Implementing an ISR defense on a MIPS architecture

Loriana Sanabria Sancho
Universidad de Costa Rica
San José Costa Rica
loriana.sanabria@ucr.ac.cr

Elena Gabriela Barrantes
Universidad de Costa Rica
San José Costa Rica
gabriela.barrantes@ecci.ucr.ac.cr

***Abstract*—Code injection attacks are an undeniable threat in today's cyberworld. Instruction Set Randomization (ISR) was initially proposed in 2003. This technique was designed to protect systems against code injection attacks by creating an unique instruction set for each machine, thanks to randomization. It is a promising technique in the growing embedded system and Internet of Things (IoT) devices ecosystem, where the lack of complex memory management make these devices more vulnerable. However, most of ISR implementations up to day are entirely software based. In this work, we implement hardware support for an ISR defense on an 32 bits, 5 pipeline stages MIPS processor (which is an embedded system compatible architecture). Two obfuscation schemes were implemented, one based on XOR encryption and the other on transposition. The hardware implementation was tested under synthetic code injection attacks and results shows the effectiveness of the defense using both encryption circuits.**

***Index Terms*—ISR, MIPS processor, encryption circuits, code injection attacks, hardware**

I. Introducción

La protección de sistemas y recursos cibernéticos es una batalla continua, donde el atacante, al parecer, siempre lleva ventaja. Las defensas de blanco móvil (MTD, por sus iniciales en inglés), o defensas de diversificación, intentan reducir esta ventaja alterando la naturaleza estática de los sistemas computacionales [1]. Por otra parte, la estandarización de dichos sistemas es también necesaria y beneficiosa. Es por ello que la introducción de componentes aleatorizados debe realizarse de forma cuidadosa y perjudicando lo menos posible la interacción entre sistemas [2].

Se han propuesto múltiples diversificaciones para diferentes componentes de los sistemas computacionales que responden a diferentes modelos de amenaza [3]. El estudio presentado en este artículo se refiere a un tipo de MTD específico: la aleatorización del conjunto de instrucciones de máquina (ISR, por sus iniciales en inglés), planteada originalmente de forma simultánea en 2003 por Barrantes et al en [4] y Gaurav et al en [5]. La ISR, en su planteamiento más abstracto, propone que cada computador tenga un conjunto propio y diferente de instrucciones de máquina. Esta diversificación perturba la ejecución de un bloque de instrucciones proveniente de otra máquina, ya que el procesador local va a interpretar los códigos de instrucción de forma diferente, o no los va a "entender" del todo, y va a generar una excepción.

Para poder evaluar el valor de seguridad de una aleatorización es necesario referirse a el o los modelos de amenaza que serían afectados. El modelo de amenaza principal al que responde ISR es la inyección de código de máquina externo en el flujo de ejecución de un proceso en una máquina local [6]. Dicha inyección puede ocurrir por distintos medios y situarse en diferentes partes del espacio de memoria de un proceso. El medio más clásico de inyección es el *Stack Overflow* [7], pero existen muchos otros [6]. La inyección de código se ha conocido como amenaza desde que se crearon los primeros sistemas operativos. A pesar de ello, no ha podido ser definitivamente resuelto con métodos convencionales, ni en sus vectores de entrada, ni en sus métodos de ejecución (véase, por ejemplo [8]). ISR, por lo tanto, continúa siendo una opción valiosa para una amenaza real.

A pesar de que ISR se ha propuesto de forma práctica desde 2003, la mayoría de las implementaciones publicadas corresponden a pruebas de concepto sobre máquinas virtuales de uno u otro tipo, lo que resulta en rendimientos muy pobres [3]. Existen, sin embargo, unas pocas implementaciones con prototipo en hardware [3], [9]. En particular, es interesante centrarse en el entorno de procesadores empotrados, cuyo uso es cada vez más ubicuo con la penetración de Internet de las Cosas (IoT, por sus iniciales en inglés). El caso que nos interesa es el procesador MIPS de 32 bits [10], el cual es uno de los utilizados para dispositivos empotrados [11], [12], [13]. Se seleccionó esta arquitectura al ser apropiada para ecosistemas de sistemas embebidos, donde debido a sus limitaciones no se cuenta con sistemas de manejo de memoria complejos, haciéndolos aún más vulnerables a ataques de inyección de código binario[14], [15].

En este trabajo se implementa defensa ISR para un procesador MIPS de 32 bits, se realiza una simulación de hardware a nivel de RTL, se evalúa su funcionalidad y su efectividad como defensa ante segmentos sintéticos de código no aleatorizados, a modo de ataque. Asimismo, se exploran dos mecanismos de aleatorización.

El artículo se organiza de la siguiente manera: en la

978-1-5386-3057-0/17/$31.00

sección II se describe la defensa a implementar, la sección III delimita el ataque. Posteriormente en la sección IV se describe el experimento. En la sección V se muestran y discuten los resultados obtenidos. Por último, en la secciones VI y VII se exploran trabajos relacionados y se brindan las conclusiones junto con recomendaciones.

II. La defensa

Una implementación real de ISR debe traducir el modelo abstracto de "diversificar" cada conjunto de instrucciones. Todas las implementaciones publicadas tienden a "aleatorizar" de forma local el conjunto, en otras palabras, usan técnicas prestadas de cifrado para - a nivel del ciclo *fetch*- interponer un filtro entre el procesador real (que muy probablemente no se puede cambiar en cada máquina) y el código que se ejecuta [3]. La solución implementada en este trabajo utiliza la misma táctica. Es por esto que se habla de "cifrar" y de "llave de cifrado". Sin embargo, debe quedar claro que el propósito de ISR no es ocultar el código ni mantener un secreto, sino contar con una versión, lo más dinámica posible, de un procesador "propio".

Como se mencionó anteriormente, a nivel práctico las técnicas de ISR cifran las instrucciones de los procesos con una llave específica para prevenir ataques de inyección de código. Esta llave define el conjunto de instrucciones válidas para el proceso y es utilizada por el procesador para decodificar cada instrucción. Cualquier código malicioso que haya sido inyectado por el atacante no estará cifrado con la llave correcta o no estará cifrado del todo (debido a que la llave no es conocida), convirtiendo el código inyectado por el atacante en instrucciones inválidas que no serán ejecutadas.

A. Preparación de la defensa

La primera etapa de la defensa es la aleatorización (o cifrado) del código del proceso a ejecutar y la segunda etapa consiste en revertir esta aleatorización antes de que código sea ejecutado por el procesador. A nivel de este artículo nos enfocamos en la segunda etapa y solo reportamos la ejecución de código previamente preparado.

Sin embargo, en nuestro modelo, cada proceso a ser ejecutado debe ser ofuscado justo antes de ser cargado a memoria. La aleatorización del código se lleva a cabo por medio de una llave exclusiva (generada de forma aleatoria) para cada proceso. Es decir, cada proceso va a contar con su propia llave y además, en cada ejecución del proceso la llave va a ser distinta. Cuando el proceso es cargado a memoria se indica al procesador cual fue la llave utilizada para ofuscar el código, de esta forma durante la ejecución se logra revertir la aleatorización realizada permitiendo que las instrucciones sean ejecutadas de forma normal. Este esquema hace que el modelo de amenaza sea más efectivo contra atacantes remotos, dado que los procesos locales cuentan con mucho más acceso a la memoria para ataques de obtención de información sobre llaves.

Gracias a la propiedad de instrucciones de longitud constante de MIPS, las llaves con las que se ofusca el proceso son de longitud constante también, y el proceso de aleatorización del código consiste entonces en enmascarar cada instrucción con la llave generada de forma aleatoria.

En este trabajo se exploran dos opciones para la aleatorización del código. Como primera opción se utilizó un mecanismo de sustitución basado en la operación XOR. En este caso, cada bit de las instrucciones del proceso fue sustituido por el resultado de la operación XOR bit a bit, entre la instrucción y la llave correspondiente. Se utilizó una llave de 32 bits (misma longitud de las instrucciones del procesador).

Como segunda opción, se utilizó un mecanismo de transposición para ofuscar el proceso. En este caso, se utilizaron llaves de 160 bits para reordenar los 32 bits de cada instrucción. Los 160 bits se dividen en grupos de cinco bits. Cada grupo de bits representa un número del 0 al 31, de forma que la llave indica la secuencia en fueron intercambiados los bits de las instrucciones del proceso.

El proceso de carga a memoria debe, por lo tanto "cifrar" el código de acuerdo al esquema de aleatorización utilizado. De nuevo, la implementación reportada en este artículo no trata con el aspecto de software de carga a memoria, sino que suponemos que el cifrado se realizó de previo antes de ejecutar el proceso en la máquina local.

B. Implementación de la defensa en el procesador MIPS

La implementación propuesta es estrictamente en hardware: se asume que el sistema operativo que corre sobre esta plataforma tiene la capacidad de ofuscar el código, cargarlo en memoria y comunicar al procesador la llave utilizada para ofuscar del proceso. La implementación consiste, entonces, en incluir al procesador MIPS la lógica necesaria para revertir la aleatorización, para proceder luego con la ejecución normal de las instrucciones. Por medio del ambiente de simulación se incluyó la capacidad de tomar el proceso, generar la llave, ofuscar el código, e inyectarlo a la memoria del procesador.

Como punto de partida se contó con un procesador MIPS, con funcionalidades básicas implementado en Verilog [16]. A este procesador se le agregó un registro para almacenar la llave con el que el conjunto de instrucciones fue aleatorizado, así como un circuito para revertir el cifrado de instrucciones.

La Figura 1 muestra un esquema de alto nivel de la implementación. Este esquema indica la posición relativa de los circuitos para revertir la aleatorización con respecto a las etapas de ejecución de instrucciones del procesador.

Como se mencionó anteriormente, se implementaron dos circuitos asociados a distintos mecanismos de aleatorización, ambos circuitos se agregaron dentro del módulo de la memoria de instrucciones. De esta forma, cuando el procesador realiza la lectura de la instrucción, esta ya no está cifrada y se ejecuta de forma correcta.

El primer circuito se basó en "compuertas O exclusivas" (XOR). La llave utilizada es del mismo tamaño de las instrucciones del procesador (32 bits). Este circuito realiza una operación bit a bit entre la instrucción cifrada y la llave, para generar la instrucción correcta.

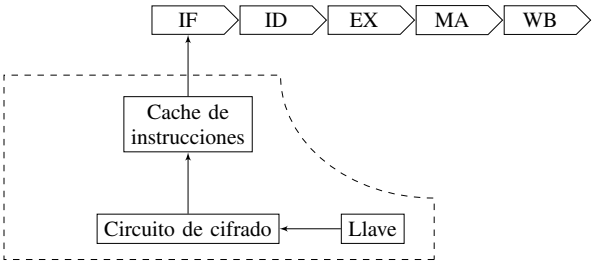

Fig. 1: Esquema de la defensa ISR. El circuito de cifrado se implementa antes del *pipeline*.

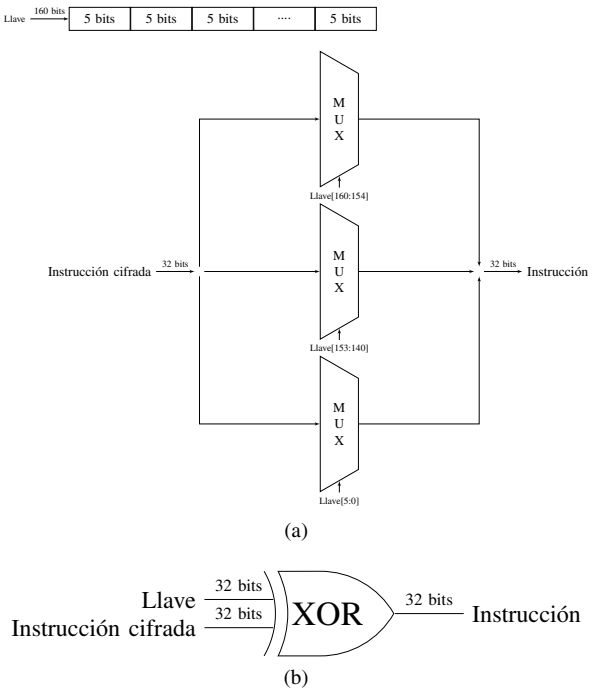

Fig. 2: Lógica para revertir la aleatorización: a) Lógica para la implementación por transposición, b) Lógica para implementación por sustitución utilizando XOR.

El segundo es un circuito es el encargado de revertir la aleatorización por transposición, en este mecanismo los elementos se reordenan de acuerdo a un patrón en particular, dado por la llave. En este caso se utiliza una llave de 160 bits como patrón para transponer los elementos originales de la instrucción. La implementación del circuito requiere 32 MUX. Cada MUX debe tener 32 entradas (longitud de la instrucción), cinco bits de selección y un solo bit de salida. La instrucción original se reconstruye uniendo la salida de los 32 MUX.

La Figura 2 muestra el diagrama de los circuitos implementados.

III. El Ataque

En un ataque de inyección de código, el atacante explota una vulnerabilidad del *software*, por ejemplo, una vulnerabilidad de *buffer overflow*, para inyectar código malicioso en el programa en ejecución. El atacante puede ejecutar cualquier código arbitrario en la máquina afectada. Este tipo de ataque

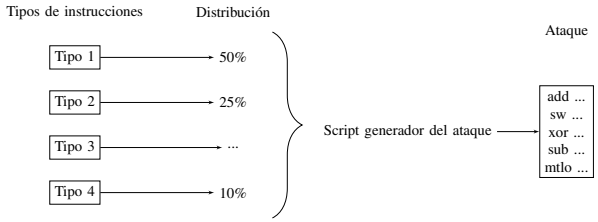

Fig. 3: Método de construcción del ataque.

es muy peligroso, pues el atacante cuenta con todos los privilegios del proceso comprometido [17].

La defensa se prueba contra un ataque sintético de inyección de código binario. Se asume que el atacante toma control del flujo del programa y lo redirecciona a la sección de memoria donde el código malicioso fue inyectado.

Como se explicó anteriormente, para esta investigación se simuló el procesador con la nueva lógica para revertir la aleatorización. El sistema no incluía el manejo la memoria virtual. Por esta razón el ataque consistió en una secuencia de instrucciones en binario fue inyectada a la memoria del procesador por medio del ambiente de simulación.

Al crear el ataque, se asumió que el atacante desconocía la presencia de la defensa, por lo que se inyecta una secuencia de instrucciones válidas para la arquitectura MIPS.

La secuencia de instrucciones del ataque se crea arbitrariamente: no nos interesa el contenido de lo que el hipotético atacante deseaba ejecutar. Naturalmente, durante el "descifrado" las instrucciones originales se convierten en ruido. Desafortunadamente parte del ruido puede corresponder a instrucciones reales, aunque ciertamente no las mismas que escribió el atacante. Cuando el proceso es legítimo, la secuencia de instrucciones de entrada se encuentra aleatorizada y al pasar por la nueva lógica, agregada al procesador, se obtienen instrucciones MIPS que el procesador puede ejecutar.

Para crear el ataque, las instrucciones del procesador se dividieron en cuatro tipos: aritméticas y lógicas; control de flujo; movimiento de datos y manejo de constantes y de acceso a memoria.

La defensa se probó con cinco mezclas de instrucciones. Los primeros cuatro ataques contenían instrucciones de un solo tipo y el quinto contaba con una distribución equitativa de todos los tipos de instrucciones.

Se construyeron secuencias de instrucciones de 20 líneas. Este es un valor arbitrario, elegido con base en resultados de experimentos preliminares, en los que se determinó que la probabilidad de ejecutar 20 instrucciones válidas era baja.

El proceso de construcción del ataque se muestra en la Figura 3. La elección de las instrucciones que forman parte del ataque se hace de forma aleatoria, por medio de un *script* que cuenta con todas las instrucciones y su clasificación. Como entrada para la construcción del ataque se recibe la distribución deseada según el tipo de instrucción.

El "ataque exitoso" se define con respecto a si el código inyectado llega a ejecutar toda la longitud del bloque (20

TABLA I: Opcode y función en una instrucción del procesador MIPS.

	Opcode							Función					
Número de bit	31	30	29	28	27	26	...	5	4	3	2	1	0
Instrucción	0	0	0	0	0	0		0	0	0	1	0	1

instrucciones) sin generar una excepción antes. Se reporta esto como ataque "exitoso" en el sentido limitado de que luego de ejecutar múltiples instrucciones aleatorias el atacante puede ser que volviera a estar en el flujo original de ejecución, y el sistema atacado no detecte el intento.

Para determinar el "éxito" del ataque, durante la simulación, el *opcode* y la función (en el caso de las instrucciones aritméticas) fueron verificadas en cada ciclo de reloj. Recordemos que el *opcode*, es el código asignado a cada operación del procesador. La Tabla I muestra como se divide una instrucción en el procesador MIPS. En el caso de este procesador, el *opcode* está dado por los 6 bits más significativos de la instrucción, es decir los bits del 31 al 26. Para las instrucciones aritméticas, el *opcode* es el mismo (000000), pero se cuenta con un valor de función que indica el tipo de operación. El valor de función esta dado por los seis bits menos significativos.

IV. Descripción del experimento

Dados los dos mecanismos de cifrado propuestos, la probabilidad de un atacante (que conoce de la presencia de la defensa), de lograr inyectar código coherente al procesador es distinta para cada uno de los métodos de aleatorización considerados. En el caso del primer mecanismo (XOR), el atacante tiene un probabilidad de 2^{-32} de acertar la llave correcta, mientras que con el segundo mecanismo (trasposición) la probabilidad es aún menor, de $1/32!$.

Se espera que al aleatorizar el conjunto de instrucciones el código del atacante se transforme en ruido: instrucciones diferentes de las que espera el atacante, posiblemente algunas instrucciones inválidas o acceso a direcciones de memoria inexistentes que pueden generar excepciones y terminar el proceso.

En el experimento propuesto se contabilizaron la cantidad de instrucciones válidas ejecutadas antes de la ocurrencia de una instrucción inválida que terminaría el proceso (para efectos de este trabajo se consideró que la instrucción era inválida si el *opcode* lo era). La probabilidad de fallo es aún mayor en un proceso real que sí acceda memoria, como se ve en el análisis realizado en [2].

Para esta investigación se consideraron dos factores: el tipo de aleatorización y el tipo de ataque. Una implementación de ISR con un buen método aleatorización logra un mayor nivel de entropía, dificultando tanto la ejecución de código malicioso válido, por parte del atacante, como el acertar la llave o el mecanismo de cifrado. Se comparó la efectividad y el costo de implementación en hardware de los dos sistemas de aleatorización bajo el mismo ataque. Como se explicó en la sección de la defensa se utilizaron dos mecanismos de aleatorización: el primero basado en XOR de 32 bits y el segundo basado transposición con una llave de 160 bits, dos técnicas con distintos niveles de entropía y complejidad en hardware.

TABLA II: Distribución de instrucciones ejecutadas para cada tipo de ataque.

Tipo1	Tipo2	Tipo3	Tipo4
100%	0%	0%	0%
0%	100%	0%	0%
0%	0%	100%	0%
0%	0%	0%	100%
25%	25%	25%	25%

El segundo factor de diseño fue la distribución porcentual del tipo de instrucciones en el ataque, esto fue representado por una secuencia de instrucciones en binario, construida a partir de instrucciones válidas. Para representar distintos tipos de ataque, se construyeron distintas secuencias de instrucciones. Cada secuencia de instrucciones contó con una distribución porcentual distinta en cuanto al tipo de instrucciones. Se consideraron cinco ataques. Para esto las instrucciones se dividieron en cuatro tipos: instrucciones aritméticas y lógicas (tipo1); instrucciones de control del flujo (tipo2); instrucciones de movimiento de datos y manejo de constantes (tipo3) e instrucciones de acceso a memoria (tipo4).

La intención de este experimento era validar que la defensa fuera efectiva sin importar la composición de instrucciones del ataque. Con este propósito se generan cinco "ataques" (mezclas de código sintético). La distribución según el tipo de instrucción se muestra en Tabla II.

Se realizaron cien repeticiones para cada combinación de factores de diseño (tipo de aletorización y tipo de ataque) y para cada experimento se contabilizó el número instrucciones del ataque que se ejecutan de forma exitosa antes de ejecutarse una instrucción inválida.

La Figura 4 describe el contexto del experimento, la interacción de la defensa y el ataque, los factores fijos y de ruido.

Para simular la defensa se construyó un banco de pruebas para el procesador MIPS modificado. El banco de pruebas se diseñó para que tomara como entradas la secuencia de instrucciones que representa el ataque y el mecanismo de aleatorización a utilizar. Además se configuró el ambiente de simulación para generar un reporte de cada experimento. Posteriormente los resultados de la simulación fueron procesados para determinar si se trató de ejecutar alguna instrucción inválida (posible ataque).

V. Resultados y discusión

La defensa es inmune al ataque en el sentido en que en todos los casos el código que llegó al procesador no se parecía al código original que quería ejecutar el atacante.

Sin embargo, sorprendentemente, sí hubo ataques "exitosos": secuencias que luego de ser aleatorizadas eran capaces de ejecutarse completamente. Sin embargo, la Tabla III muestra que las posibilidades son bajas: varían entre un 2.00% a un 8.50% para ataques formados únicamente por instrucciones de movimiento y manejo de constantes. A pesar de no contar con un explicación que respalde esta diferencia, queda clara la

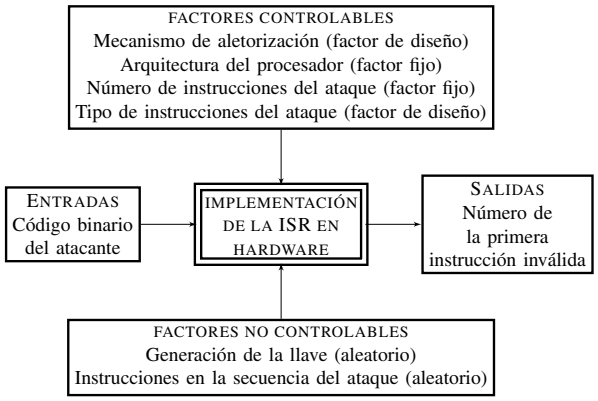

Fig. 4: Contexto de interacción entre la defensa ISR y el ataque de inyección de código.

TABLA III: Porcentaje de ataques exitosos según el tipo de instrucciones.

Tipo de Instrucción	Porcentaje de ataques exitosos
Mixta	2.00%
Aritmética	4.50%
Control de flujo	7.50%
Acceso a Memoria	8.00%
Movimiento de datos	8.50%

efectividad de la defensa. Para todos los casos, la probabilidad de éxito del ataque está por debajo del 10.00%. La Tabla IV y la Figura 5 conducen a la misma conclusión. Sin embargo, es importante resaltar que la mediana de los resultados obtenidos para todos los tipos de ataques es de dos instrucciones en el peor de los casos. Por su parte para el percentil 75% se tiene que en el peor de los casos se logran ejecutar solo cuatro instrucciones. Esto demuestra que el rango de acción del atacante es limitado, pues en pocas instrucciones (menos de cuatro) debe ser capaz de lograr su objetivo.

Al variar el mecanismo de aleatorización, se obtienen resultados más interesantes. La Figura 6 y Tabla V muestran que el mecanismo utilizado para ofuscar el proceso juega un papel fundamental en la defensa. Para el caso de la aleatorización por medio sustitución XOR, el porcentaje de ataques exitosos fue del 11.80%, mientras que para el caso del mecanismo de aleatorización por transposición fue del 0.4%. Esto era de esperar debido a la entropía de cada uno de los métodos de aleatorización, discutida en la sección II. Sin embargo, los valores de distribución de la primera instrucción inválida mostrados en la Tabla VI (según el tipo de ataque) son similares a los obtenidos en la Tabla IV (al variar el método de aleatorización), de forma que sin importar el ataque o el

TABLA IV: Valores de distribución de la primera instrucción inválida para el tipo de ataque.

Nivel	Mínimo	25%	Mediana	75%	Máximo
Instrucciones mixtas	1	1	1	2	21
Instrucciones aritméticas	1	1	1	3	21
Instrucciones de control de flujo	1	2	2	4	22
Instrucciones de acceso a memoria	1	1	1	2	21
Instrucciones de movimiento de datos	1	1	1	2	24

TABLA V: Porcentaje de ataques exitosos según el tipo de instrucción.

	Mecanismo de aleatorización XOR	Mecanismo de aleatorización por transposición
Ataques exitosos	11.80%	0.40%

TABLA VI: Valores de distribución de la primera instrucción inválida según el tipo de aleatorización.

Nivel	Mínimo	25%	Mediana	75%	Máximo
Aleatorización XOR	1	1	1	2	20
Aleatorización por transposición	1	1	1	5	24

mecanismo de ofuscación el atacante cuenta con una ventana de oportunidad pequeña.

Es importante recalcar que, para efectos de este trabajo, se considera que se ejecutó una instrucción de forma exitosa si su *opcode* es válido, es decir, no se toma en cuenta que la instrucción ejecutada fuera o no la que el atacante deseaba ejecutar y tampoco se consideran accesos inválidos a memoria, que también resultarían en una excepción del sistema.

Puesto que el porcentaje de ataques exitosos es de solo 0.4% para la aleatorización por transposición, se puede concluir que para este mecanismo el tipo de instrucciones presentes en el bloque de ataque no tiene relevancia. Las diferencias observadas en los resultados están ligadas únicamente a la aleatorización basada en XOR, que cuenta con menor entropía. La Tabla VII y la Figura 7 muestran que cuando se utilizó la aleatorización XOR el porcentaje de éxito para los distintos ataques varió de un 3.00%, en el mejor de los casos, a un 17%. Cuando se utilizó la aleatorización por transposición, sin importar el tipo de ataque, el porcentaje de éxito del ataque fue menor al 1.00%, en el peor de los casos. Esto respalda la hipótesis de que, cuando se usa un buen mecanismo de aleatorización, las instrucciones usadas en el ataque no tienen mayor efecto sobre la defensa ISR.

Es claro que la ofuscación por transposición brinda mejores resultados en términos de la defensa, pero comparando los dos tipos de aleatorización desde una implementación de

TABLA VII: Porcentaje de ataques exitosos para los distintos tipos de ataque y mecanismos de aleatorización.

Tipo de instrucción	Ataques exitosos aleatorización XOR	Ataques exitosos aleatorización transposición
Mixta	3.00%	1.00%
Aritmética	8.00%	1.00%
Control de flujo	15.00%	0.00%
Acceso a memoria	16.00%	0.00%
Movimiento de datos	17.00%	0.00%

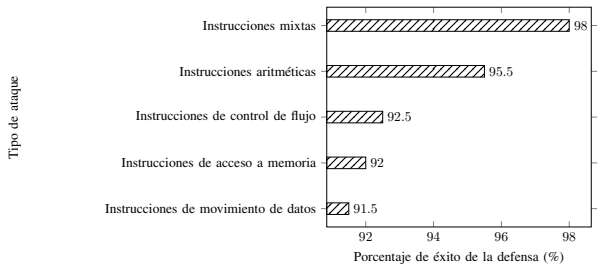

Fig. 5: Porcentaje de éxito de la defensa al variar el tipo de instrucción.

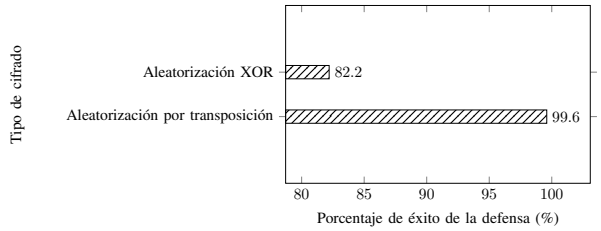

Fig. 6: Porcentaje de éxito de la defensa al variar el tipo de aleatorización.

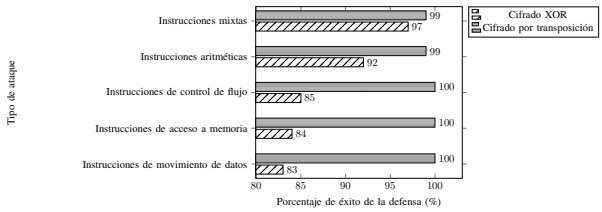

Fig. 7: Porcentaje de éxito de la defensa al variar el tipo de instrucción para cada tipo de cifrado.

hardware el mecanismo por transposición es más costoso. Para implementar la aletorización XOR se utilizan únicamente 32 compuertas XOR de dos entradas, que si se construyeran usando solo compuertas básicas la inversión en hardware requeriría 32 inversores, 62 compuertas AND y 32 compuertas OR.

Por otra parte, para construcción de la lógica encargada de revertir la aleatorización por transposición se requieren 32 MUX de 32 bits cada uno, considerando solo compuertas básicas para su implementación se necesitarían 672 inversores, 1042 compuertas AND de 3 entradas y 352 compuertas OR de 4 entradas.

Resumiendo los detalles de implementación para ambos tipos de cifrado, se necesitarían 126 compuertas en total para la implementación de la lógica XOR y de 2048 para la lógica por trasposición.

VI. Trabajo relacionado

Existe una gran cantidad de trabajos relacionados con la ISR, donde es empleada como una defensa contra ataques de inyección de código [5], [4]; sin embargo, la mayoría están enfocados a solo técnicas en el contexto de software y requieren de emulación.

El conocimiento en técnicas para la implementación de circuitos de cifrado en de hardware es también amplio [18]. Sin embargo, son limitados los estudios de sistemas de cifrado que provean soporte a la ISR. Dos investigaciones son muy similares a la propuesta: [19] y [9].

La primera corresponde a una defensa por diversificación en un procesador MIPS, donde se explora un único mecanismo de ofuscación que consiste en cifrado por transposición y tablas de mapeo para remplazar el *opcode* de las instrucciones; no obstante, esta investigación se enfoca en la detección de ataques de inyección de código al contar con dos ejecuciones paralelas del proceso: una ejecución con el código regular y otra con el código aleatorizado.

La segunda investigación es la más afín a la nuestra. En este trabajo, propuesto por Papadogiannaki [9], se planteó un soporte de arquitectura para la defensa ISR. La arquitectura propuesta en [9] es resistente a tanto a ataques de inyección de código como a ataques tipo ROP. En dicho trabajo se implementó la solución en procesador *LeonSPARC v*8 y así mismo realizaron la modificaciones en kernel de LINUX 3.8 para soportar el nuevo hardware. Se exploraron, también, dos opciones para la localización del circuito de cifrado (antes y después de la etapa de *fetch*) y se probaron dos tipos de cifrado: uno basado en XOR y el otro en transposición usando distintos tamaños de llave.

La mayor diferencia con [9] se encuentra en la elección de la arquitectura y el análisis realizado. MIPS es un procesador de propósito general, que no solo es compatible con el ecosistema de sistemas embebidos e IoT [12]. Además MIPS cuenta con un mercado en *networking*, televisores inteligentes y casas conectadas donde asegura tener entre un 40% y 50% del mercado del global [20].

VII. Conclusiones y trabajo futuro

Se logró verificar la eficacia de la defensa de tipo ISR contra ataques de inyección de código. La implementación en hardware de la defensa ISR es una solución viable que debe ser tomada en cuenta para minimizar el impacto en el rendimiento provocado por implementaciones basadas únicamente en software. Esto es extensible a otras defensas por diversificación que se puedan ver beneficiadas por soporte en el procesador.

Por otra parte se determinó que el mecanismo de aleatorización es, efectivamente, el corazón de la defensa ISR y que un mecanismo con mayor entropía brinda mejores resultados. Sin embargo, es necesario comprender que la implementación de la lógica para revertir la aleatorización puede ser costosa en cuanto a hardware.

Es necesario estudiar más a fondo el costo-beneficio de implementar un mecanismo basado XOR, que es más simple en términos de hardware pero más susceptible a ataques de *key guessing*, o alternativamente implementar una aleatorización basada transposición, que es más costosa en hardware pero brinda más entropía al sistema. Así como explorar mecanismos para ofuscar el código aún más complejos, que provean mayor seguridad.

Futuros trabajos se pueden enfocar en este análisis costo beneficio en términos de hardware y seguridad de la llave. Como parte de este análisis, se pueden también implementar, en arreglos de compuertas programables (FPGA), distintas soluciones de la defensa para un sistema embebido basado en MIPS.

Agradecimientos

Agradecemos el apoyo parcial del Posgrado en Computación e Informática (PCI), la Escuela de Ciencias de la

Computación e Informática (ECCI) y el Centro de Investigaciones en Tecnologías de la Información y Comunicación (CITIC).

REFERENCIAS

[1] S. Forrest, A. Somayaji, y D. Ackley, "Building Diverse Computer Systems," in *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, 1997, pp. 67–72.

[2] E. G. Barrantes, D. H. Ackley, S. Forrest, y D. Stefanović, "Randomized instruction set emulation," *ACM Transactions on Information and System Security (TISSEC)*, vol. 8, no. 1, pp. 3–40, 2005.

[3] H. Okhravi, T. Hobson, D. Bigelow, y W. Streilein,, "Finding focus in the blur of moving-target techniques," *IEEE Security & Privacy*, vol. 12, no. 2, pp. 16–26, 2014.

[4] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, y D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 281–289.

[5] G. S. Kc, A. D. Keromytis, y V. Prevelakis, "Countering Code-Injection Attacks With Instruction-Set Randomization," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*. Washington, D.C., U.S.A.: ACM Press, October 27-31 2003, pp. 272–280.

[6] J. Wilander y M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," in *Proceedings of the 10th Network and Distributed System Security Symposium*, San Diego, California, February 2003, pp. 149–162.

[7] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 49, no. 7, Nov. 1996.

[8] M. Mimoso, "Unpatched remote code execution flaw exists in swagger," 2016, [En línea]. Disponible en: https://threatpost.com/unpatched-remote-code-execution-flaw-exists-in-swagger/118867.

[9] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou y S. Ioannidis, "Asist: architectural support for instruction set randomization," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 981–992.

[10] Imagination Technologies, "Architecture for programmers volume i-a: Introduction to the mips32 architecture," 2014, [En línea]. Disponible en: https://www.imgtec.com/?do-download=introduction-to-the-mips32-architecture-v6-01.

[11] Imagination Technologies Limited, "Mips processors," 2017, [En línea]. Disponible en: https://www.imgtec.com/mips/.

[12] NetWorkWorld, "Defections to arm hurt powerpc, mips," 2015, [En línea]. Disponible en: http://www.networkworld.com/article/2880774/servers/defections-to-arm-hurt-powerpc-mips.html.

[13] The Linux Fundation, "Mips takes on arm in the internet of things," 2014, [En línea]. Disponible en: https://www.linux.com/news/mips-takes-arm-internet-things.

[14] A. D. Keromytis, "Randomized instruction sets and runtime environments past research and future directions," *IEEE Security & Privacy*, vol. 7, no. 1, pp. 18–25, 2009.

[15] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, y J. Rowanhill, "Secure and practical defense against code-injection attacks using software dynamic translation," in *Proceedings of the 2nd international conference on Virtual execution environments*. ACM, 2006, pp. 2–12.

[16] J.Mahler, "Mips cpu implemented in verilog," 2015, [En línea]. Disponible en: https://github.com/jmahler/mips-cpu.

[17] A. N. Sovarel, D. Evans y N. Paul, "Where's the feeb? the effectiveness of instruction set randomization." in *Usenix Security*, 2005.

[18] F. Rodríguez-Henríquez, N. A. Saqib, A. D. Perez, y C. K. Koc, *Cryptographic algorithms on reconfigurable hardware*. Springer Science & Business Media, 2007.

[19] Z. Liu, W. Shi, S. Xu, y Z. Lin, "Programmable decoder and shadow threads: Tolerate remote code injection exploits with diversified redundancy," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE, 2014, pp. 1–6.

[20] Electronicsweekly, "Restructured imagination focuses on powervr, mips and wireless ip," 2017, [En línea]. Disponible en: https://www.electronicsweekly.com/news/business/profile-restructured-imagination-focusses-powervr-mips-wireless-ip-2017-02/.