CARRERA DE ESPECIALIZACIÓN EN SISTEMAS EMBEBIDOS

MEMORIA DEL TRABAJO FINAL

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

Autor:
Ing. Germán Castro

Director:
Ing. Roberto M. Cibils (INVAP)

Codirector:
Ing. Damian F. Rosetani (INVAP)

Jurados:
Dr. Ing. Edgardo Comas (CITEDEF)
Esp. Ing. Esteban Volentini (FIUBA)
Esp. Ing. Luis Mariano Campos (CONICET)

Este trabajo fue realizado en la Ciudad Autónoma de Buenos Aires, entre marzo de 2022 y diciembre de 2022.

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 para la empresa INVAP S.E. El trabajo constituye la tesis de la Carrera de Especialización en Sistemas Embebidos dictada por la Facultad de Ingeniería de la Universidad de Buenos Aires.

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.

Conocimientos tales como arquitectura de procesadores, programación a nivel ensamblador, herramientas de depuración, ingeniería de software y técnicas de gestión de proyecto fueron necesarios para desarrollar el presente trabajo.

Índice general

Índice de figuras

Índice de tablas

Dedicado a mi familia.

Glosario

- BFI: branch-free interval.
- CFE: control flow error.
- CFG: control flow graph.
- CFI: control flow instruction.
- DUT: device under test.
- EA: executable assertion.
- IDE: integrated development environment.
- ISA: instruction set architecture.
- LR: link register.
- MBU: multiple bit upset.
- PAB: PECOS assertion block.
- PC: program counter.
- PECOS: preemptive control signature.
- PSH: PECOS signal handler.
- SBU: single bit upset.
- SE: software error (a.k.a. soft-error).
- SEE: single event effect.
- SEFI: single event functional interrupt.
- SEU: single event upset.
- SP: stack pointer.
- VMA: virtual memory address.

Capítulo 1

Introducción general

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 1.1), 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.

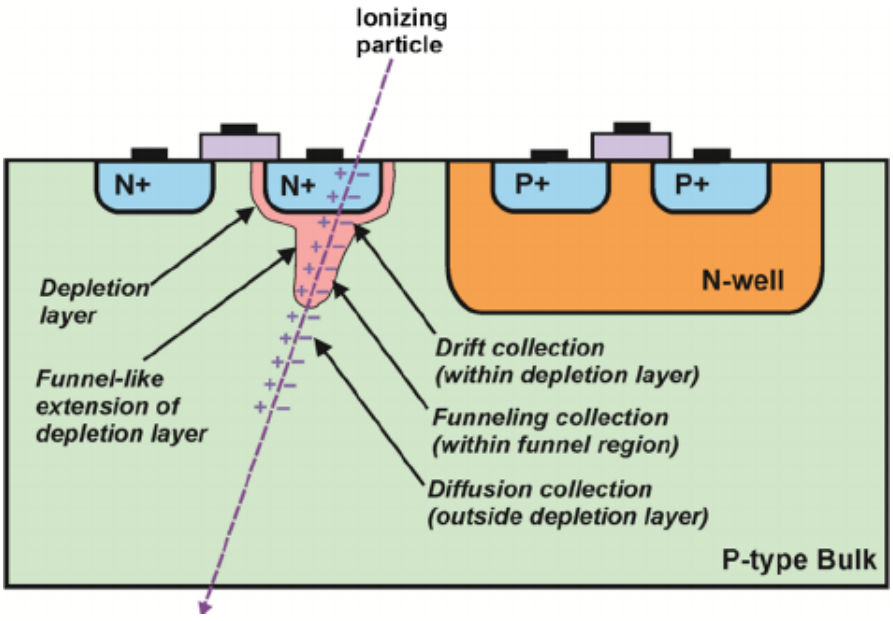


FIGURA 1.1. Partícula ionizante atravesando un dispositivo CMOS [1].

[1] Imagen tomada de [8].

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

1.1. Requerimientos

Se enumeran a continuación los requerimientos del presente trabajo:

1. Software de mitigación:
   - *a*) La comunicación con el debugger conformará con la configuración recomendada por el fabricante.
   - *b*) La comunicación con el debugger estará disponible durante todo el flujo de la secuencia.
   - *c*) Deberá incorporarse a una función a proteger desarrollada en lenguaje C.
   - *d*) No deberá modificar la funcionalidad de la función a proteger.
   - *e*) Deberá ensayar solo una estrategia de mitigación de soft-errors.
   - *f*) Deberá estar orientado a mitigar los errores del tipo SEU y SEFI específicamente.
   - *g*) Deberá implementar PECOS [3] como técnica de mitigación de soft-errors.
   - *h*) Deberá incorporarse a la función a proteger mediante herramientas de compilación cruzada sobre un microcontrolador SAMV71 [11].
2. Informe de investigación:
   - *a*) Deberá representar todos los caracteres de ISO Std. 10646 [12].
   - *b*) Cumplirá con las secuencias de escape especificadas en ISO Std. 6429 [13].
   - *c*) Usará el castellano como idioma conforme a la Real Academia Española.
   - *d*) Se aceptarán barbarismos que conformen la interfaz con los sistemas UNIX.
   - *e*) Deberá contener secciones que representen: una introducción teórica, un desarrollo del ensayo, resultados y conclusiones.
   - *f*) Títulos:
     1) Los títulos deberán tener una longitud máxima de 30 caracteres.

2) Los títulos deberán comenzar con mayúscula.

3) Los títulos deberán ser únicos.

g) Deberá incluir tablas de resultados con estadísticas de:

1) Errores inyectados en total.

2) Errores no detectados.

3) Errores detectados por el software de mitigación empleado.

4) Fallas silenciosas.

5) Programas abortados.

3. Documentación adicional:

a) Deberá realizarse un informe de avance a presentar el cuarto bimestre de la CESE.

b) Deberá realizarse la memoria técnica del proyecto a presentar en el quinto bimestre de la CESE.

1.2. Objetivos

Se enumeran a continuación los objetivos del presente trabajo:

1. Analizar diferentes estrategias de mitigación de SE, producidos por la radiación del ambiente espacial, a ser utilizadas en misiones espaciales.
2. Implementar una de dichas estrategias sobre un programa del microcontrolador SAM V71.
3. Definir una métrica de tolerancia a los efectos producidos por SEUs.
4. Utilizar una técnica de inyección de SE para determinar la efectividad de la estrategia de mitigación.

Capítulo 2

Introducción específica

Esta sección presenta los aspectos teóricos sobre errores de flujo de control, la técnica de mitigación PECOS, su instrumentación y los recursos utilizados para realizar el trabajo.

2.1. 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. Los CFEs pueden generar la corrupción de datos, fallas de programa o propagación de errores. Estudios indican que un tercio de los errores producidos en el código de una aplicación de uso de datos no intensivo conducen a CFEs [3]. 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.

En general las técnicas de detección funcionan bajo la premisa de que las detecciones propias del sistema (i.e.: falla de programa) son una manera aceptable de detectar un error y que solo los errores que escapan a las detecciones, tanto del sistema como de la técnica de detección propuesta, constituyen un problema. Desde el punto de vista de una recuperación de programa, es cuestionable que una falla de programa sea una manera aceptable de detección. Datos de sistemas reales [5] han mostrado que, mientras muchas fallas son benignas, bastantes fallas de sistema a menudo resultan de errores latentes causando propagación de errores. Estos errores terminan manifestándose como fallas, detenciones y problemas de recuperación severos. Además, el tiempo de recuperación de la aplicación es más alto cuando involucra la generación de un nuevo proceso (luego de una falla), comparado con la creación de un nuevo hilo (en el caso de procesos multi-hilos). En contraste con otras técnicas de mitigación, PECOS es una técnica preventiva por naturaleza y se activa antes que el error cause una falla del programa. Este enfoque presenta ventajas importantes tales como:

1. Reduce significativamente los incidentes de las fallas de programa. Esto permite la terminación elegante de un proceso ofensivo.
2. Minimiza la latencia de detección de error ya que es activado antes que un error se manifieste como una falla. Esto es especialmente importante a la hora de reducir los problemas provocados por la propagación de errores.

3. A diferencia de otras técnicas, PECOS también puede proteger CFIs cuyos destinos sean determinados en tiempo de ejecución, tales como saltos basados en valores de registros determinados, justamente, en tiempo de ejecución.
4. Puede usarse tanto si se dispone del código fuente como del ejecutable de la aplicación.

2.2. Técnica de mitigación PECOS

2.2.1. Principio de verificación preventiva

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.

2.2.2. Importancia de la verificación preventiva

La figura 2.1 resume el problema de los esquemas no preventivos e indica la solución propuesta por el enfoque PECOS. Las dos razones fundamentales por las que un enfoque preventivo es preferible se examinan a continuación, ambas razones están relacionadas a la facilidad de recuperación luego de una falla:

1. Prevención de falla de proceso: una falla total de la aplicación presenta un gran trabajo de recuperación debido al esfuerzo adicional de crear nuevamente sus procesos (el *kernel* asignando una nueva entrada en la tabla de procesos y actualizando el contador de *inode* de la estructura, contador de archivo, etc) y el esfuerzo adicional de recargar todo el estado del proceso desde un punto guardado. El enfoque PECOS verifica la dirección destino antes de que la CFI sea ejecutada (figura 2.1b). Además, el error es diagnosticado antes que la falla pueda ocurrir. La aplicación puede entonces terminar elegantemente, liberando los recursos (e.g., para un proceso multi-hilo, puede alcanzar tan solo con detener el hilo ofensivo). Se puede argumentar que una falla de proceso también se puede prevenir vía un manejador de falla, el cual intercepta las señales levantadas por el sistema y elegantemente termina el proceso de la aplicación o el hilo. En este escenario, sin embargo, el manejador entra en juego luego de ejecutada una instrucción corrupta, y consecuentemente, pudo haber ocurrido un daño indefinido al sistema.

2. Prevención de error de propagación: el segundo problema con los esquemas no preventivos es la posibilidad de propagación de error. Datos de [6] muestran que la latencia de detección es de aproximadamente 500 ciclos de instrucciones para esquemas basados en software y que los errores que permanecen sin detectarse por un largo período de tiempo tienen el potencial de causar problemas graves. Estos problemas pueden ir desde un punto de guardado corrupto que invalida los reintentos de restauración hasta el envío de mensajes incorrectos a los demás procesos en una aplicación distribuida. Afectar la recuperación del sistema en definitiva es atentar contra la disponibilidad general del sistema.

2.2.3. Tipos de errores contra los que protege PECOS

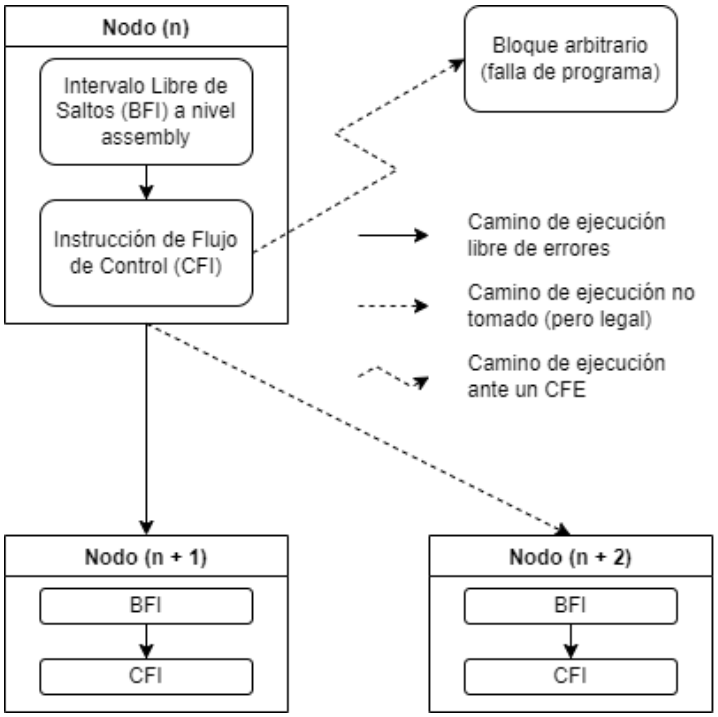
Pecos detecta CFEs, errores que pueden ocurrir por distintos tipos de fallas. A continuación se enumeran los tres modelos de errores que pueden ser manejados por PECOS con un ejemplo representativo para cada uno de ellos.

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.

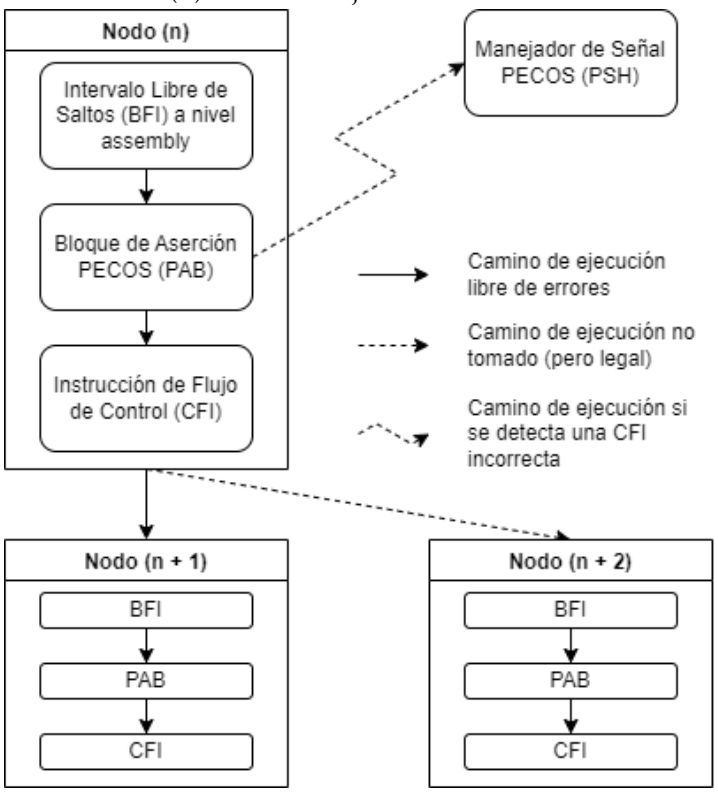
2. Transmisión de errores durante la comunicación entre dos niveles de distinta jerarquía de memoria, e.g.: un error en la linea de direcciones cuando una CFI se está trayendo desde la memoria al procesador.

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 [14], están construidas de tal manera que la distancia de Hamming [9] 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.



(A) Caminos de ejecución sin PECOS.



(B) Caminos de ejecución con PECOS.

FIGURA 2.1. Importancia de la verificación de flujo de control preventiva.

Las herramientas de detección de CFE por software generalmente no pueden detectar errores en la estructura interna del procesador. Para esto es necesario dar soporte desde el hardware mismo del procesador. Se ha estudiando una extensión para incorporar la técnica PECOS como un monitor de confiabilidad que se encuentre dentro del procesador y revise cada instrucción luego de ser extraída [7].

Con esta extensión, errores en la estructura interna del procesador tales como el PC pueden ser detectados. Con la implementación en software de PECOS detallada en este trabajo, un pequeño grupo de errores de PC pueden ser detectados, esto es, todos aquellos que eviten la ejecución de alguna instrucción clave dentro del PAB, lo que causaría que un registro no se configure correctamente y entonces, cuando se ejecute la instrucción de división, se emita la señal indicando un error.

2.3. Instrumentación de PECOS

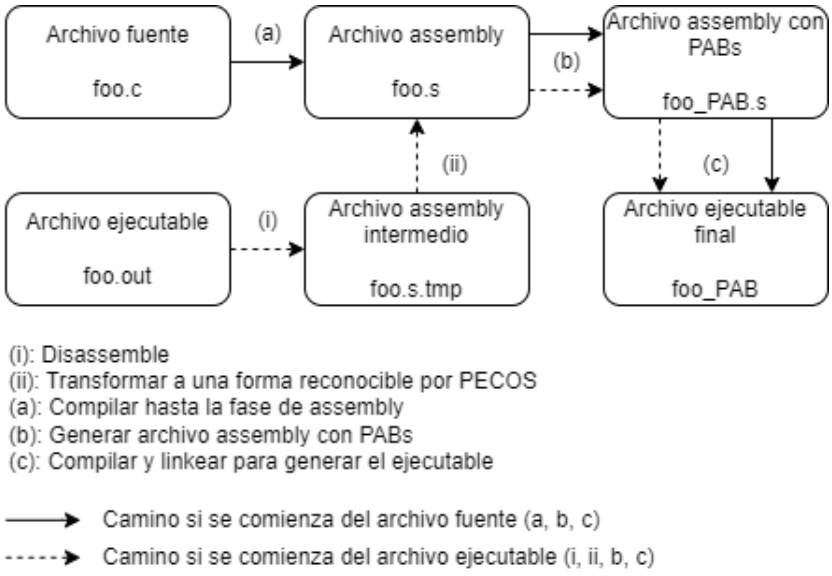
Un CFG es una representación gráfica de todos los caminos posibles que puede atravesar un programa durante su ejecución. Generar un CFG es, en esencia, lo que hacen los compiladores para representar un programa. Los nodos que componen al CFG representan los bloques básicos construidos por el compilador. Esta representación es dependiente del tipo de optimización utilizada (i.e., O0, O1, etc) ya que los bloques básicos estarán compuestos por diferentes instrucciones *assembly*. Todas las instrucciones *assembly* que contiene un bloque básico pueden agruparse en dos bloques internos: el bloque interno BFI ubicado al comienzo del bloque básico y el bloque interno CFI ubicado al final del bloque básico. El bloque interno BFI, como su nombre lo indica, está compuesto por todas aquellas instrucciones que no están destinadas a generar un salto de programa. Por el contrario, aquellas instrucciones que sí están destinadas a generar un salto de programa se denominan CFI y conforman el bloque interno CFI.

Obtener una representación que permita identificar las CFIs es clave para reconocer donde ubicar los PABs. Una vez ubicados los PABs, la compilación de la aplicación puede continuar de forma tradicional. La figura 2.2 muestra los caminos posibles para incorporar los PABs a la aplicación.

El DUT de este trabajo es un microcontrolador SAMv71 que está basado en el procesador ARM Cortex-M7 RISC [15] cuya arquitectura es ARMv7E-M [16]. La compilación para la arquitectura embebida ARMv7E-M se realizó de manera cruzada con las herramientas GNU recomendadas por ARM. Esta *toolchain* es un conjunto de compilador, linker, librerías y cualquier herramienta adicional necesaria para producir el ejecutable, a su vez el debugger o IDE pueden ser parte de la toolchain.

Es importante enfatizar que PECOS utiliza VMAs en sus PABs, logrando que la reubicación de código en memoria sea posible. Ya que PECOS se maneja a nivel de lenguaje ensamblador en lugar de un lenguaje de alto nivel (e.g.: C++, Python), el análisis de código es sustancialmente menos complicado ya que no es necesario contemplar todas las posibles variaciones de la estructura de código. Por otra parte, operar a nivel de ensamblador permite obtener la información necesaria sobre las CFIs válidas por adelantado y así facilitar la detección de errores preventiva.

Información que no se puede obtener fácilmente en un código de alto nivel y que además requeriría de una herramienta compleja comparable a un compilador.



(i): Disassemble
(ii): Transformar a una forma reconocible por PECOS
(a): Compilar hasta la fase de assembly
(b): Generar archivo assembly con PABs
(c): Compilar y linkear para generar el ejecutable

Camino si se comienza del archivo fuente (a, b, c)

Camino si se comienza del archivo ejecutable (i, ii, b, c)

FIGURA 2.2. Proceso de incorporación de PABs en una aplicación.

2.4. Recursos utilizados

La gran mayoría de los recursos utilizados, tales como tecnologías y herramientas, han sido acordados y, en algunos casos, provistos por el cliente. Si bien en esta sección se mencionan todos los recursos, no se hace un detalle sobre aquellos de carácter privativo.

2.4.1. Hardware

Uno de los objetivos fue lograr proteger funciones a ser ejecutadas en un microcontrolador SAMV71. Se utilizó SAM V71 Xplained Ultra [11] como placa de desarrollo y sus especificaciones pueden observarse en el catálogo de productos de su fabricante Microchip.

2.4.2. Software

Para desarrollar el firmware sobre el microcontrolador SAMV71 y como una guía de buenas practicas, se utilizó el entorno de desarrollo MPLAB X IDE [17] recomendado y diseñado por el fabricante Microchip.

Manteniendo el criterio de buenas practicas, para soportar el desarrollo sobre la arquitectura embebida ARMv7E-M se utilizó la herramienta Arm GNU Toolchain [18] para poder realizar compilación cruzada.

Para realizar los ensayos y validar el sistema se utilizó como herramienta de depuración tanto la toolchain como el IDE en donde se insertaron SE manualmente. Para automatizar el proceso de insertar SE y generar un ensayo más robusto, se utilizó una herramienta de inyección de errores por software denominada SISE provista por el cliente.

Capítulo 3

Diseño e implementación

Esta sección explica en detalle como construir un PAB y su interacción con el resto del sistema. Adicionalmente explica el desarrollo de software de monitoreo para verificar y validar el PAB desarrollado.

3.1. 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. Un ejemplo de un PAB completo puede observarse en el apéndice B.

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:

$$\exists n \in N : X_d = X_{sn} \tag{3.1}$$

Para lograrlo, el PAB calcula la dirección de destino verificada X como indica la ecuación 3.2 de alto nivel. Las tareas que debe realizar el PAB en tiempo de ejecución se enumeran a continuación y se detallan a lo largo de esta sección:

1. Determinar la dirección de destino X_d de la CFI en tiempo de ejecución. La sección 3.1.1 considera la situación donde la dirección destino de la CFI es estática.
2. Determinar la dirección de destino valida X_{sn} de la CFI en tiempo de ejecución. En general, 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.

$$X = \frac{X_d}{isValid} : isValid = ![(X_d - X_{s1})(X_d - X_{s2}) \dots (X_d - X_{sn})] \tag{3.2}$$

ARMv7E-M [3] es la arquitectura sobre la cual se desarrolló PECOS en el presente trabajo. Esta arquitectura solo soporta el conjunto de instrucciones *Thumb* 2 [14] que, a diferencia de *Thumb*, agrega codificaciones de 4 *bytes*.

3.1.1. Determinación de dirección de destino

La dirección de destino de la CFI es aquella que se especifica como el operando de la instrucción. Como se mencionó en la sección 2.3, 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 3.1 muestra como 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".

```
1 ldr r0, =.CFI1  @(1): Load runtime CFI's VMA (addrCFI).
2 ldr r1, [r0]    @(2): (2)-(4) Format register to big endian.
3 rev16 r1, r1    @(3)
4 rev r1, r1      @(4)
```

CÓDIGO 3.1. Extracción de CFI en la memoria del programa.

3.1.2. Determinación de dirección de destino válida

Partiendo de la codificación obtenida en la sección 3.1.1 se observa que ahora 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.

Se analiza en esta sección como construir una posible codificación de 16 *bits* para luego, de forma incremental, analizar una codificación en 32 *bits* para una determinada CFI. Específicamente se analiza la codificación T2 (figura 3.1) y T4 (figura 3.4) para la CFI *Branch*. Tanto la composición de la codificación como los algoritmos para lograrla se extrajeron del manual de la arquitectura embebida [16] y si bien no se explica el origen de la totalidad de los bits, se puede realizar ingeniería inversa para replicar su comportamiento.

Es importante destacar que la arquitectura ARMv7E-M cuenta con un *pipeline* de 3 etapas y que todos los algoritmos detallados en esta sección realizan operaciones sobre el PC, por lo que es clave entender como se comporta este registro y como interactúa con el pipeline. Para este punto se estudió la bibliografía de Joseph Yiu [19] sobre procesadores ARM Cortex de la cual se cita oportunamente:

"R15 is the Program Counter (PC). It is readable and writeable: a read returns the current instruction address plus 4 (this is due to the pipeline nature of the design, and compatibility requirement with the ARM7TDMI processor). Writing to PC (e.g., using data transfer/processing instructions) causes a branch operation.

Since the instructions must be aligned to half-word or word addresses, the Least Significant Bit (LSB) of the PC is zero. However, when using some of the branch/memory read

instructions to update the PC, you need to set the LSB of the new PC value to 1 to indicate the Thumb state."

Codificación T2 (16 *bits*)

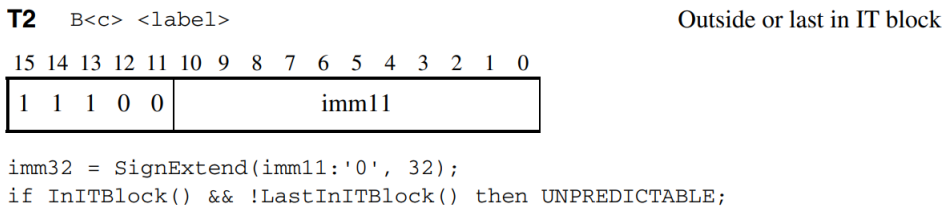
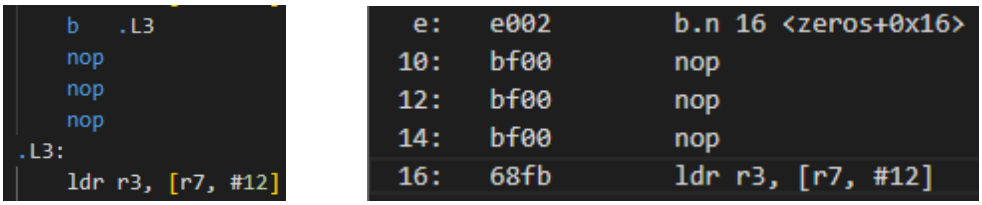


FIGURA 3.1. CFI: *Branch* - Codificación: T2 [1].

Se comienza con el análisis de la codificación T2 con salto hacia adelante (hacia direcciones de memoria crecientes) como se ejemplifica en la figura 3.2. Se enumeran a continuación los pasos necesarios para validar la codificación tal que *b .L3 = 0xE002*.

1. Partiendo de la figura 3.1 se observa que el comienzo 0xE0 corresponde perfectamente con el comienzo b'11100' de T2. Entonces *imm11 = 0x002*.
2. El salto va desde 0xE hasta 0x16, lo que es igual a un offset de 0x8 (0x16 - 0xE). Pero es necesario tener en cuenta que el PC se encuentra dos instrucciones delante (4 *bytes*), entonces el "offset verdadero" será 0x4. El manual de la arquitectura embebida se refiere a este "offset verdadero" como el inmediato de 32 bits o *imm32*.
3. Pero el valor 0x4 no corresponde con el valor 0x2 esperado. Esto se debe a un detalle implícito por diseño, y es que los saltos no pueden realizarse a direcciones impares de memoria, debido a que las instrucciones Thumb se encuentran alineadas a palabras de 2 *bytes*. Al poder hacer solo saltos pares, es posible ahorrarse el bit menos significativo del inmediato para referenciar al mismo offset, o lo que es igual a dividir por 2. Entonces *imm11 = 0x2* (0x4 / 2).



(A) *Assembly*. (B) *Listing*.

FIGURA 3.2. Análisis de CFI con salto hacia adelante para la codificación T2.

Se concluye que el operando inmediato de 11 bits para la codificación T2 responde a:

$$imm11 \cdot 2 = imm32 \tag{3.3}$$

[1] Imagen tomada de [16].

Lo que en el manual de la arquitectura se especifica, con extensión de signo explícita, como:

$$imm32 = SignExtend(imm11 : `0`, 32) \tag{3.4}$$

Siendo *imm32*, por definición, la diferencia entre la dirección de destino del salto respecto del PC al momento de ejecutar el salto, se obtiene:

$$imm32 = addrTarget - (addrCFI + 4) \tag{3.5}$$

Análogamente, se validaron las ecuaciones para un salto hacia atrás (hacia direcciones de memoria decrecientes) en *complemento de 2* como se ejemplifica en la figura 3.3. Se enumeran a continuación los pasos necesarios para validar la codificación tal que *.L3: b .L3 = 0xE7FE*.

1. El comienzo 0xE7 corresponde perfectamente con el comienzo b'11100' de T2. Entonces *imm11 = 0x7FE (-2)*.
2. El salto va desde 0xE hasta 0xE, lo que es igual a un offset de 0x0, lo cual es lógico si el salto se realiza al mismo lugar. Pero es necesario tener en cuenta que el PC se encuentra dos instrucciones adelante (4 bytes), entonces *imm32 = 0xFFFF FFFC (-4)*.
3. Dividir por 2 se asimila a desplazar con extensión de signo hacia la derecha, por lo que *imm11* será 0x7FE (-2). Lo cual coincide perfectamente con *imm11* del *listing*.

(A) *Assembly*. (B) *Listing*.

FIGURA 3.3. Análisis de CFI con salto hacia atrás para la codificación T2.

Codificación T4 (32 *bits*)

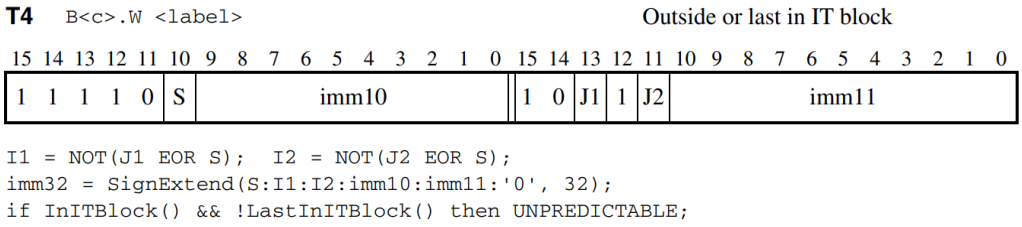


FIGURA 3.4. CFI: *Branch* - Codificación: T4 [2].

Se continúa con el análisis de la codificación T4 con salto hacia adelante como se ejemplifica en la figura 3.5. En lugar de analizar esta codificación como una única palabra de 4 bytes, se analizó según lo describe el manual, como dos medias palabras de 2 bytes concatenadas. Se enumeran a continuación los pasos necesarios para validar la codificación tal que *b.w .L3 = 0xF000 B803*.

[2]Imagen tomada de [16].

1. Partiendo de la figura 3.4 se observa que el comienzo b'11110' de la primera (mas significativa) media palabra indica que es una instrucción indefinida (para arquitecturas que no sean ARMv6T2, ARMv7) o una extensión de Thumb 2 (para arquitecturas ARMv6T2, ARMv7). La segunda (menos significativa) media palabra debe comenzar con b'10X1'. Se observa que en su lugar hay 0xB por lo que es un valor válido y corresponde a una codificación T4.
2. Partiendo de la ecuación 3.5, se debe verificar que: $imm32 = 0x18 - (0xE + 0x4) = 0x6$.
3. Del binario codificado en la figura 3.5 se observa que: *S = 0*, *imm10 = 0*, *J1 = 1*, *J2 = 1*, y *imm11 = 0x3*. Con estos datos se debe hacer ingeniería inversa para calcular I1, I2 y armar finalmente imm32. Los algoritmos para lograrlo también se indican en la figura 3.4 y se obtiene que *I1 = 0*, *I2 = 0*, y *imm32 = 0x6* (0:0:0:0:0x3:0) lo cual verifica perfectamente.

```
    b.w .L3
    nop
    nop
    nop
.L3:
    ldr r3, [r7, #12]
```

(A) *Assembly*.

```
 e:   f000 b803   b.w 18 <zeros+0x18>
12:   bf00        nop
14:   bf00        nop
16:   bf00        nop
18:   68fb        ldr r3, [r7, #12]
```

(B) *Listing*.

FIGURA 3.5. Análisis de CFI con salto hacia adelante para la codificación T4.

Queda en evidencia que se deberán desarrollar PABs con algoritmos de reconstrucción binaria distintos según la codificación de la CFI.

Se pueden calcular los parámetros de la codificación T4 sabiendo que el valor de *imm32* responde a:

$$imm32 = SignExtend(S : I1 : I2 : imm10 : imm11 : \text{`0'}, 32); \tag{3.6}$$

Lo que se traduce a:

$$S = imm32 < 24 > \tag{3.7}$$

$$I1 = imm32 < 23 > \tag{3.8}$$

$$I2 = imm32 < 22 > \tag{3.9}$$

$$imm10 = imm32 < 21 : 12 > \tag{3.10}$$

$$imm11 = imm32 < 11 : 1 > \tag{3.11}$$

Si:

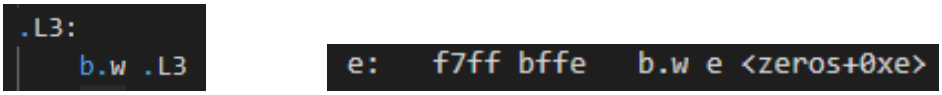
$$Ix = NOT(Jx\ EOR\ S) \tag{3.12}$$

Entonces:

$$Jx = NOT(Ix\ EOR\ S) \tag{3.13}$$

Análogamente, se validaron las ecuaciones para un salto hacia atrás en *complemento a 2* como se ejemplifica en la figura 3.6. Se enumeran a continuación los pasos necesarios para validar la codificación tal que *.L3: b.w .L3 = 0xF7FF BFFE*.

1. El comienzo b'11110' de la primera media palabra indica que es una instrucción indefinida (para arquitecturas que no sean ARMv6T2, ARMv7) o una extensión de Thumb 2 (para arquitecturas ARMv6T2, ARMv7). La segunda media palabra debe comenzar con b'10X1'. Se observa que en su lugar hay 0xB por lo que es un valor válido y corresponde a una codificación T4.
2. Partiendo de la ecuación 3.5, hay que verificar que: $imm32 = 0xE - (0xE + 0x4) = 0xFFFFFFFC$.
3. Del binario codificado en la figura 3.6 se observa que: *S = 1*, *imm10 = 0x3FF*, *J1 = 1*, *J2 = 1*, y *imm11 = 0x7FE*. Entonces *I1 = 1*, *I2 = 1*, y *imm32 = 0xFFFF FFFC* (1:1:1:0x3FF:0x7FE:0) lo cual verifica perfectamente.



(A) *Assembly*. (B) *Listing*.

FIGURA 3.6. Análisis de CFI con salto hacia atrás para la codificación T4

Bloque de código ejemplo

El bloque de código 3.2 muestra como se puede reconstruir una codificación T4 válida de una CFI *Branch*.

```
1  ldr r2, =.L3          @(5): Load CFI target address (addrTarget).
2  add r0, r0, #4        @(6): Add PC relative offset to CFI's VMA.
3  sub r2, r2, r0        @(7): imm32 = addrTarget - (addrCFI + 4)
4  mov r3, #0            @(8): Clean register to store valid CFI.
5  mov r4, r2, lsr #1    @(9): (9)-(11) Take imm11 from imm32.
6  mov r0, #0x7FF        @(10)
7  and r4, r4, r0        @(11)
8  orr r3, r3, r4        @(12): Load imm11.
9  mov r4, r2, lsr #12   @(13): (13)-(15) Take imm10 from imm32.
10 mov r0, #0x3FF        @(14)
11 and r4, r4, r0        @(15)
12 mov r4, r4, lsl #16   @(16): (16)-(17) Place and load imm10.
13 orr r3, r3, r4        @(17)
14 mov r0, r2, lsr #24   @(18): (18)-(19) Take S from imm32.
15 and r0, r0, #0x1      @(19)
16 mov r4, r0, lsl #26   @(20): (20)-(21) Place and load S.
17 orr r3, r3, r4        @(21)
18 mov r4, r2, lsr #22   @(22): (22)-(23) Take I2 from imm32.
19 and r4, r4, #0x1      @(23)
20 eor r4, r4, r0        @(24): (24)-(26) J2 = not (I2 xor S)
21 mvn r4, r4            @(25)
22 and r4, r4, #0x1      @(26)
23 mov r4, r4, lsl #11   @(27): (27)-(28) Place and load J2
24 orr r3, r3, r4        @(28)
25 mov r4, r2, lsr #23   @(29): (29)-(30) Take I1 from imm32.
26 and r4, r4, #0x1      @(30)
27 eor r4, r4, r0        @(31): (31)-(33) J1 = not (I1 xor S)
28 mvn r4, r4            @(32)
29 and r4, r4, #0x1      @(33)
30 mov r4, r4, lsl #13   @(34): (34)-(35) Place and load J1
31 orr r3, r3, r4        @(35)
32 ldr r4, =#0xF0009000 @(36): (36)-(37) Load CFI's nemonic opcode.
33 orr r3, r3, r4        @(37)
```

CÓDIGO 3.2. Reconstrucción de CFI válida.

3.1.3. Validación de salto de programa

Habiendo obtenido la CFI en memoria de programa en la sección 3.1.1 y reconstruido la CFI válida en la sección 3.1.2, 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 3.2 y disparar una excepción de división por 0 en caso de detectar una diferencia. El bloque de código 3.3 ejemplifica como realizar la validación.

```
1 sub r3, r3, r1        @(38): Should store zero for correct execution.
2 cmp r3, #0            @(39): (39)-(43) rx = !rx
3 ite eq                @(40)
4 moveq r3, #1          @(41)
5 movne r3, #0          @(42)
6 uxtb  r3, r3          @(43)
7 sdiv  r3, r1, r3      @(44): Raises exception if rx was not zero.
```

CÓDIGO 3.3. Validación de CFI.

Notese que hasta aquí llega el alcance de la técnica PECOS. Esta técnica no define un prototipo de manejador de falla y queda a criterio del diseñador el desarrollo del mismo.

3.1.4. 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:

$$C = \frac{a}{n} \cdot 100\,\% \tag{3.14}$$

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.

3.2. PECOS Monitor

Al no contar con un prototipo de manejador de falla para PECOS, una excepción de división por 0 es atendida por el procesador con los manejadores de fallas propios de la arquitectura. Este procedimiento no es trivial ya que es posible habilitar y deshabilitar ciertos manejadores o simplemente enmascarar la atención de señales de falla particulares. Por lo que ante un mismo error, se pueden disparar acciones diferentes en el procesador que hacen difícil depurar el mismo. Para homogeneizar la depuración se creó el modulo de software "PecosMonitor" con la intensión de unificar el manejo de depuración y generar reportes que permitan al usuario comprender la falla. Esta solución permite imprimir por un puerto serie el *stack frame* del procesador, al momento de ingresar al manejador, donde se encuentra la información necesaria para poder rastrear el origen de la falla. Para sistemas que manejen distintos niveles de privilegio, ergo distintos SP, se debe considerar el nivel de privilegio en el que se origina la falla ya que el stack frame será distinto. Este análisis fue contemplado siguiendo la guía de Chris Coleman [20]. De esta forma se espera que el responsable de la prueba pueda ver rápidamente los registros de interés (i.e.: PC, LR) con sus valores, pero no le quita la necesidad de conocer el mapa de memoria de programa donde se descargó el firmware, ya que mediante el mismo podrá identificar la instrucción que originó la falla mediante su dirección de memoria.

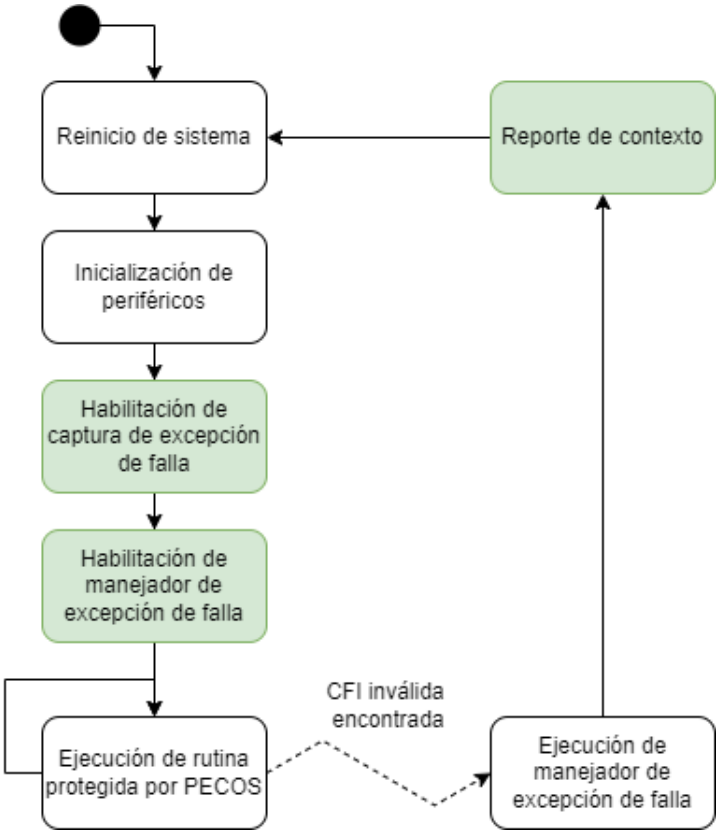


FIGURA 3.7. Diagrama de flujo de actividades del DUT, señalizando en verde el software generador de ambiente de ensayo.

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. Esta adición se ilustra en bloques verdes en la figura 3.7.

El software para generar un ambiente de ensayo utilizará mayormente recursos de la arquitectura del DUT. ARMv7E-M cuenta con el *System Control Block (SCB)*, que controla las excepciones de falla y provee la información de su causa, como también con dos punteros de pila *Main Stack Pointer (MSP)* y *Process Stack Pointer (PSP)* para obtener el contexto de ejecución.

3.2.1. Configuración de manejador de excepción de falla

Se describe a continuación únicamente el manejador de excepción de falla de interés, en este caso *UsageFault*, el cual detecta la ejecución de instrucciones indefinidas (i.e., una división por 0) y accesos a memoria desalineados.

El SCB provee registros de configuración, control y reporte de las excepciones de sistema. "PecosMonitor" hace uso de algunos de estos registros tales como:

- *Configuration and Control Register (CCR)*: controla el comportamiento de *UsageFault*. El bit *DIV_0_TRP* indica, cuando vale 1, que se encuentra habilitada la captura de una división por 0 (con o sin signo).

TABLA 3.1. Configuration and Control Register (CCR).

Registro	Dirección	Acceso	Valor por reinicio
CCR	0xE000ED14	R/W privilegiado	0x00000000

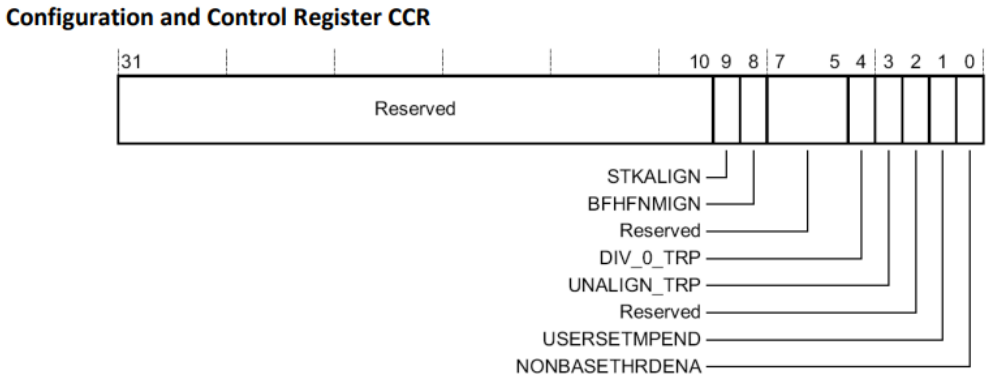


FIGURA 3.8. Asignación de bits del registro SCB ->CCR [3].

- *System Handler Priority Registers (SHP)*: controla la prioridad de la excepción. Se utilizaron los valores por defecto ya que permiten que *HardFault* siga siendo más prioritaria que *UsageFault* y pueda ejecutarse en caso de ocurrir un problema imprevisto.

TABLA 3.2. System Handler Priority Registers (SHP).

Registro	Dirección	Acceso	Valor por reinicio
SHP[12]	0xE000ED18	R/W privilegiado	0x00

- *System Handler Control and State Register (SHCSR)*: habilita los manejadores del sistema. Cuando *USGFAULTENA* vale 1 *UsageFault* está habilitada.

[3]Imagen tomada de [25].

TABLA 3.3. System Handler Control and State Register (SHCSR).

Registro	Dirección	Acceso	Valor por reinicio
SHCSR	0xE000ED24	R/W privilegiado	0x00000000

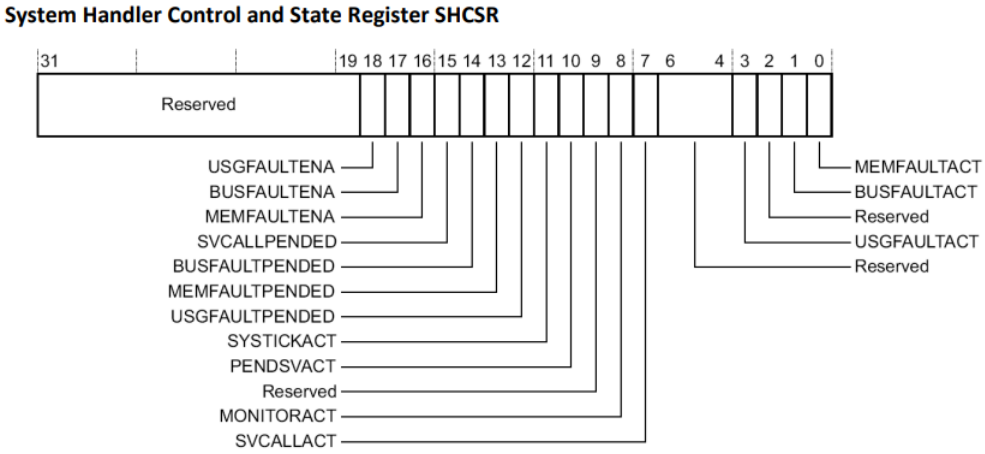


FIGURA 3.9. Asignación de bits del registro SCB ->SHCSR [4].

3.2.2. Registro de estado del manejador de excepción de falla

Los registros de estado de los manejadores *UsageFault*, *BusFault* y *MemManage* están contenidos dentro de un mismo registro de estado *Configurable Fault Status Register (CFSR)* con la disposicion que se muestra en la figura 3.10.

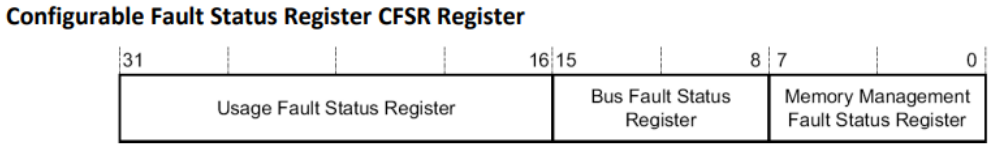


FIGURA 3.10. Asignación de bits del registro SCB ->CFSR [4].

Se analiza a continuación únicamente el registro de interés *Usage Fault Status Register (UFSR)* que se muestra en la figura 3.11. El bit *DIVBYZERO* es escrito por el procesador a 1 cuando el mismo ejecuta una división por 0 (con o sin signo), cuando esto ocurre el valor del PC que apunta a la instrucción que originó la falla se inserta en el stack frame y esto es lo que debe analizarse para verificar que la falla vino de un PAB. Un acceso no privilegiado generará una excepción del manejador *BusFault*.

TABLA 3.4. Usage Fault Status Register (UFSR).

Manejador	Registro de estado	Dirección	Acceso
UsageFault	UFSR	0xE000ED2A	R/W privilegiado

[4]Imagen tomada de [25].

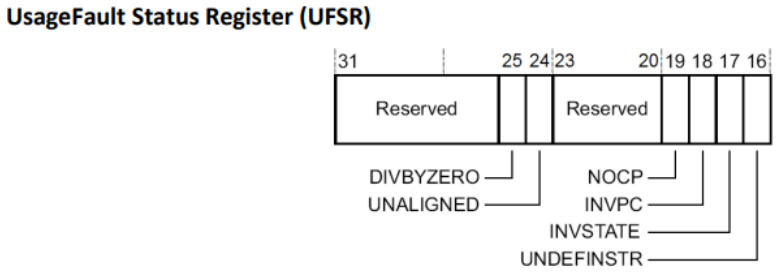


FIGURA 3.11. Asignación de bits del registro UFSR [5].

3.2.3. Reporte de contexto de ejecución

Las fallas relacionadas a *UsageFault* siempre son sincrónicas y precisas, lo que significa que el hardware es capaz de determinar la ubicación exacta de la falla. Esto asegura que los registros guardados en el *stack frame* siempre apunten al código que originó la falla.

Pueden ocurrir fallas que no estén relacionadas al PAB, por eso es importante que el manejador pueda descubrir y actuar según el origen de la falla. Para lograrlo se debe recuperar el estado de los registros del procesador al momento que se generó la excepción.

Al ingresar al manejador de excepción de falla, algunos registros siempre se guardan automáticamente en la pila. Dependiendo si se está utilizando o no una *Floating-point Unit (FPU)*, la arquitectura ARM carga un stack frame extendido o básico respectivamente, como se observa en la figura 3.12. Estos son los registros que deben analizarse durante la ejecución del manejador de excepción de falla. Si la FPU está habilitada pero no se está ejecutando ninguna instrucción de punto flotante o si *ASPEN* está deshabilitada, solo se carga el stack frame básico [21].

TABLA 3.5. Registro EXC_RETURN.

Valor de EXC_RETURN	Descripción
0xFFFFFFF1	Retorna a modo Handler. El retorno de la excepción toma el estado de la pila principal. Luego de retornar, la ejecución continua utilizando MSP.
0xFFFFFFF9	Retorna a modo Thread. El retorno de la excepción toma el estado de la pila principal. Luego de retornar, la ejecución continua utilizando MSP.
0xFFFFFFFD	Retorna a modo Thread. El retorno de la excepción toma el estado de la pila del proceso. Luego de retornar, la ejecución continua utilizando PSP.
Cualquier otro valor	Reservado.

[5] Imagen tomada de [25].

Los dispositivos ARM Cortex-M tienen dos punteros de pila *Main Stack Pointer (MSP)* y *Process Stack Pointer (PSP)*. Al ingresar al manejador de excepción, el puntero de pila que estaba activo se indica en el bit 2 del valor *EXC_RETURN* guardado en el registro LR por el procesador. Si dicho bit se encuentra en 1, entonces PSP estaba activo, de lo contrario MSP estaba activo. El comportamiento completo se ilustra en la tabla 3.5.

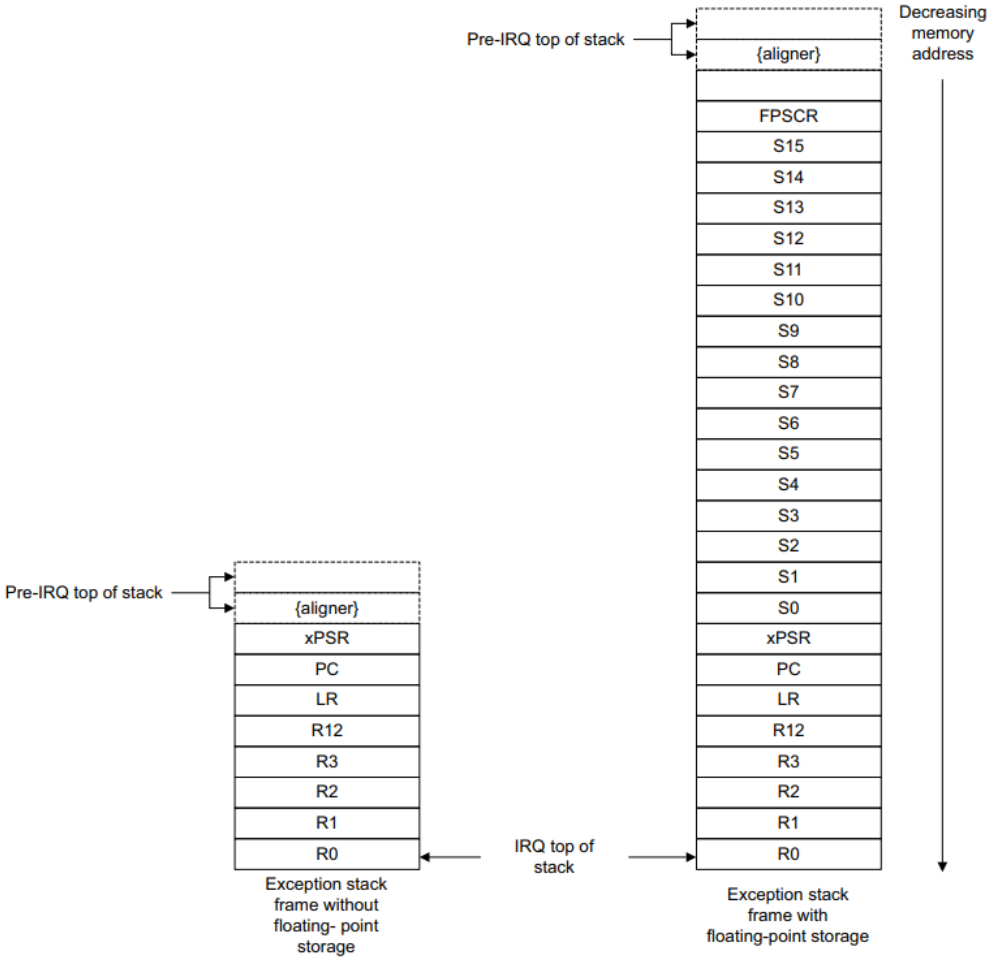


FIGURA 3.12. *Stack frame* ante una excepción sin y con almacenamiento de FPU [6].

3.2.4. Automatización del análisis

Todo el análisis de registros explicado en las sub-secciones anteriores puede realizarse manualmente con una herramienta de depuración sobre el procesador, un proceso que al tener intervención humana suele ser lento y propenso a errores. Pero además, los análisis de eventos de radiación son difíciles de depurar ya que son difíciles de reproducir, incluso contando con una herramienta de depuración algunos eventos y valores de registros podrían perderse o sobre-escribirse antes de que sea posible detener el procesador para inspeccionarlo. Por tal motivo es que automatizar el análisis de las fallas es prácticamente un deber.

[6] Imagen tomada de [21].

La solución adoptada fue crear una clase de monitoreo *PecosMonitor* bajo el paradigma de programación orientada a objetos (OOP) [22], que integre funcionalidades de ASM extendido provistas por GCC [23] para conservar eventos clave, e.g., el valor de *EXC_RETURN* al ingresar a un manejador de excepciones de falla. El diagrama de clases de PecosMonitor se observa en la figura 3.13. La lógica para lidiar con las excepciones de falla es típicamente:

- Imprimir el mensaje y el estado de falla para un análisis *postmortem*.
- Si la falla es recuperable, entonces limpiar el error y retornar a un modo seguro. Si la falla no es recuperable, entonces reiniciar el sistema.

La clase PecosMonitor permite imprimir por puerto serie el stack frame independientemente de contar con una herramienta de depuración y elevar el manejo de ASM a lenguaje C para poder facilitar y extender el análisis.

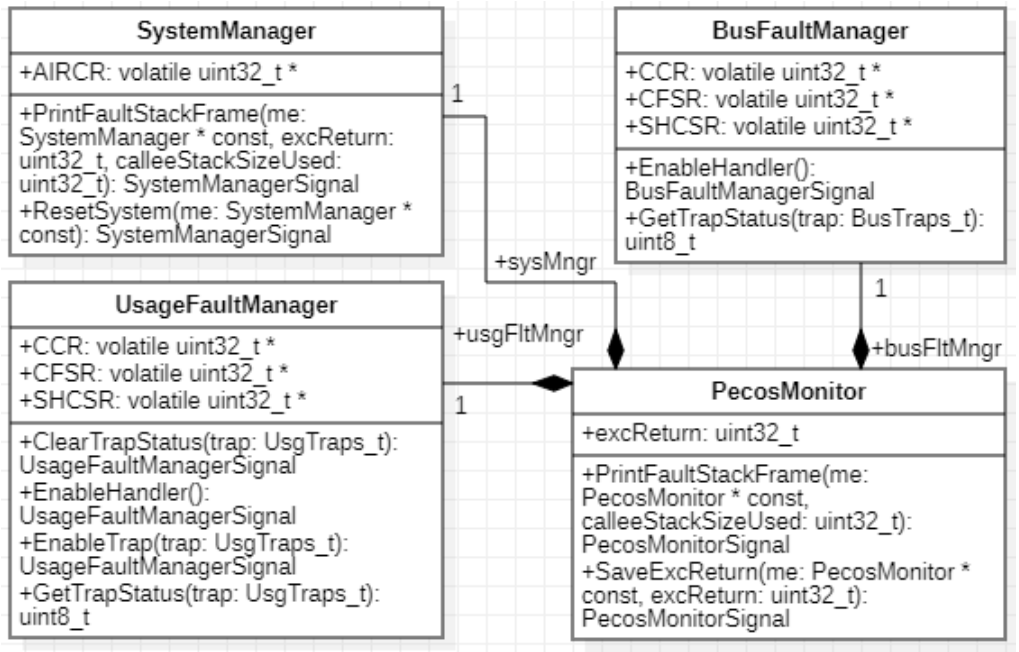


FIGURA 3.13. Diagrama de clases del software de ambiente de ensayo PECOS.

Capítulo 4

Ensayos y resultados

Esta sección explica como se condujeron los ensayos para validar el trabajo y sus resultados. Se describe la composición del ambiente de ensayo, las actividades del DUT y la técnica de diseño de prueba implementada.

4.1. Ambiente de ensayo

El ambiente de ensayo utilizado consiste en un esquema de conexión trivial al trabajar con placas de desarrollo. La figura 4.1 ilustra la conexión que permite comunicar al ordenador con el microcontrolador a través de una sonda de depuración.

Teniendo disponible el hardware provisto por el cliente de manera física, se decidió abordar un esquema de ensayos donde, sobre la implementación real del hardware, se simulen los efectos de la radiación mediante la herramienta SISE también provista por el cliente. Las características de este escenario de inyección de fallas, junto a otros, se observan en la tabla 4.1.

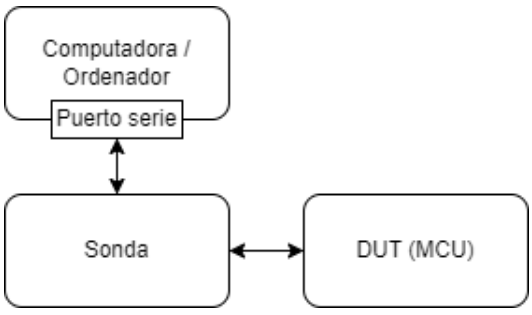


FIGURA 4.1. Diagrama en bloques de conexiones del ambiente de ensayo.

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. SISE es instrumentada sobre una consola de texto plano que 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. Estos errores representan modelos tales como:

- Transmisión de errores en el *bus* entre una memoria y el núcleo del procesador.
- Errores en el disco, la memoria principal o memoria cache dentro del *chip*.

- Errores de software. Estudios previos [10] han demostrado que un gran conjunto de SEs puede imitarse mediante bit flips de memoria.

TABLA 4.1. Comparación de escenarios de inyección de fallas.

Hardware	Radiación	Características
Simulado	Simulada	Costos bajos. Eficiencia baja. Ofrece flexibilidad máxima ya que se basa en modelos de hardware (HDL). Requiere gran capacidad de procesamiento y de tener disponible los modelos de hardware. Requiere un simulador de instrucciones del procesador del DUT.
Implementado	Simulada	Costos medios. Eficiencia media. Poco tiempo de ensayo. Requiere un prototipo de hardware. Limitaciones para inyectar SEU a determinados objetivos. Utiliza un modo de ejecución particular del procesador (i.e.: *debugging*).
Implementado	Implementada	Costos altos. Eficiencia alta. Ensayos cercanos a la realidad. Requiere equipamiento de radiación. Poco control sobre la locación y el tiempo de inyección de la falla.

4.1.1. Rutina del DUT.

La rutina del DUT consiste en ejecutar, dentro de un bucle infinito, la función desarrollada VarianTheFirst (ver apéndice A) 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. Recordar que la ejecución de la rutina del DUT es la parte más importante de todas actividades que debe realizar el mismo, vistas en la figura 3.7.

4.2. Técnica de diseño de prueba

La técnica utilizada fue *Control Flow Test (CFT)* [24]. 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. Los pasos para desarrollar esta técnica consisten en:

1. Realizar un inventario de los puntos de decisión.
2. Determinar los caminos de prueba.
3. Especificar los casos de uso.

4. Establecer el conjunto de datos iniciales.
5. Construir el *script* de prueba.
6. Ejecutar la prueba.

4.2.1. Puntos de decisión

La figura 4.2 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 serán etiquetadas aunque se ilustren separadas para comprender el detalle individual de cada una de ellas.

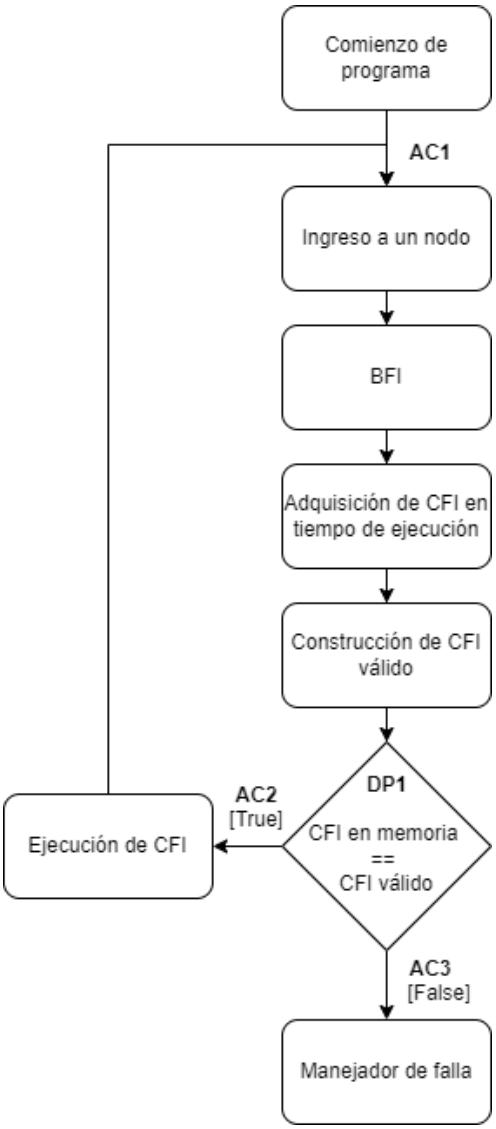


FIGURA 4.2. Diagrama de flujo de la estructura de un nodo del CFG con un PAB.

4.2.2. Caminos de prueba

La combinación de acciones en caminos de pruebas depende del nivel de profundidad de prueba deseado. Dicho nivel influye directamente en el número de casos de prueba y el nivel de cobertura de las mismas.

El nivel de profundidad de prueba utilizado para el PAB fue de 1 al tener una representación trivial como se observa en la figura 4.2.

La combinación de acciones para los puntos de decisión son entonces:

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

Esto lleva a la siguiente combinación de caminos:

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

4.2.3. Casos de prueba

Para cada camino de prueba se debe determinar el conjunto de parámetros de entrada de la prueba. Los parámetros de entrada deben ser tales que permitan recorrer todos los caminos posibles en un punto de decisión. Los casos de prueba deben también especificar resultados de salida esperados. Aquellos parámetros que no tengan impacto alguno sobre el camino deben recibir un valor por defecto. Se desarrollaron 3 casos de pruebas para el presente trabajo detallados en las tablas 4.2, 4.3 y 4.4.

4.2.4. Conjunto de datos iniciales

La ejecución de los caminos de prueba especificados en la sección 4.2.2 no requieren un conjunto especifico de datos iniciales que no se hayan descripto dentro de los mismos caminos de pruebas y casos de pruebas.

4.2.5. Script de prueba

El script de prueba describe la secuencia correcta en que deben ejecutarse los casos de prueba y sus verificaciones. También debe contar con un listado de precondiciones para ejecutarse, estas condiciones están relacionadas al conjunto de datos iniciales. Al no necesitar especificar precondiciones adicionales a las especificadas en los casos de pruebas de la sección 4.2.3 y al no estar dichos casos de pruebas sujetos a ejecutarse en un orden particular, no fue necesario desarrollar un script de prueba.

4.2.6. Ejecución de ensayo

El ensayo debe ser ejecutado acorde al script de prueba, comparando su resultado con el esperado. Para el presente trabajo y en ausencia de dicho script, el responsable del ensayo es libre de ejecutar los casos de prueba en cualquier orden.

Titulo	Descripción
1. Nombre	PECOS: ausencia de error.
1.1. Breve descripción	Se ejecuta una función (ver apéndice A) protegida por el software de mitigación para validarlo. No se espera encontrar un error durante la ejecución por lo que se espera una terminación exitosa de la función.
1.2. Actor principal	Procesador.
1.3. Disparadores	Flujo de control del programa.
2. Flujo de eventos	
2.1. Flujo básico	1. Se inicia la secuencia de programa. 2. El software decide ejecutar la función protegida. 3. La función ejecuta su código base, ingresando al primer nodo de su gráfico de flujo de control que se encuentra protegido con software de mitigación. 4. La función ejecuta el Bloque de Aserción PECOS previo a la instrucción de control de flujo (CFI). 5. La aserción confirma que no hay errores en los datos del programa. 6. La función ejecuta la CFI del nodo con normalidad. 7. La función retorna al proceso que la llamó el resultado esperado.
2.2. Flujo alternativo	5.1. La aserción confirma que sí hay errores en los datos del programa. Este flujo alternativo representa el flujo básico del caso de prueba "PECOS: presencia de error". Para mas detalle, referirse al mismo.
3. Precondiciones	1. El procesador debe contar con el programa a evaluar. 2. El procesador debe estar alimentado y funcional.
4. Poscondiciones	1. El procesador finaliza la ejecución de la función de manera exitosa. 2. La función retorna el valor esperado.

TABLA 4.2. Caso de prueba 1. PECOS: ausencia de error.

Titulo	Descripción
1. Nombre	PECOS: presencia de error.
1.1. Breve descripción	Se ejecuta una función (ver apéndice A) protegida por el software de mitigación para validarlo. Se inyecta manualmente un error mediante la herramienta de depuración por lo que se espera una interrupción en la ejecución de la función que será atendida por un manejador de fallas.
1.2. Actor principal	Procesador con intervención de usuario mediante interfaz de depuración.
1.3. Disparadores	1. Flujo de control del programa. 2. Responsable del ensayo.
2. Flujo de eventos	
2.1. Flujo básico	1. Se inicia la secuencia de programa mediante una herramienta de depuración que permita obtener la vista de ejecución a nivel de lenguaje ensamblador. 2. Se coloca un *breakpoint* en la instrucción *sdiv* dentro de un Bloque de Aserción PECOS. 3. El software decide ejecutar la función protegida. 4. La función ejecuta su código base, ingresando al primer nodo de su gráfico de flujo de control que se encuentra protegido con software de mitigación. 5. La función ejecuta el Bloque de Aserción PECOS previo a la instrucción de control de flujo (CFI). 6. La herramienta de depuración detiene la ejecución del programa en el *breakpoint*. 7. El responsable del ensayo modifica el registro que cumple el rol de divisor de la operación y le asigna el valor 0. 8. El responsable del ensayo reanuda la ejecución del programa. 9. La aserción confirma que sí hay errores ejecutando una operación de división por 0. Operación que dispara un manejador de excepción en el procesador para ser atendida. 10. El manejador de la excepción determina que la causa fue sobre un bloque de aserción. 11. El manejador realiza las acciones necesarias para no perder el control del programa. 12. El manejador finaliza.
2.2. Flujo alternativo	9.1. La aserción no detecta el error inyectado y no se dispara una excepción de división por 0. Este flujo alternativo representa el flujo básico del caso de prueba "PECOS: ausencia de error". Para mas detalle, referirse al mismo.
3. Precondiciones	1. El procesador debe contar con el programa a evaluar. 2. El procesador debe estar alimentado y funcional.
4. Poscondiciones	1. El procesador finaliza la ejecución de la función de manera forzada (no exitosa). 2. El procesador termina en un manejador de falla.

TABLA 4.3. Caso de prueba 2. PECOS: presencia de error.

Titulo	Descripción
1. Nombre	PECOS + SISE + PecosMonitor.
1.1. Breve descripción	Se ejecuta una función (ver apéndice A) protegida por el software de mitigación (PECOS) para validarlo con la ayuda de una herramienta de inyección de SE (SISE). Se espera perturbar la rutina de ejecución del DUT y observar su comportamiento a través de una consola serial (PecosMonitor).
1.2. Actor principal	Procesador con intervención de SISE.
1.3. Disparadores	1. Flujo de control del programa. 2. Herramienta de inyección de SE.
2. Flujo de eventos	
2.1. Flujo básico	1. Se inicia la secuencia de programa. 2. El software decide ejecutar la función protegida. 3. Se inicia una sesión de inyección de SEs con SISE. 4. SISE inyecta un SE en un registro del procesador (r0 - r12) de forma aleatoria. 5. La función ejecuta su código base, ingresando al primer nodo de su CFG que se encuentra protegido con software de mitigación. 6. La función ejecuta el PAB del nodo previo a la CFI. 7. La aserción confirma que no hay errores en los datos del programa. 8. La función ejecuta la CFI del nodo con normalidad. 9. La función retorna al proceso que la llamó el resultado esperado.
2.2. Flujo alternativo	7.1. La aserción confirma que sí hay errores ejecutando una operación de división por 0. Operación que dispara un manejador de excepción en el procesador para ser atendida. 7.2. El manejador de la excepción determina que la causa fue sobre un bloque de aserción. 7.3. El manejador de la excepción imprime el reporte del stack frame por puerto serie. 7.4. El manejador realiza las acciones necesarias para no perder el control del programa. 7.5. El manejador finaliza.
3. Precondiciones	1. El procesador debe contar con el programa a evaluar. 2. El procesador debe estar alimentado y funcional. 3. El DUT debe estar configurado como objetivo de la herramienta SISE.
4. Poscondiciones	1. El procesador finaliza la ejecución de la función de manera elegante. 2. SISE finaliza su sesión de inyección de SEs.

TABLA 4.4. Caso de prueba 3. PECOS + SISE + PecosMonitor.

4.2.7. Ejemplo de prueba unitaria

Para verificar el desarrollo de la sección 3.2, se procedió a:

1. Insertar manualmente una falla en un PAB dentro de las funciones de prueba desarrolladas (ver apéndice A para mas detalle).
2. Capturar la salida del bloque de reporte de contexto (ver figura 3.7) por el puerto serie.
3. Validar los datos con el archivo listing generado en la compilación del firmware (ver figura 4.3).

Se observó que el valor de PC puesto en el stack frame, que aloja la dirección de memoria de la instrucción que originó la falla, coincide con la instrucción de división *sdiv* del PAB (vista previamente en la seccion 3.1.3). También se observó que el valor de retorno en LR puesto en el stack frame coincide con la siguiente instrucción a la llamada de la función de prueba *VarianTheFirst* (recordar que el bit menos significativo se encuentra en 1 para indicar el estado Thumb del procesador).

```
8003eba:    fb91 f3f3    sdiv    r3, r1, r3
```

(A) Instrucción de división en PAB vista desde archivo listing.

```
8000930:    f003 fa6e    bl  8003e10 <VarianTheFirst>
 {
8000934:    e7ed         b.n 8000912 <main+0x3a>
```

(B) Llamada a función de prueba VarianTheFirst vista desde archivo listing.

```
[Fault Stack Frame]
r0 = 0x0
r1 = 0xf000b817
r2 = 0x30
r3 = 0x0
r12 = 0xe1000000
LR = 0x8000935
PC = 0x8003eba
PSR = 0x21000200
```

(C) Reporte de falla capturado por puerto serie.

FIGURA 4.3. Análisis de reporte de falla con archivo listing.

4.3. Pruebas de aceptación

Si bien todos los casos de pruebas conformaron las pruebas de aceptación con el cliente, el caso de prueba con mayor protagonismo fue el número 3 (ver tabla 4.4). Su principal diferenciador es la inyección de errores de software mediante una herramienta dedicada como reemplazo a la intervención manual del responsable del ensayo.

4.3.1. Resultados

La tabla 4.5 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 SI-SE.

TABLA 4.5. Resultados de las inyección de errores de software en la aplicación.

Categoría	Sin PECOS	Con PECOS	Factor de mejora [(valor medido sin PECOS) / (valor medido con PECOS)]
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 4.5 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.

Capítulo 5

Conclusiones

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 en 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.

Por ultimo, se destaca el cumplimiento de los objetivos del trabajo como así también de las expectativas y requerimientos del cliente en el tiempo planificado con los recursos previstos.

5.1. Trabajo futuro

Las investigaciones realizadas durante el desarrollo del presente trabajo delatan las siguientes funcionalidades como posibles mejoras del mismo:

- El proceso de desarrollar e incorporar PABs es tedioso en términos de horas invertidas por una persona para lograrlo. Sería adecuado desarrollar una herramienta que permita automatizar el proceso y sea adaptable a varias ISAs con cambios mínimos.
- Se observó durante el desarrollo que la toolchain de ARM siempre codificó los operandos de las CFI con etiquetas. Agregaría datos estadísticos analizar situaciones donde el operando se encuentre dentro del valor de un registro o donde la CFI corresponda al llamado de una función perteneciente a una librería dinámica.

Apéndice A

Funciones de prueba en lenguaje C a ser protegidas

En el *listing* A.1 se observan las funciones en lenguaje C desarrolladas para realizar pruebas básicas de mitigación de errores de software. La función *VarianTheFirst* realiza un incremento unitario circular de 0 a 3 sobre los elementos de un vector. Esta función cuenta con varias CFIs a proteger, entre ellas la llamada a la función *AnduinTheSecond* que se encarga únicamente del incremento unitario del elemento del vector. La implementación de *AnduinTheSecond* es trivial y no posee ninguna CFI, por lo que los bloques PECOS se encuentran en su totalidad dentro de *VarianTheFirst*.

AnduinTheSecond se implementó en una función aparte para poder designarla, mediante el linker script, a una sección de código independiente lejos de la sección de código donde se encuentre *VarianTheFirst* y así poder hacer pruebas para saltos lejanos.

```
1  void AnduinTheSecond(uint32_t *value){
2      ++(*value);
3  }
4
5  void VarianTheFirst(uint32_t *vector, uint32_t length){
6      for(int i = 0; i<length; ++i){
7          if(vector[i] >= 3){
8              vector[i] = (uint32_t) 0;
9          }else{
10             AnduinTheSecond(&vector[i]);
11         }
12     }
13 }
```

CÓDIGO A.1. Código de prueba a proteger (C).

Apéndice B

Bloque de aserción PECOS

El bloque de código B.1 muestra un ejemplo completo de un PAB desarrollado para proteger la CFI ".CFI1".

```
1  @Begin PECOS Block.
2  @T4 Branch Encoding.
3  @Symbolic Operands
4  @Step A: Save context.
5    push  {r0-r4}          @(A1): Save used registers.
6  @Step 1: Load original CFI.
7    ldr r0, =.CFI1         @(1): Load runtime CFI's VMA (addrCFI).
8    ldr r1, [r0]           @(2): (2)-(4) Format register to big endian.
9    rev16 r1, r1           @(3)
10   rev r1, r1             @(4)
11 @Step 2: Build valid CFI.
12   ldr r2, =.L3           @(5): Load CFI target address (addrTarget).
13   add r0, r0, #4         @(6): Add PC relative offset to CFI's VMA.
14   sub r2, r2, r0         @(7): imm32 = addrTarget - (addrCFI + 4)
15   mov r3, #0             @(8): Clean register to store valid CFI.
16   mov r4, r2, lsr #1     @(9): (9)-(11) Take imm11 from imm32.
17   mov r0, #0x7FF         @(10)
18   and r4, r4, r0         @(11)
19   orr r3, r3, r4         @(12): Load imm11.
20   mov r4, r2, lsr #12    @(13): (13)-(15) Take imm10 from imm32.
21   mov r0, #0x3FF         @(14)
22   and r4, r4, r0         @(15)
23   mov r4, r4, lsl #16    @(16): (16)-(17) Place and load imm10.
24   orr r3, r3, r4         @(17)
25   mov r0, r2, lsr #24    @(18): (18)-(19) Take S from imm32.
26   and r0, r0, #0x1       @(19)
27   mov r4, r0, lsl #26    @(20): (20)-(21) Place and load S.
28   orr r3, r3, r4         @(21)
29   mov r4, r2, lsr #22    @(22): (22)-(23) Take I2 from imm32.
30   and r4, r4, #0x1       @(23)
31   eor r4, r4, r0         @(24): (24)-(26) J2 = not (I2 xor S)
32   mvn r4, r4             @(25)
33   and r4, r4, #0x1       @(26)
34   mov r4, r4, lsl #11    @(27): (27)-(28) Place and load J2
35   orr r3, r3, r4         @(28)
36   mov r4, r2, lsr #23    @(29): (29)-(30) Take I1 from imm32.
37   and r4, r4, #0x1       @(30)
38   eor r4, r4, r0         @(31): (31)-(33) J1 = not (I1 xor S)
39   mvn r4, r4             @(32)
40   and r4, r4, #0x1       @(33)
41   mov r4, r4, lsl #13    @(34): (34)-(35) Place and load J1
42   orr r3, r3, r4         @(35)
43   ldr r4, =#0xF0009000 @(36): (36)-(37) Load CFI's nemonic opcode.
44   orr r3, r3, r4         @(37)
45 @Step 3: Compare Step 1 and Step 2.
```

```
46   sub r3, r3, r1        @(38): Should store zero for correct execution.
47   cmp r3, #0            @(39): (39)-(43) rx = !rx
48   ite eq                @(40)
49   moveq r3, #1          @(41)
50   movne r3, #0          @(42)
51   uxtb  r3, r3          @(43)
52   sdiv  r3, r1, r3      @(44): Raises exception if rx was not zero.
53 @Step B: Restore context.
54   pop {r0-r4}           @(B1): Restore used registers.
55 @End PECOS Block.
56 .CFI1:   b.w .L3
```

CÓDIGO B.1. PAB completo.

Bibliografía

[1] Hillmann, S., & Wachter, M. (2022, January 27). New Space is becoming increasingly important for German industry. *The Federation of German Industries (BDI).*

[2] Asciolla D., Dilillo L., Santos D., Melo D., Menicucci A., Ottavi M. (2020). Characterization of a RISC-V Microcontroller Through Fault Injection. In: Applications in Electronics Pervading Industry, Environment and Society (pp.91-101). DOI: 10.1007/978-3-030-37277-4_11.

[3] Bagchi S., Kalbarczyk Z., Iyer R., Levendel Y., "Design and Evaluation of Preemptive Control Signature (PECOS) Checking," in *IEEE Transactions on Computers*, 2003.

[4] S. S. Yau and Fu-Chung Chen, "An Approach to Concurrent Control Flow Checking," in *IEEE Transactions on Software Engineering*, vol. SE-6, no. 2, pp. 126-137, March 1980, doi: 10.1109/TSE.1980.234478.

[5] 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, pp. 240-249, doi: 10.1109/FTCS.1998.689475.

[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," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 627-641, June 1999, doi: 10.1109/71.774911.

[7] N. J. Wang and S. J. Patel, "ReStore: Symptom-Based Soft Error Detection in Microprocessors," in *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 188-201, July-Sept. 2006, doi: 10.1109/TDSC.2006.40.

[8] Andjelković, Marko & Petrovic, Vladimir & Stamenkovic, Zoran & Ristic, Goran & Jovanović, Goran. (2015). Circuit-Level Simulation of the Single Event Transients in an On-Chip Single Event Latchup Protection Switch. Journal of Electronic Testing. 31. 275-289. 10.1007/s10836-015-5529-1.

[9] R. W. Hamming, "Error detecting and error correcting codes," in *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147-160, April 1950, doi: 10.1002/j.1538-7305.1950.tb00463.x.

[10] 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. DSN 2000*, 2000, pp. 417-426, doi: 10.1109/ICDSN.2000.857571.

[11] SAM V71 Xplained Ultra.

[12] ISO/IEC 10646:2017 Information technology — Universal Coded Character Set (UCS).

[13] ISO/IEC 6429:1992 Information technology — Control functions for coded character sets.

[14] ARM Architecture Reference Manual Thumb-2 Supplement.

[15] Arm Cortex-M7.

[16] ARMv7-M Architecture Reference Manual.

[17] MPLAB® X Integrated Development Environment (IDE).

[18] Arm GNU Toolchain.

[19] The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors - Third Edition - Joseph Yiu.

[20] How to debug a HardFault on an ARM Cortex-M MCU. Chris Coleman.

[21] ARM Cortex-M4(F) Lazy Stacking and Context Switching.

[22] Object-Oriented Programming With ANSI-C.

[23] GCC: Extended ASM.

[24] Testing Embedded Software. Bart Croakman & Edkin Notenboom. 2003.

[25] ARM Keil: Cortex-M3/M4/M7 Fault Exceptions.