4, r2, lsr #1
mov     r0, #0x7FF
and r4, r4, r0
orr     r3, r3, r4
@Load imm10 from imm32.
mov     r4, r2, lsr #12
mov     r0, #0x3FF
and r4, r4, r0
mov     r4, r4, lsl #16
orr     r3, r3, r4
@Load S from imm32.
mov     r0, r2, lsr #24
and r0, r0, #0x1
mov     r4, r0, lsl #26
orr     r3, r3, r4
@Load I2 from imm32.
mov     r4, r2, lsr #22
and r4, r4, #0x1
@Load J2 = not (I2 xor S)
eor     r4, r4, r0
mvn     r4, r4
and r4, r4, #0x1
mov     r4, r4, lsl #11
orr     r3, r3, r4
 @Load I1 from imm32.
mov     r4, r2, lsr #23
and r4, r4, #0x1
@Load J1 = not (I1 xor S)
eor     r4, r4, r0
mvn     r4, r4
and r4, r4, #0x1
mov     r4, r4, lsl #13
orr     r3, r3, r4
@Load CFI's nemonic opcode.
ldr     r4, =#0xF0009000
orr     r3, r3, r4
```

II-E. Validación de salto de programa

Habiendo obtenido la CFI en memoria de programa y reconstruido la CFI válida en las secciones anteriores, solo resta verificar que coincidan y de no hacerlo enviar una señal de error al procesador para ser atendida.

A nivel de código ensamblador, la forma más elemental de verificación de igualdad entre dos valores es efectuar la diferencia entre ambos y comparar si el resultado de la diferencia es igual a 0. Se opera de esta forma para obtener la expresión *isValid* como indica la ecuación 2 y disparar una excepción de división por 0 en caso de detectar una

diferencia. El bloque de código 3 ejemplifica cómo realizar la validación.

Listing 3: Validación de CFI.

```
@Should store zero for correct execution.
sub     r3, r3, r1
@rx = !rx
cmp     r3, #0
ite     eq
moveq   r3, #1
movne   r3, #0
uxtb    r3, r3
@Raises exception if rx was not zero.
sdiv    r3, r1, r3
```

II-F. Overhead de código

Basados en la arquitectura del procesador objetivo a proteger, el *overhead* que producen las verificaciones de control de flujo debe compararse contra la probabilidad de ejecución de instrucciones ilegales. Si se inserta un PAB por cada CFI ha de esperarse que el overhead de PECOS sea elevado. El overhead de memoria se define como el aumento de tamaño del segmento de código de la aplicación debido a la inserción de PABs. Una aplicación orientada al manejo de datos intensivos tiene una menor cantidad de CFIs que una aplicación orientada al control intensivo y, por lo tanto, un menor overhead de memoria. Siendo n y a la cantidad de instrucciones *assembly* de largo que tiene un BFI y un PAB respectivamente, el overhead C de memoria se define como:

$$C = \frac{a}{n} \cdot 100\,\% \tag{3}$$

Estudios previos sobre técnicas de detección de CFE, tanto por hardware y software, no proveen un detalle explícito de overhead de memoria y solo algunos proveen una estimación. El valor de n es dependiente del tipo de aplicación, siendo menor cuanto más orientada al control intensivo sea la aplicación. En general, para todas las técnicas de detección de CFE, se estima un overhead en el rango de 50-150 %. PECOS puede ser aplicado únicamente a secciones de código criticas para reducir el overhead y es responsabilidad del diseñador evaluar el criterio de implementación.

III. Ensayos y resultados

El ambiente de ensayo utilizado consiste en un esquema de conexión trivial al trabajar con placas de desarrollo. La figura 4 ilustra la conexión que permite comunicar al ordenador con el microcontrolador a través de una sonda de depuración.

Se abordó un esquema de ensayos donde, sobre la implementación real del hardware, se simulen los efectos de la radiación mediante una herramienta privativa de inyección de errores por software denominada SISE.

El objetivo del ensayo fue evaluar la mejora relativa del sistema al agregar la técnica de mitigación de SEs PECOS. SISE fue utilizada como herramienta modelo de inyección de SEs para realizar la totalidad de los ensayos [9]. SISE es instrumentada sobre una consola de texto plano que

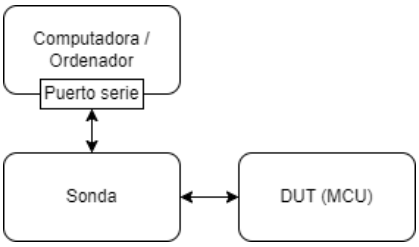

Figura 4: Diagrama en bloques de conexiones del ambiente de ensayo.

permite, mediante una conexión por puerto serie, inyectar SEs al DUT y recibir los reportes del mismo. Los errores que inyecta SISE están basados en técnicas de *bit flip* de memoria.

III-A. Rutina del DUT

La rutina del DUT consiste en ejecutar una función de prueba dentro de un bucle infinito y verificar que el retorno de los datos sea el esperado. De esta manera se pretende que el procesador destine la mayor cantidad de ciclos de reloj a la ejecución de la función bajo análisis. La ejecución de la rutina del DUT es la parte más importante de todas actividades que debe realizar el mismo, las cuales se ilustran en la figura 5.

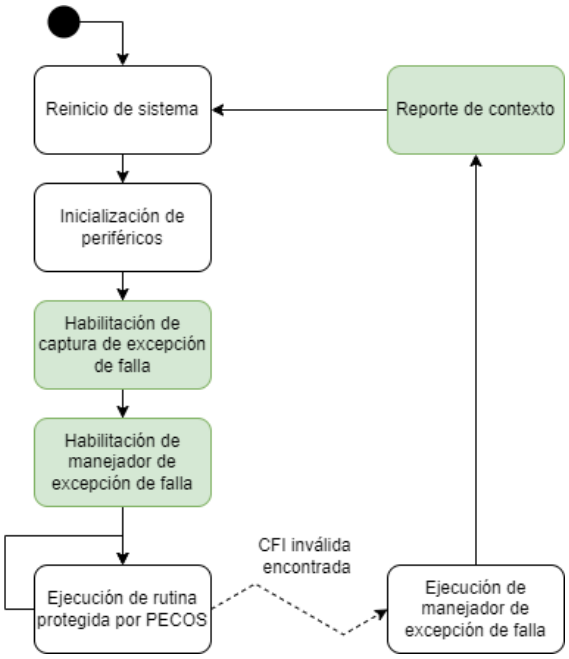

Figura 5: Diagrama de flujo de actividades del DUT, señalizando en verde el software generador de ambiente de ensayo.

La adición del software para generar un ambiente de ensayo tiene como fin poder configurar las excepciones de fallas a disparar, habilitar sus respectivos manejadores y obtener la información asociada a la falla para realizar un análisis posterior.

III-B. Técnica de diseño de prueba

La técnica utilizada fue *Control Flow Test (CFT)*. Su objetivo es ensayar la estructura del programa. Cada ensayo

o prueba consiste en un grupo de acciones que cubren un camino determinado a lo largo de un algoritmo o del programa. Esta es una técnica de diseño de prueba formal mayormente utilizada en pruebas unitarias y de integración.

La figura 6 ilustra un diagrama de flujo que representa el diseño técnico de un PAB insertado en un nodo del CFG. Cada punto de decisión y acción en el diagrama de flujo es identificado con una etiqueta única del tipo *DPx* y *ACx* respectivamente. Las acciones que se encuentren entre dos puntos de decisión sin bifurcaciones son consideradas como una acción conjunta y no son etiquetadas aunque se ilustren separadas para comprender el detalle individual de cada una de ellas.

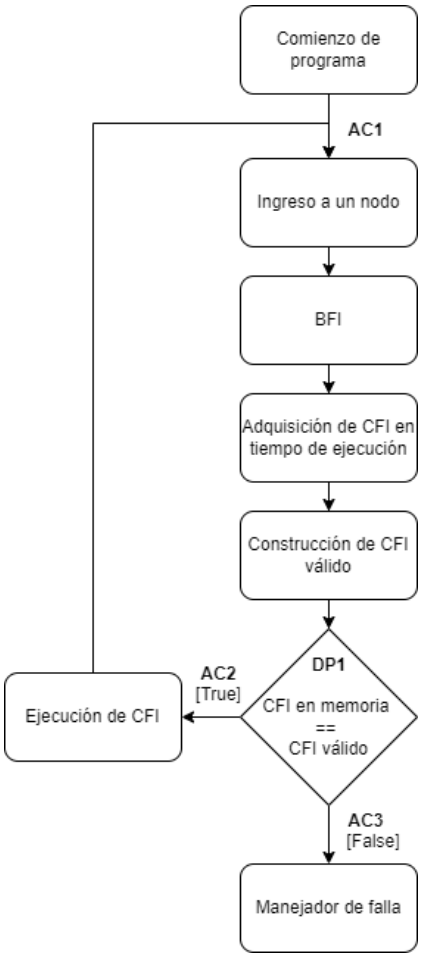

Figura 6: Diagrama de flujo de la estructura de un nodo del CFG con un PAB.

El nivel de profundidad de prueba utilizado para el PAB fue de 1 debido a su representación trivial. La combinación de acciones para los puntos de decisión son:

- - : (AC1)
- DP1 : (AC2); (AC3)

Esto lleva a la siguiente combinación de caminos:

- Camino 1: (AC1, AC3)
- Camino 2: (AC1, AC2, AC3)

III-C. Resultados

La tabla I presenta los resultados de las inyecciones realizadas sobre el DUT sin y con la técnica de mitigación de SEs PECOS aplicada junto con su factor de mejora. Cada fila muestra la sumatoria numérica de todos los ensayos realizados.

Los resultados de la inyección de errores se dividieron en las siguientes categorías:

- Error no manifestado: la instrucción errónea no es ejecutada por la aplicación, ya sea porque fue ejecutada previa a corromperse o porque el dato corrupto fue sobre-escrito con un valor válido.
- Detección de PECOS: el PAB detecta el error de forma preventiva antes que el sistema.
- Detección del sistema: el procesador detecta el error levantando una señal hacia un manejador de fallas (e.g., HardFault).
- Falla silenciosa: la aplicación retorna un valor erróneo luego de ejecutar su algoritmo. Se considera esta categoría como la más perjudicial para la disponibilidad del sistema.
- Inyecciones totales: cantidad total de SEs inyectados por la herramienta SISE.

Tabla I: Resultados de las inyección de errores de software en la aplicación.

Categoría	**Sin PECOS**	**Con PECOS**	**Factor de mejora**
Inyecciones totales	2520	2520	n/a
No manifestado	2290	2315	n/a
Detección: PECOS	n/a	163 (79.5 %)	n/a
Detección: sistema	186 (80.9 %)	39 (19.0 %)	4.8
Falla silenciosa	44 (19.1 %)	3 (1.5 %)	14.7

Los resultados en la tabla I caracterizan la eficiencia de PECOS en detectar errores cuando estos afectan directamente una CFI. Las mejoras sobresalientes que se observaron fueron:

- PECOS detectó aproximadamente el 80 % de todos los errores manifestados.
- Las fallas silenciosas se redujeron en un factor de 14.7 veces.
- Las detecciones del sistema se redujeron en un factor de 4.8 veces.

IV. Conclusión

Este trabajo presenta PECOS, una técnica de detección preventiva de errores de flujo de control. Esta técnica basada en software embebe bloques de aserción en el código assembly de la aplicación para detectar preventivamente cualquier camino de flujo de control ilegal o incorrecto y ejecutar una terminación elegante del proceso ofensivo.

Se desarrolló una aplicación que cubre todas las posibles codificaciones de instrucciones de flujo de control por parte de la arquitectura bajo análisis. Se implementó la aplicación sobre un hardware dedicado, mientras que la simulación de los efectos de la radiación espacial se realizó mediante una herramienta de inyección automática de errores por software.

A lo largo de este documento se hizo énfasis en la importancia de la detección preventiva y las mejoras que presenta el sistema. Se observaron también los beneficios de la detección preventiva tales como: la reducción de la propagación de errores, el aumento de la disponibilidad del sistema y la reducción de las detecciones del sistema, entre otros. Los resultados indican que PECOS es muy eficiente a

la hora de reducir las vulnerabilidades del sistema, logrando capturar aproximadamente un 80 % de los errores manifestados.

REFERENCIAS

[1] S. Hillmann and M. Wachter, "New space is becoming increasingly important for german industry." *The Federation of German Industries (BDI).*, Jun. 2022.

[2] D. Asciolla, L. Dilillo, D. Santos, D. Melo, A. Menicucci, and M. Ottavi, "Characterization of a risc-v microcontroller through fault injection." *Applications in Electronics Pervading Industry, Environment and Society.*, 2020. DOI: 10.1007/978-3-030-37277-4_11.

[3] S. S. Yau and F.-C. Chen, "An approach to concurrent control flow checking." *IEEE Transactions on Software Engineering.*, Mar. 1980. DOI: 10.1109/TSE.1980.234478.

[4] S. Chandra and P. M. Chen, "How fail-stop are faulty programs?." *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224).*, 1998. DOI: 10.1109/FTCS.1998.689475.

[5] S. Bagchi, Z. Kalbarczyk, R. Iyer, and Y. Levendel, "Design and evaluation of preemptive control signature checking." *IEEE Transactions on Computers.*, 2003.

[6] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection." *IEEE Transactions on Parallel and Distributed Systems.*, Jun. 1999. DOI: 10.1109/71.774911.

[7] R. W. Hamming, "Error detecting and error correcting codes." *The Bell System Technical Journal.*, Apr. 1950. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[8] N. J. Wang and S. J. Patel, "Restore: Symptom-based soft error detection in microprocessors." *IEEE Transactions on Parallel and Distributed Systems.*, Jul. 2006. DOI: 10.1109/TDSC.2006.40.

[9] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection." *Proceeding International Conference on Dependable Systems and Networks.*, 2000. DOI: 10.1109/ICDSN.2000.857571.