Seguridad en redes y aplicaciones distribuidas
Análisis de la detección de emulación basada en tiempos

José Miguel Fernández Sánchez
Master en seguridad de las TIC

Seguridad en redes y aplicaciones distribuidas
Análisis de la detección de emulación basada en tiempos

Profesor colaborador: Ángel Elbaz Sanz
Profesor responsable: Guille Navarro Arribas

RESUMEN

El objetivo de este trabajo de fin de master es analizar las técnicas de detección de emulación basada en el análisis de tiempos de ejecución que podrían ser usadas por el *malware*, buscar una solución general para evitar las detecciones mencionadas, e implementar una prueba de concepto que permita evaluar la viabilidad de la solución propuesta.

Se han desarrollado diversos ejemplos de detección, concluyendo que una solución sería que el emulador incorporase a la emulación aspectos relativos a la implementación de la arquitectura. Se ha desarrollado una prueba de concepto basada en la solución propuesta, y se ha comprobado que dicha prueba de concepto es capaz de evitar la detección de todos los ejemplos presentados.

ABSTRACT

The goal of this Master`s Thesis is to analyze time based emulation detection techniques who could be used by malware, propose a general solution to avoid those detection techniques and implement a proof of concept to assess the viability of the proposed solution.

Several detection examples have been developed, inferring that a plausible solution could be to incorporate arquitectural implementation details to the emulation. A proof of concept has been developed based on the proposed solution, and it has been proven that it is able to avoid the detection of the provided detection examples.

1.Introducción

La emulación es uno de los mecanismos usados comúnmente en la detección y el análisis de malware [Po07], [Gy09], [Eg09], [Kr14], [St05]. El rendimiento del emulador, así como la exactitud de la emulación son aspectos claves a la hora de definir la capacidad real del emulador para llevar a cabo las tareas de detección y análisis. A lo largo de los años el *malware* ha ido buscando soluciones cada vez más elaboradas e implementando técnicas evasivas que dificultan su detección. Evitar la emulación, o modificar su comportamiento en entornos emulados ha sido una de las técnicas utilizadas. La capacidad del *malware* para implementar dichas técnicas se basa en las diferencias observables entre el sistema emulado y el sistema real.

Este trabajo explora los límites de la emulación y las posibles técnicas de detección basadas en los tiempos de ejecución de diversos trozos de código binario. El trabajo se limita a estudiar la emulación del procesador, dejado de lado otros elementos, como puede ser la emulación del sistema operativo. De la misma forma, el trabajo se centra en la arquitectura x64, aunque muchos de los resultados obtenidos se pueden extrapolar a otras arquitecturas.

Se presentan como resultado de la investigación diversos ejemplos de detección. Del estudio de los ejemplos de detección expuestos, se concluye que el problema fundamental en la emulación es que no tiene en cuenta los detalles de la implementación de la arquitectura, detalles que en muchos casos tienen efectos que se pueden observar en el procesador real, y por lo tanto deberían poder observarse en el entorno emulado.

Se presenta, así mismo, una prueba de concepto consistente en una emulación rudimentaria de la microarquitectura del procesador. Esta prueba de concepto permite que un emulador seleccionado pueda ejecutar los ejemplos expuestos sin que la emulación sea detectada. El trabajo concluye que el uso de un emulador implementado de forma que emulase la microarquitectura del procesador proporcionaría un entorno más difícil de diferenciar del entorno real, lo que tendría un impacto positivo en la lucha contra el *malware*. También se concluye que dicha implementación no es trivial, siendo la falta de documentación respecto de la microarquitectura y su rendimiento dos aspectos críticos y a tener en cuenta.

2.Trabajos previos

Existen gran cantidad de trabajos sobre el uso de la emulación y su papel en el análisis y detección de *malware*. Algunos ejemplos pueden ser [Po07], [Gy09] o [Eg09]. Los trabajos que analizan los límites actuales de su uso son más escasos y carecen de la profundidad de los primeros. Es notable destacar que la inmensa mayoría de los trabajos, si no todos ellos, se centran en la arquitectura x86.

En lo tocante a la detección del entorno, la virtualización ha atraído cierta atención, con trabajos bien conocidos, como [Ru05], que generó un gran debate, y con trabajos interesantes como [5], [To10], o recientemente [As17]. Ideas como la expuesta en [Ru05], basada en discrepancias en la dirección de memoria en la que se alojan estructuras globales a nivel de sistema operativo[1], no se pueden aplican al caso de la emulación. Sin embargo, algunas ideas expuestas en [Fr08] y analizadas en detalle en [To10] son totalmente aplicables a la detección de emulación. En el trabajo mencionado se expone un análisis basado en el tiempo de ejecución, y un estudio sobre los efectos de la virtualización sobre la caché.

Los trabajos en la detección de sistemas de emulación escasean y tratan el tema en menor detalle. Por ejemplo, [Na14] expone varios métodos que podrían ser usados por un malware para evitar su detección mediante la emulación. En todos los casos, la idea subyacente es que la emulación no es lo suficientemente precisa o lo suficientemente rápida como para emular

[1] Un sistema operativo virtualizado duplica estructuras como tablas de descriptores de interrupciones, tablas de descriptores globales, etc. Normalmente, un sistema operativo almacena estas estructuras en direcciones de memoria específicas. En el sistema virtualizado, se puede observar que dichas estructuras se encuentran en direcciones de memoria diferentes.

adecuadamente un trozo de código. Los ejemplos propuestos en este trabajo se centran exclusivamente, por tanto, en problemas asociados a la calidad de la implementación. Un trabajo más interesante es [Ra07], que hace referencia tanto a la detección de virtualización como de emulación. En concreto, en el caso de la emulación, el trabajo propone varios mecanismos de detección. Entre ellos, propone la detección de emulación basada en el análisis de tiempos relativos tanto en la ejecución de instrucciones[2] como en el comportamiento de la caché. El trabajo [Bl16] también expone varios mecanismos para la detección de emulación, entre los que cuenta la detección de tiempos. En este caso, propone la detección usando mediciones de tiempos absolutas basadas en invocaciones a servicios del sistema[3]. [Eg09] y [Ga04] también mencionan la detección de emulación basada en discrepancias en los tiempos, tanto relativos como absolutos.

Como se puede apreciar, los trabajos consultados tratan la detección de emulación basada en la diferencia de tiempos, pero lo hacen de forma superficial. Solamente se analizan tres casos: Los análisis relativos y absoluto de tiempos, y el efecto de la caché en los tiempos de ejecución en un entorno emulado con respecto al entorno real.

3. Malware

3.1. Definición

El término *malware* surge de la combinación de dos palabras: *malicious* y *software*. Este término se usa para hacer referencia a cualquier *software* no deseado, y es definido en [Mc00] como cualquier código añadido, modificado o eliminado de un sistema informático con el objetivo de causar daño o modificar el comportamiento previsto del sistema.

El *malware* se caracteriza por su capacidad para auto replicarse y para propagarse, para auto ejecutarse, para persistir en un sistema, y para evitar su análisis. Entre las propiedades del *malware* que evitan el análisis se encuentran la anti máquina virtual, la anti depuración, la anti emulación, el cifrado, y el empaquetado [Le13].

3.2. Clasificación

El *malware* se suele clasificar normalmente basándose en su comportamiento. Por ejemplo, virus para el *malware* que se replica modificando los programas informáticos instalados en el sistema informático, gusano para el *malware* que se propaga de computadora a computadora usando las conexiones de red, o puerta trasera para el malware que modifica un sistema informático habilitando un mecanismo para acceder al sistema evitando sus sistemas de seguridad. En algunos casos, virus se ha utilizado en castellano como un término genérico, siendo prácticamente un sinónimo de *malware*.

La clasificación del *malware* basándose en su comportamiento principal resulta insuficiente debido a dos razones.

Primero, no es lo suficientemente explicativa, de forma que una segunda clasificación basada en aspectos técnicos del *malware* es usada, mezclándose con la clasificación basada en comportamiento, y dando como resultado una clasificación poco intuitiva. Así, por ejemplo, un

[2] Comparando el tiempo de ejecución de una instrucción relativamente lenta, con una instrucción relativamente rápida

[3] Por ejemplo, invocar a la función Sleep, usando como argumento 1000ms

virus puede sub clasificarse en residente o no residente, pero también en EPO[4], polimórfico[5], etc. El problema fundamental es que esta clasificación, basada en detalles técnicos, no es mutuamente excluyente, de forma que un virus puede utilizar varias de las técnicas mencionadas. Es más, algunas de las técnicas pueden ser usadas por otro *malware* que no pertenece a la categoría virus.

La segunda de las razones está relacionada con la creciente complejidad del *malware*, que hace que tengan varios comportamientos de la clasificación al mismo tiempo, lo que a su vez ha llevado a que los analistas de *malware* clasifiquen el mismo *malware* con tipos diferentes. Una clasificación sin ambigüedades ha sido propuesta en [Le13], basada en observaciones del comportamiento del *malware* a bajo nivel.

4. Emulación

4.1. Definición

La emulación consiste en proporcionar un entorno simulado a una aplicación o código binario, de forma que dicho entorno simulado se comporta como el entorno real para el que la aplicación fue creado.

La creación de dicho entorno simulado conlleva el modelar distintos componentes *hardware* como el procesador y algunos chips del sistema, como controladores de interrupciones o de bus, DMA, etc.), dispositivos de entrada/salida, un posiblemente un sistema operativo.

4.2. Clasificación de la emulación

Se han estudiado diversos aspectos importantes en la emulación, que pueden dar lugar a diversas clasificaciones posibles [Ti00]. Por ejemplo, desde el punto de vista de la exactitud de la emulación, se pueden definir varios niveles. Otra posible clasificación podría basarse en el grado de emulación. Esto es, en cuanto del sistema es basado en *software* y cuanto en *hardware*.

En algunos trabajos, la virtualización se contempla como un caso específico de emulación. En lo que se refiere al presente trabajo, se opta por una visión más restringida del concepto de emulación, separado del de virtualización, en la línea de trabajos como [Wa15]. En este caso, la característica fundamental que permite diferenciar entre emulación y virtualización es que la primera no comparte necesariamente ninguna característica física entre el sistema emulado y el sistema en el que dicho emulador se ejecuta. Existe otra característica importante que diferencia ambos sistemas. En el caso de los sistemas emulados, el grado de control sobre el código emulado es mucho mayor, y este mayor control no es una diferencia cuantitativa, sino cualitativa, puesto que permite un análisis detallado del código emulado sin que la emulación en sí misma se vea afectada, lo cual no es posible en el caso de los sistemas virtualizados [Kr14].

---

[4] *Entry Point Obfuscation.* Este tipo de virus intenta, al llevar a cabo la infección de un fichero, que el punto de entrada del ejecutable no se modifique sustancialmente.

[5] Al infectar otras aplicaciones, los virus polimórficos generan copias de sí mismos con variaciones respecto al virus original. Estas variaciones no modifican el comportamiento del virus, y se llevan a cabo normalmente cifrando partes del virus con una clave aleatoria. Se pueden encontrar casos más elaborados que utilizan algoritmos de selección de instrucciones o asignación de registros con el objetivo de generar un conjunto de instrucciones diferente en cada infección.

4.3. La emulación en la detección de malware

Como se describe en [Sz01], la emulación fue una de las piezas clave en la lucha contra el *malware* cuando los virus polimórficos comenzaron a hacerse más comunes. Desde entonces, los sistemas de emulación han sido usados de forma habitual para el análisis y la clasificación de *malware* [Gy09], [Kr14].

Como se menciona en [Kr14], aunque la emulación es un mecanismo eficaz desde el punto de vista teórico, la aplicación práctica de esta idea encuentra diversas dificultades. Una de ellas es la necesidad de emular un sistema operativo relativamente complejo, y con muchos detalles no documentados. Lo mismo se puede decir del procesador, donde la documentación de la arquitectura del juego de instrucciones no es suficiente para llevar a cabo una emulación fidedigna, y donde los detalles de la microarquitectura no están siempre completamente documentados.

Otro de los problemas clave asociados a la emulación es el tiempo de ejecución del código emulado. Cada instrucción del programa emulado se puede traducir en decenas o cientos de instrucciones ejecutadas para su emulación. Se han estudiado diversos mecanismos y técnicas para mejorar el rendimiento de la emulación, como por ejemplo [St05]. Nótese que, a pesar del título de trabajo, el resultado final sigue teniendo cabida en la definición de emulación usada.

5. Elección del emulador

5.1. Requisitos

Dado que el estudio se centra en las detecciones de emulación basadas en tiempos, el emulador elegido debe cumplir como mínimo los siguientes requisitos:

- Debe tratarse de un emulador basado solamente en software. Esto es, quedan excluidos los sistemas de emulación que hacen uso de virtualización[6], parcial o totalmente. Este requisito es necesario debido al grado de control y cantidad de información que un sistema de emulación orientado al análisis de *malware* necesita[7].

- Debe ser un emulador con código abierto, con una licencia compatible con este trabajo de fin de master, y debe ser gratuito.

- Debe emular correctamente el juego de instrucciones completo de la arquitectura x86, tanto de 32 bits como de 64 bits[8].

- Debe proporcionar un rendimiento aceptable. Es difícil enunciar este requisito de forma objetiva. Un emulador utilizado en el análisis de *malware* debe proporcionar un alto rendimiento debido a algunos sus casos de uso típicos. Por ejemplo, una pasarela de correo puede recibir miles de correos por hora. Un producto de seguridad puede analizar todos los ficheros adjuntos a

---

[6] Ya se había mencionado que la definición de emulación usada en este documento no incluye la virtualización. Se menciona de nuevo la virtualización en este apartado dado que podrían existir sistemas híbridos haciendo uso de la virtualización.

[7] En este sentido, se pueden consultar los criterios formales para la virtualización, por Popek y Goldsberg [Po74], donde se puede apreciar que uno de ellos es que una cantidad notable de las instrucciones máquina deben ser ejecutadas sin intervención del sistema de virtualización. Como es obvio, este criterio limita notablemente la posibilidad de extraer información o de analizar el código emulado. Aunque el documento mencionado es antiguo, su contenido sigue estando, esencialmente, vigente.

[8] Se excluye la emulación de algunos detalles de la arquitectura, como contadores de rendimiento, registros o instrucciones presentes en versiones específicas del procesador, etc.

los correos. Los ficheros ejecutables son analizados, y durante dicho análisis son emulados para obtener información sobre su comportamiento, como parte del proceso de desempaquetado si se detecta que el ejecutable está empaquetado, para aplicar firmas que dependan directamente de la emulación, etc. La emulación debe ser lo suficientemente rápida como para que la pasarela no se convierta en un cuello de botella. Todo lo dicho para el ejemplo anterior también se aplica a otros escenarios donde le rendimiento es aún más crítico. Por ejemplo, una suite de seguridad instalada en un equipo de escritorio puede contar con un componente residente que analice los ficheros a los que se accede. Al ejecutar una aplicación, el residente analiza el fichero ejecutable antes de permitir su ejecución. Un emulador demasiado lento tendría en este caso un efecto negativo evidente al demorar el comienzo de la ejecución de todas las aplicaciones.

Informalmente, este requisito se podría enunciar como la necesidad de que el rendimiento del emulador sea los suficientemente bueno como para que pueda ser considerado para su uso en un producto de seguridad. Podemos asumir que, si un emulador implementa técnicas orientadas a la mejora del rendimiento, dicho emulador será aceptable. La traducción dinámica de código[9] es quizás la más conocida de estas técnicas, pero no la única. La evaluación deferida de flags[10], las optimizaciones basadas en caché de trazas[11], u optimizaciones relacionadas con la traslación de páginas son otras técnicas que pueden mejorar notablemente el rendimiento de un emulador.

Además de los requisitos expuestos anteriormente, existen otras cualidades deseables:

- Tratarse de un emulador cuyo uso sea lo más simple posible. En este caso simple significa simplicidad y conveniencia en cuanto a interfaces, buena documentación y acceso a ejemplos útiles.

- Poder emular diversas plataformas. Aunque inicialmente el trabajo se centra en la arquitectura x86, no se descarta el llevar pruebas básicas en otras plataformas.

- Poder ejecutar trozos arbitrarios de código. Es decir, que no sea necesario generar un fichero ejecutable de un formato específico para poder llevar a cabo una prueba en el emulador. Esta flexibilidad agilizaría el trabajo y permitiría llevar a cabo más pruebas.

5.2. Emuladores contemplados

El primero de los emuladores que se han contemplado como base para el trabajo es Qemu[12]. Este emulador proporciona un gran número de funcionalidades, entre las que se encuentra la virtualización, la emulación a nivel de sistema (incluyendo la emulación de periféricos), y además proporciona un buen rendimiento. Como se indica en su documentación, el emulador implementa la traducción dinámica de código con el fin de optimizar su rendimiento. Además, es capaz de emular diferentes arquitecturas (x86, ARM, MIPS, etc.)

---

[9] La traducción dinámica de código es una técnica basada en la compilación en tiempo de ejecución, de forma que el código emulado es convertido en código que puede ser ejecutado de forma nativa. Esta técnica, también conocida como *JIT (Just-In-Time)* es clave en la implementación de las máquinas virtuales de muchos lenguajes interpretados.

[10] *Lazy Flags Evaluation* en inglés. En un procesador, cada operación llevada a cabo por las unidades aritmético-lógicas actualiza el registro de *flags* de acuerdo con los resultados de las operaciones. Un emulador puede ignorar dicha actualización si ello no afecta la corrección de la emulación.

[11] Una caché de trazas almacena información de las instrucciones ejecutadas, de forma que si la misma instrucción se vuelve a ejecutar en un futuro cercano no es necesario volver a descodificar la instrucción desde cero. Este tipo de optimizaciones se pueden ver tanto en emuladores como en la implementación de microarquitecturas. En el caso de los procesadores Intel x86, esta caché se suele llamar *DSB (Decoded Stream Buffer), Decoded ICache, o Decoded μop Cache*.

[12] https://www.qemu.org

Otro de los emuladores contemplados es Bochs[13]. Al igual que Qemu, Bochs emula no solo el procesador, sino también un conjunto de periféricos y una BIOS ad hoc, proporcionando una emulación a nivel de sistema. Las principales diferencias entre Bochs y Qemu son:

- Bochs es más lento que Qemu, implementando algunas optimizaciones como la evaluación deferida de los *flags*, pero sin llegar al rendimiento ofrecido por Qemu.

- Bochs se centra exclusivamente en la arquitectura x86

Unicorn es un proyecto basado en Qemu, pero que se centra en la emulación del procesador, dejando de lado el intentar emular un sistema completo. Debido a centrarse exclusivamente en la emulación del procesador, su interfaz de usuario es sencilla, comparada con Qemu, y además permite la ejecución arbitraria de trozos de código binario.

Proyectos como VirtualBox[14] y Xen[15] se basan exclusivamente en la virtualización, por lo que no cumplen con los requisitos exigidos por este trabajo.

JPC[16] es un emulador de x86 implementado en Java, y que pretende ofrecer una emulación a nivel de sistema, similar A Bochs, por ejemplo. Intenta explotar las optimizaciones existentes en la máquina virtual de Java para llevar a cabo una emulación con un rendimiento razonable. Otro proyecto similar es Dioscuri[17], basado en JPC. Los dos problemas principales en estos dos proyectos son que no parece que exista un desarrollo activo (La última versión de JPC tiene cuatro años), y que no parece que puedan llevar a cabo una emulación muy fiable (La documentación indica que pueden emular Windows 95, pero no emula otros Windows más modernos correctamente)

5.3. Emulador seleccionado

De todos los emuladores analizados, se seleccionaron Qemu y Unicorn, por ser los que mejor se adaptaban a los requisitos del proyecto. Tras revisar las interfaces disponibles, y llevar a cabo una pequeña prueba de concepto, se ha decidido elegir Unicorn, debido a que era el que ofrecía una interfaz más versátil y que se adaptaba mejor a las necesidades del trabajo. La posibilidad de poder ejecutar trozos arbitrarios de código de forma sencilla y pudiendo ignorar la inmensa mayoría de la configuración previa necesaria en otros sistemas es clave a la hora de tomar la decisión, dado que dicha capacidad ofrece una gran agilidad. Las posibilidades en lo tocante a instrumentación disponibles en Unicorn es otro de los aspectos decisivos, puesto que ofrecen el tipo de interfaz necesario para el análisis detallado llevado a cabo durante el análisis de *malware,* así como para el desarrollo de este trabajo.

6. Detección de emulación

6.1. Introducción

Existen diversas opciones a la hora de detectar que el sistema bajo el que se ejecuta un código es un sistema emulado. La más obvia es detectar diferencias en el comportamiento debido a la implementación defectuosa o incompleta del juego de instrucciones de la arquitectura. En el caso de la emulación de un sistema completo, la misma idea se puede aplicar a los periféricos o al

---

[13] http://bochs.sourceforge.net

[14] https://www.virtualbox.org

[15] https://www.xenproject.org

[16] https://github.com/ianopolous/JPC

[17] http://dioscuri.sourceforge.net/index.html

conjunto de servicios proporcionados por el sistema operativo. Eligiendo la opción más generalista y evitando hacer asunciones acerca del sistema operativo, una correcta emulación del juego de instrucciones no es suficiente para garantizar que la emulación no puede ser detectada. Trabajos anteriores han mostrado que el análisis de los tiempos de ejecución puede ser usado para llevar a cabo dicha detección [Ra07], [Bl16], [Eg09], [Ga04].

Como se ha comentado, el presente trabajo pretende avanzar en esta dirección, mostrando que el problema fundamental es que la emulación debe tener en cuenta la microarquitectura subyacente, dado que ésta no queda oculta tras la arquitectura del juego de instrucciones. Ejemplos como [Li18] y [Ko18] evidencian que los ataques de canal lateral[18] son factibles en los procesadores modernos, [Ya16] muestra cómo, desde 2014 y hasta la fecha de publicación del trabajo, el estudio de este tipo de ataques había ganado tracción y se había incrementado notablemente, y trabajos como [Wa07] muestran que existe una preocupación por su peligrosidad e interés en buscar soluciones que los impidan o dificulten.

Tomando como ejemplo la arquitectura x64, los apartados sucesivos abordan distintos aspectos de su microarquitectura y de cómo ésta afecta los tiempos de ejecución, efecto que no es visible cuando el mismo código es ejecutado en el emulador seleccionado. Dicha discrepancia puede usarse como prueba de emulación.

6.2. Descripción de la Microarquitectura x64

En este apartado se presenta una descripción general de la microarquitectura típica de los procesadores x64. La descripción se centra en los aspectos de la microarquitectura que pueden resultar útiles para entender mejor las explicaciones relativas a los ejemplos de detección de emulación expuestos en los siguientes apartados.

A no ser que se indique lo contrario, toda la información técnica mencionada relativa a los procesadores en éste, así como en los sucesivos apartados, se menciona en [In16], [In16-2] o [Fo18].

La arquitectura x64 es una arquitectura relativamente compleja, con instrucciones de tamaño variable, de entre 1 y 16 bytes. Estas instrucciones se traducen a un conjunto de instrucciones internas, más sencillas, llamadas *micro opcodes*. Son estos *micro opcodes* (μ*ops* en lo sucesivo), lo que el procesador ejecuta. Su microarquitectura[19] es *super escalar*[20], fuera de orden[21], y *super pipelined*[22].

Debido a la complejidad de su juego de instrucciones, los procesadores cuentan con un *front-end* relativamente complejo, encargado de la descodificación de las instrucciones, de su traducción en μops, y de su almacenamiento en una cola de μops pendientes de ser ejecutados. Dicha cola se denominada *IDQ*[23]. Este proceso se lleva a cabo en el orden en el que las instrucciones aparecen en el programa.

---

[18] Ataques basados en la propia implementación de un sistema

[19] Obviamente, no es correcto hablar de una microarquitectura x64. Existen muchos procesadores x64 con implementaciones diferentes. Aquí se hace referencia a características que son comunes a todas las implementaciones. Cuando las características mencionadas sean aplicables solo a una microarquitectura concreta, se indicará usando su nombre.

[20] Puede ejecutar varias instrucciones en paralelo

[21] Puede ejecutar las instrucciones en un orden diferente al que aparecen en el código

[22] Divide la ejecución de una instrucción en un número relativamente alto de fases, permitiendo que las primeras fases de la ejecución las instrucciones se ejecuten en paralelo con las fases finales de las instrucciones precedentes. Por ejemplo, la microarquitectura *Haswell*, introducida en 2013, puede llegar a estar ejecutando cerca de 200 instrucciones en paralelo.

[23] *Instruction Decode Queue*

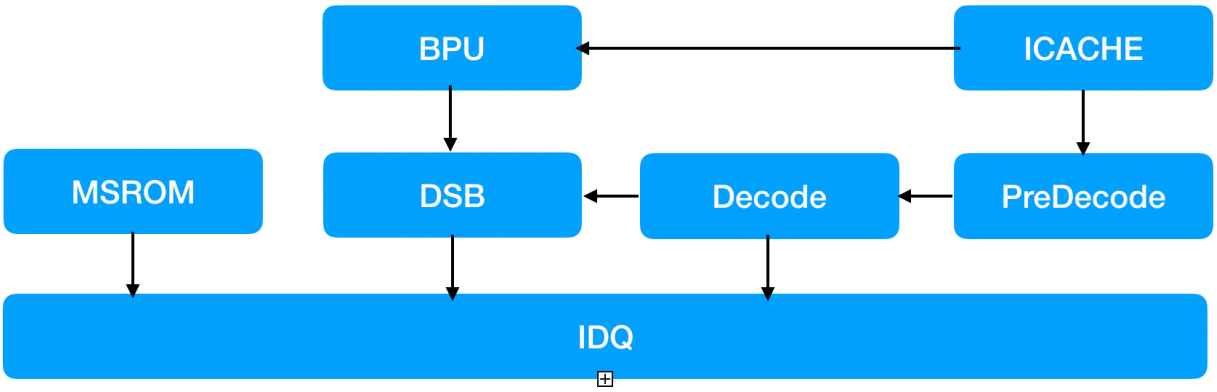

Fig 1: Visión esquemática del front-end

En la figura 1 se pueden ver los diferentes componentes que forman parte del *front-end*. El bloque *ICache* hace referencia a la caché de instrucciones[24]. El bloque denominado *PreDecode* se encarga de leer bloques de 16 bytes de la caché, de calcular el tamaño de cada instrucción en el bloque, y de proporcionar al elemento *Decode* las instrucciones encontradas[25]. El bloque *Decode* se encarga de descodificar las instrucciones, traduciéndolas en los *μops* adecuados, y de enviarlos tanto a la cola *IDQ* como al *DSB*.

Los procesadores modernos tienen cuatro descodificadores. Tres de ellos son solo capaces de descodificar instrucciones sencillas que se convierten en un único *μop*, mientras que el cuarto descodificador puede descodificar instrucciones complejas, que se convierten en hasta cuatro μops. El ancho de banda total es de hasta cinco μops por ciclo. En el caso de encontrarse instrucciones muy complejas, que se traducen en más de cuatro *μops*, es el componente *MSROM*[26] el se encarga de su descodificación, generando cuatro *μops* por ciclo hasta terminar de descodificar la instrucción.

El bloque *DSB* es una caché de μops, y tiene un ancho de banda de seis *μops* por ciclo[27]. Por último, el bloque *BPU* es la unidad de predicción de saltos, y se encarga de detectar la dirección de destino de los diferentes saltos[28] y de continuar la descodificación en la dirección prevista.

Es importante notar que los bloques que se encargan de enviar información a la *IDQ* son mutuamente excluyentes, de forma que, si el *MSROM* se requiere debido a la necesidad de

---

[24] En los procesadores Intel, la caché L1 se divide en dos partes, una para datos y otra para instrucciones. La caché de instrucciones suele tener un tamaño de 32Kb, una línea de caché de 64 bytes, y ser asociativa por conjuntos de ocho vías.

[25] Entre los bloques *PreDecode* y *Decode* existe una cola de instrucciones que permite al procesador continuar calculando la posición de instrucciones con cierta independencia de la velocidad de descodificación.

[26] *MicroSequencer ROM*

[27] Los anchos de banda mencionados en este apartado coinciden con la arquitectura *SkyLake*, introducida en 2015. En arquitecturas anteriores, el ancho de banda del *Decoder* y del *DSB* solía ser de cuatro *μops* por ciclo. Estos valores, como se puede ver, no cambian excesivamente entre microarquitecturas, siendo el tamaño de la *IDQ* un valor mucho más variable.

[28] Incluyendo saltos condicionales, incondicionales, llamadas a funciones, retorno de funciones, etc.

descodificar una instrucción compleja, entonces los bloques *Decode* y *DSB* están inactivos, reduciendo el rendimiento del *front-end* a cuatro *μops* por ciclo.

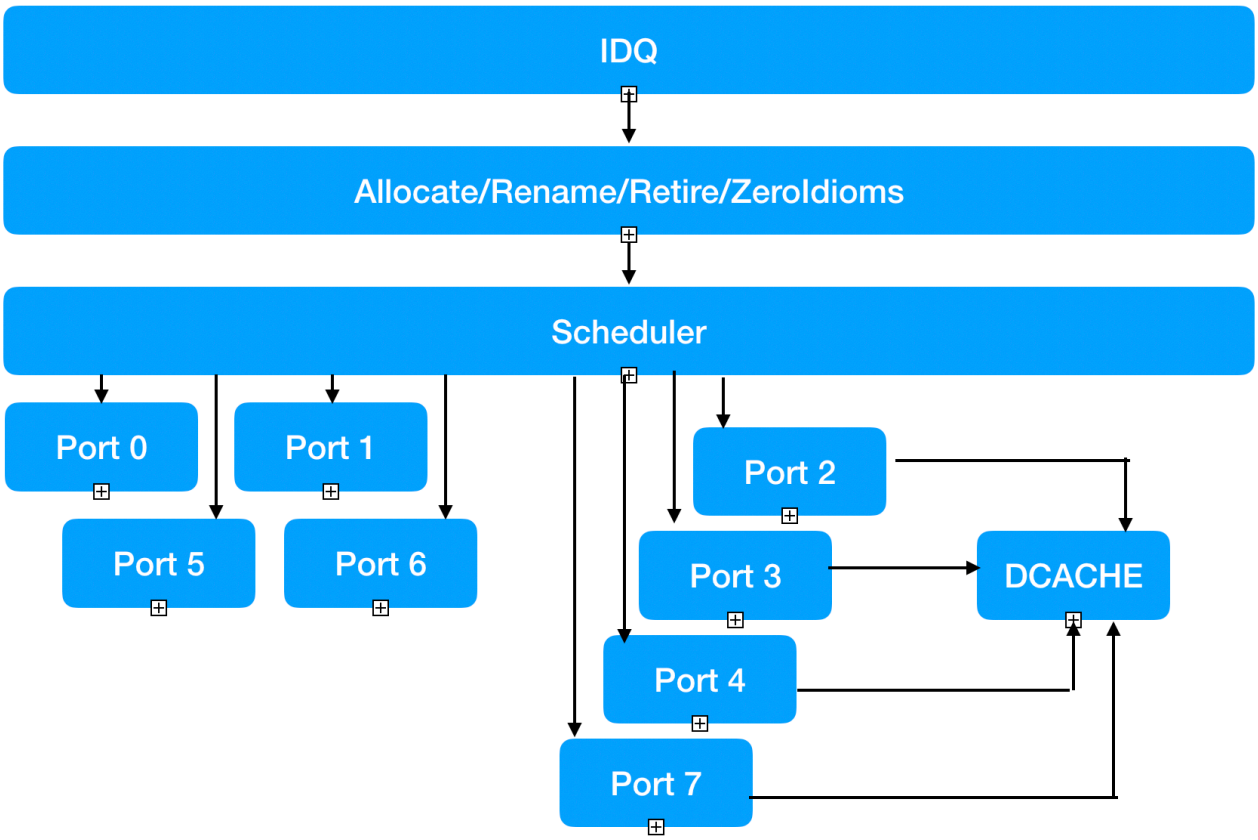

Fig 2: Visión esquemática del back-end

Como se puede apreciar, el rendimiento del *front-end* depende del tipo de instrucciones que aparecen en el programa, de su orden, de su alineamiento, de si la memoria en la que se encuentran almacenados se encuentra presente en la caché, o de si se han descodificado recientemente y están presentes en el *DSB*. Todos estos detalles pueden ser controlados en una aplicación, controlando el rendimiento del *front-end* y, por ende, el tiempo de ejecución de un trozo de código.

El contenido de la *IDQ* es consumido por un bloque denominado *allocate/rename*, que cumple varias funciones y es la interfaz entre el *front-end* que se funciona basado en el orden establecido por el programa, y el *back-end* que funciona basado en el orden establecido por las dependencias de los datos. Simplificando mucho su funcionalidad, se puede decir que se encarga de reordenar los *μops* según el flujo de datos, de forma que puedan ser ejecutados tan pronto como los datos de los que dependen estén disponibles, y de enviarlos al planificador para su ejecución. También se encarga de retirar las instrucciones, una vez que éstas han sido ejecutadas[29].

El bloque *allocate/rename* está asociado a varias optimizaciones como la detección de bucles, llamada *LSD*[30], y que consiste en la detección de bucles lo suficientemente pequeños como para poder estar almacenados dentro de la *IDQ*. Cuando estos bucles se detectan, los *μops* pueden

[29] Hacer los resultados de su ejecución visible a nivel arquitectónico. Este proceso tiene lugar en el orden en el que las instrucciones aparecen en el programa.

[30] *Loop Stream Detector*

servirse directamente desde la *IDQ* sin que sea necesario seguir descodificando instrucciones. Otra optimización llevada a cabo por la *IDQ*, y que es quizás más interesante por ser mucho más común son los llamados idiomas cero. Los idiomas cero son optimizaciones que reconocen una instrucción cuyo objetivo es hacer que el contenido de un registro sea cero. Cuando dicha instrucción es detectada, la operación se lleva a cabo directamente en este bloque, sin requerir la participación del planificador ni de los puertos de ejecución. El bloque *allocate/rename* es capaz de encargarse de la ejecución de dicha instrucción dado que conlleva básicamente un renombrado asociando un registro de la arquitectura con un registro microarquitectónico que contiene el valor 0. Cuando una instrucción puede ejecutarse sin la necesidad de un puerto de ejecución se dice que tiene latencia[31] cero.

Como se puede observar, existen diversos aspectos de la implementación de este bloque que tienen un impacto en el tiempo de ejecución en las instrucciones de un programa. Una operación *XOR*, requerirá menos tiempo en caso de que sus argumentos coinciden (Idioma cero). Es probable que dichas diferencias no estén modeladas adecuadamente en un emulador, de forma que se puedan usar como mecanismo de detección de emulación.

El planificador es el componente encargado de enviar los *μops* a las unidades de ejecución, denominadas puertos, que son los componentes que ejecutan la acción indicada por el *μop*. Los procesadores relativamente modernos cuentan con ocho puertos, cada una con diferentes capacidades de ejecución. Por ejemplo, la capacidad de leer y/o escribir datos en memoria suele ser proporcionada por puertos que no tienen la capacidad de llevar a cabo operaciones aritmético lógicas, como se muestra en la figura 2, donde solo los puertos dos, tres, cuatro y siete acceden a la caché de datos. Normalmente, no todos los puertos que acceden a la caché pueden leer y escribir. Por ejemplo, en arquitectas anteriores a la Sandy Bridge[32], el único puerto capaz de realizar lecturas de memoria era el puerto dos, de forma que el procesador podía realizar un máximo de una lectura de memoria en cada ciclo. A partir de la microarquitectura Sandy Bridge, los puertos dos y tres tienen capacidad de lectura, de forma que el ancho de banda de lectura se duplica, proporcionando un ancho de banda típico de 96 bytes por ciclo (64 bytes leídos y 32 bytes escritos). Además, el procesador cuenta con el espacio de almacenamiento necesario para permitir que varias instrucciones de acceso a memoria estén ejecutándose en paralelo en un momento dado. Este número suele aumentar con cada nueva implementación de la arquitectura. Por ejemplo, la microarquitectura Nehalem[33] permite 32 accesos de escritura y 48 accesos de lectura concurrentes.

Los puertos cero, uno, cinco y seis tampoco son iguales entre sí. Todos ellos pueden llevar a cabo operaciones aritmético lógicas simples, pero no todos ellos pueden llevar a cabo operaciones complejas, como por ejemplo la división, operación que está normalmente disponible solamente en el puerto cero. Por lo tanto, el procesador ejecuta las operaciones de división en serie. Este hecho también se puede apreciar revisando la información sobre la latencia y el rendimiento[34] de las instrucciones, donde se indica que, en el caso de la división, la latencia y el rendimiento coinciden.

El tiempo de ejecución de un conjunto de instrucciones depende, entonces, de las dependencias de sus datos, de la disponibilidad de los puertos de ejecución necesarios, del ancho de banda del procesador, etc. Todos estos detalles de implementación pueden, de nuevo, ser explotados para detectar discrepancias en el tiempo de ejecución de diversos trozos de código.

---

[31] El número de ciclos de procesador que requiere la ejecución de todos los *μops* de la instrucción. Normalmente este valor es aproximado, y no tiene en cuenta retardos potenciales como por ejemplo fallos de caché.

[32] Introducida en 2011

[33] Introducida en 2008

[34] El número aproximado de ciclos de procesador necesarios desde que una instrucción comienza a ejecutarse hasta que la misma instrucción puede comenzar a ejecutarse nuevamente.

6.3. Consideraciones iniciales

Se han barajado diversas opciones a la hora de definir la estructura general de los diversos ejemplos de detección.

Se descarto el crear una aplicación completa que pudiese ser ejecutada tanto en un entorno real como en el entorno emulado puesto que ello requeriría una configuración relativamente compleja del emulador, sin que aportase valor alguno al objetivo del trabajo. La complejidad en la configuración del emulador viene dada, en gran medida, por la necesidad de crear un entorno de ejecución adecuado y de un mecanismo de carga del fichero ejecutable en el emulador.

```
EMU = emulate1
PROG = example1
RADARE = radare2

TEST_HEADER = emublk.h

LDFLAGS2 += $(LDFLAGS) -lm -lunicorn
RADAREFLAGS = -qc "aaa;s sym._detect;bf sym._detect;b-1;pc> $(TEST_HEADER)"

PROG_OBJS = detect.o main.o

all: $(EMU)

clean:
    @rm -f $(PROG) $(PROG_OBJS) $(TEST_HEADER) $(EMU) $(EMU).o

%.o: %.c
    @$(CC) $(CFLAGS) -o $@ -c $<

$(PROG): $(PROG_OBJS)
    @$(CC) $^ $(LDFLAGS) -o $@

$(TEST_HEADER): $(PROG)
    @$(RADARE) $(RADAREFLAGS) $(PROG) ||:

$(EMU).o: $(TEST_HEADER)
    @$(CC) $(CFLAGS) -o $@ -c $(EMU).c

$(EMU) : $(EMU).o
    @$(CC) $(CFLAGS) $^ $(LDFLAGS2) -o $@

.PHONY: all clean
```

Fig 3: Makefile usando Radare

La siguiente opción, que también fue descartada, se basaba en la idea de crear dos ficheros ejecutables diferentes. El primero de ellos se ejecutaría en el entorno real, conteniendo una función con el código de detección de emulación. El segundo ejecutable sería el host para el emulador, que obtendría el código de la función de detección del primer ejecutable, y la ejecutaría dentro del emulador. Para obtener el código de la función de detección se usaría el framework de ingeniería inversa *Radare*[35]. La única limitación notable impuesta a la implementación del código de la función de detección sería que debería ser independiente de la posición. La figura 3 muestra el fichero *Makefile* para la compilación de ambos ejecutables, donde la función de detección copiada de un ejecutable a otro se llama *detect*. La principal ventaja de esta opción es que permitiría implementar los ejemplos en cualquier lenguaje de programación compilado. Esta opción se descartó dado que los lenguajes de medio y alto nivel no resultan lo suficientemente expresivos como para controlar el código binario generado con el nivel de detalle necesario en este trabajo. Una opción habría sido utilizar un compilador con funciones intrínsecas que proporcionase un acceso más directo a las capacidades del procesador[36], pero no se ha visto ventaja alguna entre esta opción y escribir el código directamente en ensamblador.

Una vez que la necesidad de implementar los ejemplos de detección directamente en ensamblador se hizo evidente, se optó por una tercera opción, en la que se genera un solo ejecutable conteniendo una función de detección implementada en ensamblador[37], que es ejecutada y después emulada. Al igual que en el caso anterior, la limitación principal impuesta al código de la función de detección es que debe ser independiente de su posición. La figura 4 muestra el contenido de un fichero *Makefile* utilizado para los ejemplos.

---

[35] https://rada.re/r/

[36] El compilador de C/C++ de Intel es un notable ejemplo, con funciones intrínsecas permitiendo ejecutar prácticamente cualquier instrucción en ensamblador de forma directa.

[37] Nótese que la elección del ensamblador como lenguaje para implementar la función de detección habilita esta opción, puesto que no existe un mecanismo fiable para conocer el tamaño de una función en C/C++. En general, ni siquiera se puede asumir que una función será compilada generando un bloque de código compacto. En cambio, en ensamblador es posible conocer el tamaño de la función, y asegurarse de que su estructura es la esperada.

Todos los ejemplos comparten, entonces, una estructura similar. Un fichero en ensamblador, llamado *detect.asm*, y que contiene dos funciones: La función *detect*, que lleva a cabo la detección, retornando un valor *booleano* que indica si la emulación ha sido detectada, y la función *detect_length*, que indica el tamaño de la función *detect*, y que es usada para copiar la función de detección al emulador. El fichero de cabecera *detect.h*, que permite utilizar la función *detect* en C/C++. El fichero *main.c*, que es el esqueleto de la aplicación, y un fichero *Makefile* para llevar a cabo la compilación lo más cómodamente posible.

```
PROG = example$(EXAMPLE)
NASM = nasm

UNAME = $(shell uname -s)

ifeq ($(UNAME),Linux)
    NASMFLAGS = -g -f elf64
    LDFLAGS += -lpthread
endif

ifeq ($(UNAME),Darwin)
    NASMFLAGS = -g -f macho64
endif

LDFLAGS += -lm -lunicorn
CFLAGS += -O3 -Wall -DNDEBUG


PROG_OBJS = detect.o main.o
HEADERS = detect.h

all: $(PROG)

clean:
    @rm -f $(PROG) $(PROG_OBJS)

%.o: %.c $(HEADERS)
    @$(CC) $(CFLAGS) -o $@ -c $<

%.o: %.asm
    @$(NASM) $(NASMFLAGS) $<

$(PROG): $(PROG_OBJS)
    @$(CC) $^ $(LDFLAGS) -o $@
```

Fig 4: Ejemplo de Makefile

Se ha optado por minimizar el proceso de configuración del emulador, de forma que la emulación tan solo copia la función dentro del entorno del emulador y trata de ejecutarla. Al no llevarse a cabo una configuración adecuada, se añade una nueva restricción a la implementación de las rutinas de detección: No pueden usar ninguna instrucción que interactúe con la pila, puesto que ésta no ha sido inicializada debidamente. Por convenio, se decide que la función *detect* solo puede tener un punto de retorno, y debe ser la última instrucción de la función. El emulador comienza su ejecución en la primera instrucción de la función *detect*. La última instrucción (el retorno) no se copia al emulador, garantizando que la pila no se usa. Por convenio, también se decide que en caso de que la rutina de detección requiera de un área de memoria, esta será proporcionada como argumento. Esta decisión ánade una nueva restricción a la rutina de detección: El convenio de llamada utilizado por la función debe ser del tipo *fastcall*[38].

En cuanto al entorno de trabajo, se han utilizado dos entornos diferentes para llevar a cabo las diferentes pruebas. El entorno principal, donde se ha llevado a cabo el grueso del trabajo, cuenta con un procesador *Intel Core i5-7400*, con microarquitectura *Kaby Lake*[39]. El sistema operativo usado es *MacOs Mojave*. El segundo entorno, utilizado solamente para llevar a cabo un segundo test de las rutinas de detección, cuenta con un procesador Intel Core i5-5200U, con microarquitectura *Broadwell*[40], y con un sistema operativo *Debian 9.6*. También se ha usado una máquina virtual, usando el sistema *VirtualBox*, virtualizando un sistema operativo *Debian 9.6*, y ejecutándose en el entorno principal. El objetivo del entorno virtual fue el poder llevar a cabo pruebas de validez de forma ágil.

La estructura de la función de detección en los diferentes ejemplos es prácticamente siempre la misma. Dos bloques de código se ejecutan, calculando sus tiempos de ejecución. Los bloques de código están diseñados de forma que el bloque que se ejecuta más rápidamente en el entorno real se pasará a ser el que se ejecuta más lentamente en el entorno emulado. En los casos en que los ejemplos no se correspondan con este esquema, se indicará y se explicará la estructura de la función de detección. Ha de hacerse notar que los bloques de código se ejecutan una única vez. Cada uno de los ejemplos proporcionados tienen una probabilidad de detección errónea o de no

[38] Todos los sistemas x64 usan convenios de llamadas a funciones de tipo *fastcall*. Esta restricción se ha mencionado para proporcionar una lista completa de restricciones, pero es un mero formalismo. En concreto, el entorno de trabajo principal usa la convención *System V AMD64 ABI*, que se puede consultar en https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI

[39] Introducida en 2017

[40] Introducida en 2015

detección[41]. Una estructura más compleja, ejecutando los bloques mencionados varias veces, combinando diversos conceptos, etc. reduciría la probabilidad de error drásticamente. Se ha optado, por el contrario, por ejemplos más sintéticos, que muestren los conceptos utilizados con mayor claridad.

```
mfence
rdtsc
sal rdx, 32
mov rsi, rdx
add rsi, rax
```

Fig 5: Ejemplo de toma de tiempo, almacenándolo en psi

A la hora de analizar la probabilidad de error de los diversos ejemplos, se ha ejecutado cada uno de ellos un total de un millón de veces. Normalmente, la probabilidad de error se sitúa por debajo del 1%. En caso de que la probabilidad de error sea mayor, se indicará explícitamente.

El cálculo de los tiempos de ejecución se ha llevado a cabo utilizando la función *RDTSC*[42]. La figura 5 muestra un ejemplo de una toma de tiempo, almacenando el resultado en el registro *RSI*. La instrucción *RDTSC* no serializa la ejecución, por lo que instrucciones anteriores al *RDTSC* podrían estar aun ejecutándose, o instrucciones posteriores podrían haber comenzado su ejecución. Para atajar este problema, se sitúa una instrucción *MFENCE*[43] antes del *RDTSC*. Una última limitación impuesta a los ejemplos de detección implementados es que deben funcionar correctamente en modo usuario. Si un método de detección requiriese de la ejecución de una instrucción privilegiada, la aplicación de dicho método se vería seriamente limitado, hasta el punto de resultar prácticamente inútil desde el punto de vista práctico.

```
▶ ./example1
[*] Executing emulation detection function
[+] Emulation not detected
[*] Emulating emulation detection function
[+] Emulation detected
```

Fig 6: Resultado de un ejemplo de detección

6.4. Ejemplo 1: Rendimiento de las instrucciones

El manual de optimización de Intel [In16] documenta, como ya se ha mencionado, la latencia y el rendimiento de cada una de las instrucciones. Estos valores son aproximados, pero cuanto más distantes sean, especialmente el rendimiento de la instrucción, mayor será la probabilidad de poder observar dicha diferencia de forma empírica.

Hay que tener en cuenta es que el resultado de una instrucción tiene que ser computado tanto durante la emulación como durante la ejecución real. En los casos en los que la emulación y el sistema que ejecuta el emulador comparten la misma arquitectura puede observarse un comportamiento similar, dado que el emulador puede ejecutar efectivamente la instrucción emulada.

---

[41] Detección errónea se refiere a reportar que se detecta emulación cuando la función de detección se ejecuta en el entorno real. La no detección se refiere a no reportar la detección de emulación cuando la función de detección está siendo emulada.

[42] La instrucción *RDTSC* lee el contenido del contador de marca de tiempos, un registro que se incrementa con cada ciclo del procesador.

[43] *MFENCE* serializa las instrucciones de lectura y escritura de datos en memoria. El manual de intel recomiendo usar la instrucción *LFENCE* antes del *RDTSC*. En este caso, la implementación propuesta sigue la línea de la implementación del kernel de Linux, que usa *MFENCE* en algunos casos. Para más detalles, se pueden consultar los ficheros https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/include/asm/barrier.h?h=v4.20-rc7 y https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/include/asm/msr.h?h=v4.20-rc7

Un emulador puede llegar a traducir cada instrucción emulada en decenas o cientos de instrucciones ejecutadas, por lo que esta posibilidad es relativamente pequeña, pero se incrementa notablemente si el emulador implementa ciertas optimizaciones, como por ejemplo la traducción dinámica de código.

Una opción interesante a la hora de diseñar una detección es utilizar instrucciones que además de llevar a cabo una operación, tengan otros efectos microarquitectónicos concretos, o instrucciones que sean utilizadas para controlar algún aspecto del estado microarquitectónico sin modificar el estado arquitectónico. Este ejemplo se basa en el segundo caso, seleccionando las instrucciones *LFENCE* y *MFENCE*. Ambas instrucciones son barreras de memoria, y como tales sirven para serializar las operaciones de acceso a memoria, y son de utilidad en implementaciones multihilo. En principio, un emulador estándar solo necesita aceptar las instrucciones, pudiendo implementarlas como operaciones nulas. La instrucción *MFENCE* es mucho más lenta que *LFENCE*. Además, la instrucción *MFENCE* hace que el código al que acompaña se ejecute más lentamente que en el caso de *LFENCE*. Esto se debe a que la serialización impuesta por *MFENCE* se aplica a todas las operaciones de acceso a memoria, mientras que en el caso de *LFENCE* solo se aplica a las operaciones de lectura.

La figura 7 muestra dos bucles. El bucle 1 ejecuta una instrucción *LFENCE*, mientras que el bucle 2 ejecuta una instrucción *MFENCE*. La instrucción *XOR EAX, EAX* se añade al bucle para que éste lleve a cabo alguna computación extra. Nótese que la instrucción en el bucle no es serializada por ninguna de las dos barreras de memoria.

```
.loop1:
        lfence
        xor rax, rax
        dec r8
        jne .loop1
```

```
.loop2:
        mfence
        xor rax, rax
        dec r8
        jne .loop2
```

Fig 7: Rendimiento de las instrucciones

En la ejecución de estos dos bucles en la máquina real se puede apreciar que el bucle número dos es más lento que el bucle número 1, lo cual es consistente con los datos proporcionados por la documentación del fabricante. En cambio, cuando estos dos bucles se ejecutan dentro del emulador, el bucle número 1 es más lento que el bucle número dos. Además, este ejemplo de detección es uno de los más estables, con un ratio de error de aproximadamente un 0,1%. Aunque no se muestra en la figura, los bucles son ejecutados un total de 100 veces. El objetivo de ejecutar las instrucciones en un bucle es el de ejecutar suficientes instrucciones como para que las mediciones de tiempos sean lo más estables posibles, pero sin ejecutar más instrucciones de las estrictamente necesarias.

6.5. Ejemplo 2: Optimizaciones en allocate/rename

El bloque *allocate/rename* funciona como interfaz entre el *front-end* y el *back-end* del procesador. Ya se han mencionado los idiomas cero como ejemplo de optimización llevada a cabo por este componente, y que podrían ser usados para diseñar un mecanismo de detección de emulación. En este caso, se va a utilizar una optimización similar disponible en dicho bloque. A partir de la microarquitectura *Ivy Bridge*[44], ciertas operaciones de copia se ejecutan directamente en este bloque. En concreto, muchas de las copias en las que sus dos operandos son registros son completadas directamente en el bloque allocate/rename. Una limitación interesante es que el

[44] Introducida en 2012

registro origen y el registro destino no pueden coincidir, en cuyo caso la instrucción es enviada al planificador para su ejecución[45].

La figura 8 muestra los dos bucles usados en este caso para la detección de emulación. El bucle 1 ejecuta varias copias entre registros, mientras que el bucle 2 tiene el mismo número de copias, pero los operados de cada una de ellas siempre coincide. Como se ha indicado, en un procesador real el bucle 1 se ejecuta más rápidamente que el bucle 2. Nótese, además, que el cuerpo de bucle es lo suficientemente pequeño como para que se almacene en la *IDQ*, y pueda ser optimizado por el *LSD*. De esta forma, nos aseguramos que el rendimiento del *front-end* no tendrá impacto alguno en la ejecución de los bucles.

```
        mov r8, 128
.loop1:
        mov rax, rbx
        mov rcx, rbx
        mov rdx, rbx
        mov rax, rbx
        mov r9, rbx
        dec r8
        jne .loop1
```

```
        mov r8, 128
.loop2:
        mov rax, rax
        mov rcx, rcx
        mov rdx, rdx
        mov rax, rax
        mov r9, r9
        dec r8
        jne .loop2
```

Fig 8: Instrucciones con latencia cero en el bucle 1

Cuando estos dos bucles son emulados, el emulador no tiene en cuenta los aspectos mencionados, de forma que es el bucle 1 el que se ejecuta más lentamente.

6.6. Ejemplo 3: Super escalar

El siguiente es uno de los ejemplos más interesantes. En este caso, la estructura de la función de detección no sigue la estructura típica. El objetivo de este ejemplo es el detectar la emulación basándose en el hecho de que la microarquitectura del procesador es super escalar. Se ha comentado con anterioridad que los procesadores modernos cuentan con dos puertos para llevar a cabo una operación de lectura de memoria. Por lo tanto, un grupo de operaciones de lectura en memoria se podrán ejecutar de dos en dos siempre que no existan dependencias de datos. El ancho de banda también puede suponer una limitación, por lo que es crucial que no se supere el ancho de banda para la lectura, que es de 64 bytes.

```
        mov rax, [rdi]

        mfence
        rdtsc
        sal rdx, 32
        mov rsi, rdx
        add rsi, rax

        times 256 mov rax, [rdi]

        mfence
        rdtsc
        sal rdx, 32
        add rdx, rax
        sub rdx, rsi
        sar rdx, 8
        mov r9, rdx
```

Fig 9: Calculo del tiempo promedio de carga

La figura 9 muestra la primera fase de la lógica de detección de emulación. Su objetivo es calcular el tiempo de ejecución promedio de un conjunto de accesos de lectura, que por simplicidad son siempre un acceso a la misma dirección de memoria. Se comienza llevando a cabo una lectura de la dirección de memoria que se usará como zona de acceso. Esta primera lectura garantiza que la memoria se encuentre en la caché, evitando que un fallo de caché pueda alterar los cálculos. Después, se toma el tiempo inicial,

[45] Puede sonar extraño que una instrucción que no tiene efecto alguno sea enviada al planificador. Este comportamiento se debe a que la instrucción *NOP* es la codificación 0x90, que es en realidad una copia del contenido de *RAX* en *RAX*. Si esta instrucción no se enviase al planificador y pasase a tener latencia cero, el comportamiento de muchos bucles de espera activa que usan *NOP* en vez de *PAUSE* cambiaría drásticamente. Nótese, además, que la codificación de *PAUSE* coincide con un *NOP* con un prefijo.

y se ejecuta la operación de carga un total de 256 veces. Finalmente, se vuelve a tomar el tiempo, calculando el tiempo promedio por instrucción, que se almacena en el registro *R9*. Como se puede apreciar, no se ha introducido ningún bucle, sino que se han generado las 256 instrucciones. El objetivo es evitar que la introducción de instrucciones de control de flujo, en especial la predicción errónea asociada a la salida del bucle, alteren el cálculo del tiempo promedio.

La ejecución de cada instrucción lee ocho bytes de memoria. Se asume que *RDI* contiene una dirección correctamente alineada, de forma que el contenido leído esté alojado en una única línea de la caché. Se espera que el procesador pueda utilizar los dos puertos disponibles para ejecutar las operaciones de carga. Un aspecto interesante a tener en cuenta es que el código actual, tal y como se muestra en la figura 9, presenta un problema menor. Durante la ejecución del bloque de instrucciones usadas para calcular el tiempo promedio, el *prefetcher* cargará datos extra en la cache de datos L1. La documentación existente no explica claramente si esta acción puede afectar los tiempos de ejecución[46]. Llevar a cabo tres accesos a la dirección de memoria en cuestión antes de ejecutar el conjunto de instrucciones destinado a tomar la medición debería solucionar el problema. De todas formas, la detección de emulación funciona correctamente sin llevar a cabo modificación alguna, por lo que se ha optado por dejar el código sin modificar.

```
mfence
rdtsc
sal rdx, 32
mov rsi, rdx
add rsi, rax

mov rax, [rdi]

mfence
rdtsc
sal rdx, 32
add rdx, rax
sub rdx, rsi
```

Fig 10: Tomar el tiempo de una instrucción de lectura

La figura 10 muestra la segunda parte de la lógica de detección. Como se puede apreciar, se trata, simplemente, de tomar el tiempo de ejecución de una sola instrucción de lectura de memoria. La instrucción se encuentra precedida y sucedida por instrucciones *MFENCE*, sin que existan otras operaciones de acceso a memoria en el bloque de código limitado por las barreras de memoria. Por lo tanto, durante la ejecución de ese bloque, los puertos de ejecución que proporcionan acceso a memoria no se usarán, a excepción del puerto que se encargue de la ejecución de la instrucción de lectura. En la fase anterior, los dos puertos de ejecución con capacidad para leer datos de memoria estaban en uso, de forma que el tiempo promedio de ejecución calculado en el bloque anterior debe ser considerablemente menor que el tiempo necesario para ejecutar la instrucción mostrada en la Figura 10.

Al ejecutar la rutina de detección en la máquina real, se puede observar el comportamiento esperado. Sin embargo, al emular la rutina de detección, el tiempo de ejecución de la instrucción de carga mostrada en la figura 10 es menor que el tiempo promedio calculado al ejecutar el código mostrado en la figura 9.

6.7. Ejemplo 4: Comportamiento de la Caché

Utilizar el estado de la caché y su impacto en los tiempos de ejecución de las instrucciones de lectura y escritura en memoria es un concepto que ya ha sido tratado en trabajos previos. En este trabajo, por lo tanto, no se ahonda en los aspectos ya estudiados, pero se presentan otros ejemplos de funcionalidades asociadas a la caché y que también pueden ser utilizadas como mecanismo de detección de emulación.

Como es común en todos los sistemas con caché, el primer acceso a una dirección de memoria que no se encuentre en la caché lleva asociado un coste extra. En este ejemplo los dos bloques de código son exactamente el mismo, como se puede apreciar en la figura 11.

---

[46] Tampoco deja claro cuál es el funcionamiento esperado en este caso. La documentación indica que el *prefectcher* de datos de la caché L1 (*DCU Prefetcher*) se activa cuando se llevan a cabo accesos a direcciones de memoria crecientes. Sin embargo se indica en un ejemplo que tres accesos consecutivos a la misma dirección de memoria activarían al *prefetcher L1*.

```
        mov rdi, r8
        mov rcx, 256
.loop2:
        mov rax, [rdi]
        add rdi, 8
        sub rcx, 8
        jne .loop2
```

```
        mov rdi, r8
        mov rcx, 256
.loop3:
        mov rax, [rdi]
        add rdi, 8
        sub rcx, 8
        jne .loop3
```

Fig 11: Efecto de la caché en los tiempos de ejecución

En este caso los bucles están etiquetados como bucle 2, que es el primero en ejecutarse, y bucle 3, que es el segundo en ejecutarse. El buffer al que apunta *R8* no está en la caché, por lo que el bucle 2 debe ejecutarse más lentamente que el bucle 3. Cuando el bucle 2 comienza a ejecutarse, los accesos a memoria con un patrón de direcciones crecientes a distancias constantes activará el *prefetcher*, lo que hará que la diferencia de tiempos no sea excesivamente grande. La diferencia es, no obstante, suficiente como para que la detección funcione.

Estos dos bucles por sí solos no son suficiente para generar una detección. El emulador seleccionado se comporta de la misma forma que lo hace el hardware real. En la figura 12 se muestra el contenido del primer bucle. En él se puede ver que el buffer es inicializado. La instrucción utilizada para inicializar el buffer es *MOVNTPS*, que es una de las instrucciones no temporales. Este tipo de instrucciones permiten controlar el comportamiento de la caché al mismo tiempo que se lleva a cabo una lectura o escritura. En este caso, el buffer es inicializado, pero la caché no se actualiza basándose en el acceso a memoria. Es interesante notar que a no ser que un emulador quiera proporcionar información de conteo de tiempos precisa, no tiene por qué tener en cuenta las implicaciones microarquitectónicas de las instrucciones no temporales. Si el objetivo es emular solo el comportamiento, las instrucciones no temporales pueden ser implementadas exactamente como las funciones de acceso a memoria estándar.

```
        mov r8, rdi
        mov rcx, 256
.loop1:
        movntps [rdi], xmm0
        movntps [rdi+16], xmm0
        movntps [rdi+32], xmm0
        movntps [rdi+48], xmm0
        add rdi,64
        sub rcx,64
        jne .loop1
```

Fig 12: Inicialización sin tocar la caché

Entonces, la lógica de este ejemplo es cómo se explica a continuación. En un equipo real, el primer bucle, que utiliza instrucciones no temporales no modifica el estado de la caché, de forma que el bucle número 2 incurrirá en fallos de caché, haciendo dicho bucle notablemente más lento que el bucle número 3. En el caso del sistema emulado, o bien los dos bloques se ejecutan en tiempos similares al no tener encuesta ningún aspecto de la caché, o el bucle número 1 se implementa en base a operaciones estándar, haciendo que el bucle número 2 no incurra en fallos de caché, y haciendo, por lo tanto, que los bucles 2 y 3 se ejecuten en tiempos similares. Este segundo escenario es el que provoca la detección de emulación en el caso del emulador seleccionado.

6.8. Ejemplo 5: Comportamiento de la Caché (2)

La caché cuenta con *prefetchers* hardware, que carga en la caché de datos de forma automática, asumiendo que dichos datos van a ser necesarios. Esta decisión se hace en base a los patrones de acceso a memoria. En el caso de la caché L1, el *prefetcher* es activado cuando se detectan accesos a direcciones de memoria en dirección ascendiente. En el caso de la caché L2, su *prefetcher* es más complejo y puede ser activado por accesos a memoria en direcciones tanto ascendientes como descendientes, además de llevar a cabo lecturas adicionales basándose en las líneas de caché presentes en la caché L1.

```
        xor rax, rax
        xor rcx, rcx
.loop2:

        fld qword [rdi + rax + 64]
        mov eax, [rdi + rcx*4]
        prefetcht0 [rdi + rax + 64]

        mov r8, 64
.loop22:
        fldpi
        fdiv
        dec r8
        jne .loop22

        inc rcx
        cmp rcx, 16
        jne .loop2
```

```
        xor rax, rax
        xor rcx, rcx
.loop3:
        fld qword [rdi+rax + 64 + 1024]
        mov eax, [rdi + rcx*4]

        mov r8, 64
.loop33:
        fldpi
        fdiv
        dec r8
        jne .loop33

        inc rcx
        cmp rcx, 16
        jne .loop3
```

Fig 13: prefetching

Además de los *prefetchers* hardware, la arquitectura proporciona la opción de llevar a cabo *prefetching* por software, de forma que se puede indicar al procesador que cargue unos datos concretos en la caché. Al igual que en el ejemplo 1, este tipo de instrucciones, que tan solo se encargan de modificar el estado de la caché, pueden ser ignoradas por un emulador, proporcionando una posibilidad de detección.

La figura 13 muestra dos bloques de código. En el bucle 2 se utiliza el *prefetcher* software por medio de la instrucción *PREFETCH0*. El buffer de datos usado, cuya dirección se almacena en el registro *RDI*, no está en la caché. El buffer contiene en su inicio una serie de números aleatorios, de forma que la instrucción *FLD* cargará en cada iteración un dato de una posición de memoria, sin que el *prefetcher* hardware pueda predecirla ni cargar el bloque de memoria por adelantado. El bucle 2 usa el *prefetcher* software para que el sistema cargue los datos requeridos en la siguiente iteración en la caché mientras ejecuta el código de la iteración actual.

El bucle 3 no utiliza el *prefetcher*, de forma que la instrucción *FLD* siempre tendrá que esperar a que los datos sean leídos de la memoria principal. En este ejemplo el tiempo de la computación llevada a cabo con los datos leídos en un ciclo dado tiene que ser mayor que el tiempo de ejecución de la instrucción *PREFETCH0.* De no ser así, obtendríamos el efecto contrario, siendo el bucle 1 más lento que el bucle 2.

Existe un primer bucle que no es mostrado, que inicializa el buffer. La inicialización del buffer es necesaria para asegurar que las operaciones en ambos bucles se ejecutan con los mismos operandos[47].

Éste es uno de los ejemplos menos estable, con un ratio de error de aproximadamente el tres por ciento. El ejemplo también funciona correctamente sin el bucle de inicialización, pero el ratio de error aumenta ligeramente, llegando a fallar en un cinco por ciento de los casos.

6.9. Ejemplo 6: Reenvío de datos

Una de las optimizaciones más importantes en los procesadores de Intel, y que no se ha mencionado hasta ahora, es el reenvío de datos. Cuando una instrucción escribe un dato en memoria, y otra instrucción posterior lee dicho dato de memoria, el procesador es capaz de enviar el dato directamente de una instrucción a la siguiente, sin necesidad de esperar a que los accesos a memoria se completen.

[47] El valor de los operandos es uno de los factores que define el tiempo de ejecución de las divisiones, tanto de números enteros como en coma flotante.

```
        xor ecx, ecx
.loop1:

        mov [rdi + rcx * 8], rcx
        mov rcx, [rdi + rcx * 8]

        inc rcx
        cmp rcx, 64
        jne .loop1
```

```
        xor ecx, ecx
.loop2:

        mov [rdi + rcx * 8], ecx
        mov dword [rdi + rcx * 8 + 4], 0
        mov rcx, [rdi + rcx * 8]

        inc rcx
        cmp rcx, 64
        jne .loop2
```

Fig 14: Reenvío de datos entre escritura y lectura

Existen diversas restricciones para que el reenvío de datos mencionado se pueda llevar a cabo. La mayoría de las restricciones hacen referencia a los tipos de datos reenviados, su alineamiento en memoria, y a las posiciones en las instrucciones involucradas. Este ejemplo escoge una de las limitaciones y la intenta explotar construyendo un método de detección basado en ella.

Una de las restricciones existentes es que para que un dato escrito en memoria se pueda reenviar directamente a otra instrucción que lleva a cabo su lectura es necesario que el dato haya sido escrito por una única instrucción. Por ejemplo, la escritura de cuatro bytes que conforma un *DWORD* seguida de la lectura de dicho *DWORD* no es optimizado mediante este sistema. Esta restricción será usada en este ejemplo de detección de emulación. La figura número 14 muestra los dos bucles usados en el ejemplo. El bucle 1 escribe un Q*WORD* que es leído por la siguiente instrucción. En el bucle 2, la escritura del Q*WORD* se lleva a cabo en dos fases, con dos instrucciones de escritura que escriben un *DWORD* cada una. Se asume que la zona de memoria a la que los dos bucles acceden ya está presente en la caché cuando éstos se ejecutan. En caso contrario, el primer bucle incurriría en un coste extra probablemente mayor que la mejora obtenida de esta optimización. También se asume que los accesos a memoria están correctamente alineados, de forma que no se incurren en costes extra debido a problemas de alineamiento.

Este ejemplo es especialmente interesante dado que podría esperarse que el bucle número dos se ejecutase más lentamente independientemente de la optimización mencionada, dado que contiene más instrucciones, lo que conlleva más trabajo para el *front-end*, y más μ*ops* que ejecutar. Lo cierto es que, cuando el bucle 1 no se aprovecha de la ventaja proporcionada por la optimización discutida en este apartado, el tiempo de ejecución de ambos bucles es prácticamente el mismo, de forma que no pueden utilizarse para la detección de emulación. Este comportamiento se puede obtener, por ejemplo, haciendo que los accesos a memoria no estén debidamente alineados. La razón por la que el tiempo de ejecución de los bucles es similar a pesar de la diferencia en el número de instrucciones se debe a dos factores. El primero es que el tiempo de ejecución del código en los bucles está limitado por el ancho de banda del procesador, de forma que el resto de factores pasan a ser prácticamente irrelevantes. El segundo es una optimización conocida como combinación de escrituras, mediante la cual el procesador ensambla escrituras múltiples en una sola escritura.

Otra cuestión interesante es el que el emulador de referencia reporta que es el primer bucle el que se ejecuta más lentamente que el segundo. Este resultado no parece muy intuitivo.

6.10. Ejemplo 7: Macro fusión

La macro fusión es una optimización presente en todas las microarquitecturas relativamente recientes, y de la que no se ha hablado previamente. La combinación de instrucciones consistente en una comparación seguida de un salto condicional es muy común en el código

binario. Aunque la mayoría de las instrucciones actualizan el registro de *FLAGS*[48], es difícil que dicho registro contenga los datos adecuados para controlar el flujo de la aplicación, lo que obliga a introducir instrucciones de comparación explícitas, normalmente instrucciones *TEST* o *CMP*. La macro fusión detecta esta combinación de instrucciones, generando un único μop. Existen varias restricciones en la aplicación de esta optimización. La alineación de las instrucciones en memoria debe ser la correcta, y no todas las combinaciones de instrucción de comparación y salto condicional permiten la macro fusión[49]. Por ejemplo, el salto condicional basado en el estado de los *flags* de signo y paridad (*JS*, *JP, etc.*) no se fusionan con la instrucción *CMP.*

```
        mov rdi, r8
        xor r10, r10
.loop1:
        mov al, [rdi]
        movzx eax, al
        add r10, rax
        add rdi, 1
        cmp byte [rdi], cl
        jge .loop1
```

```
        mov rdi, r8
        xor r10, r10
.loop2:

        mov al, [rdi]
        movzx eax, al
        add r10, rax
        add rdi, 1
        cmp byte [rdi], 1
        jge .loop2
```

Fig 15: Macro fusión

La documentación disponible relativa a la macro fusión es poco clara en algunos aspectos. Hay, de todas formas, un comentario que dice explícitamente que una comparación de un valor en memoria con un valor inmediato seguido de una instrucción *JGE* inhibe la macro fusión. El ejemplo de detección presentado se basa en el comentario que se acaba de mencionar. La figura número 15 muestra los dos bloques usados en este ejemplo de detección.

El código en el bucle suma un conjunto de valores de tipo *char* almacenados en un buffer. El valor 0 indica el final de la lista. Para comprobar si se debe seguir ejecutando el bucle, el valor actual se compara con 1, ejecutando una nueva iteración si dicho valore es mayor o igual a 1. En el primer bucle, el valor 1 ha sido almacenado en el registro *CL,* mientras que en el segundo bucle el valor de comparación se da como un valor inmediato. Se asume que el buffer al que se accede está en la caché, de forma que ninguno de los bucles incurre en fallos de caché. De acuerdo al comentario mencionado anteriormente, al ejecutar ambos bucles en un procesador real, el primer bucle debe ser el único que aprovecha la macro fusión, siendo más rápido que el segundo. En un sistema emulado, se espera que el segundo bucle sea más rápido, puesto que los registro están modelados como variables y por lo tanto es de esperar que los valores inmediatos ahorren accesos a memoria.

Las pruebas realizadas con este ejemplo muestran que los resultados son consistentes con las expectativas. El bucle número 1 es el más rápido en la máquina real, mientras que en el emulador es el segundo bucle el que se percibe como el más rápido.

---

[48] Se pueden consultar ejemplos mostrados anteriormente, como el mostrado en la figura 7, donde no se lleva a cabo una comparación explícita. La instrucción *DEC* realiza una actualización parcial del registro de *FLAGS*. El salto condicional controla el flujo basándose en dicha actualización.

[49] Es interesante notar que antes de la microarquitectura *Nehalem*, la instrucción *CMP* no podía fusionarse con las instrucciones de salto condicional asociadas a tipos de datos con signo (*JL, JG, JNL*, etc.). Esta circunstancia planteaba una disyuntiva interesante, puesto que el uso de tipos enteros sin signo podían aprovechar la macro fusión, pero al mismo tiempo en C/C++ se suelen preferir los tipos enteros con signo, puesto que su comportamiento está parcialmente definido, abriendo posibilidades para potenciales optimizaciones.

6.11. Ejemplo 8: Delaminación

La delaminación es otro de los comportamientos interesantes en las microarquitecturas de Intel y que todavía no se ha introducido. Se ha mencionado que las instrucciones se traducen en *μops*, y que estos *μops* se almacenan en una caché, también conocida como *DSB.* Esta caché tiene una fase de post procesado, en el que algunos de los *μops* almacenados en el *DSB* se dividen en dos *μops* al ser enviados a la *IDQ*. Este proceso se conoce como delaminación. Por lo tanto, algunos *μops* en el *front-end*, dependiendo de su formato, se convertirán en dos *μops* en el *back-end.* Esta técnica permite optimizar el funcionamiento del front-end, gestionando los datos de forma más compacta.

Algunos de los ejemplos clásicos de delaminación están relacionados con accesos a memoria en los que el cálculo de la dirección efectiva es complejo y que conllevan una operación extra. En este ejemplo se utiliza la instrucción *ADD*, con el primer argumento conteniendo un acceso a memoria. Cuando el acceso a memoria conlleva una operación lo suficientemente compleja, se produce la delaminación, generándose un *μop* para calcular la dirección efectiva y llevar a cabo la lectura, y otro *μop* para llevar a cabo la adición. Si la operación de acceso a memoria es sencilla, entonces la delaminación no tiene lugar.

Uno de los casos en los que el cálculo de la dirección se considera demasiado compleja es cuando se usa un modo de acceso indexado en una instrucción que depende de varios registros. La figura 16 muestra los bloques usados en este ejemplo.

```
        mov rcx, 512
.loop1:

        add byte [rdi + 16], r10b
        add rdi, 1
        sub rcx, 1
        jnz .loop1
```

```
        mov rcx, 512
        mov rax, 16
.loop2:

        add byte [rdi + rax], r10b
        add rdi, 1
        sub rcx, 1
        jnz .loop2
```

Fig 16: Delaminación

El bucle número 1 utiliza un acceso indexado que no involucra a ningún registro a parte del registro que ejerce de base en el direccionamiento. En cambio, en el bucle número 2, la misma dirección de memoria es accedido (*RAX* contiene el valor 16), pero la instrucción involucra un total de tres registros. La instrucción en el bucle 2 se traducirá en dos μops dentro de la *IDQ*, a diferencia de la instrucción en el bucle 1.

El comportamiento en el procesador es el esperado. El bucle número 1 se ejecuta más rápidamente que el bucle número 2. En el emulador pasa justo al contrario. Es interesante notar que el uso de un direccionamiento indexado en bucle 1 no es caprichoso.

Se podría haber optado por un direccionamiento indirecto en el bucle 1, e inicializar *RAX* a 0 para la ejecución del bucle 2. Con estos cambios, el emulador pasa a reportar, al igual que el procesador real, el bucle 1 como el más rápido de los dos, de forma que se obtiene el mismo resultado independientemente de si la función de detección es emulada o ejecutada. La razón por la que el emulador detecta incorrectamente el bucle 1 como el más lento de los dos parece ser el coste asociado al análisis de la instrucción *ADD.*

6.12. Ejemplo 9: Determinismo

El ejemplo expuesto a continuación no sigue la estructura estándar común a la mayoría de los ejemplos expuestos. En este caso el ejemplo está estructurado como un gran bucle dentro del cual se ejecutan dos bloques de código y se miden sus tiempos de ejecución.

En cada iteración se incrementa un contador indicando cuál de los dos bloques ha sido el más rápido en la iteración actual. Además, los dos bloques de código tienen exactamente el mismo código, que se muestra en la figura 17. Como se puede apreciar, se trata de un trozo de código sin ninguna utilidad, y que se ejecuta en muy pocos ciclos de CPU.

```
nop
nop
nop
```

Fig 17: Código sin utilidad alguna

Lo interesante en este trozo de código es que, si se ejecuta millones de veces, siempre aparecerán pequeñas divergencias. Esto es debido a que existen multitud de aspectos que inciden en el tiempo de ejecución de un trozo de código. El estado del procesador, el número de instrucciones en el *Pipeline,* interrupciones, el estado de la caché, cambios de contexto llevados a cabo por el sistema operativo, etc. Si se analizan estas desviaciones, nos encontramos con una distribución normal, de forma que, al elegir dos ejecuciones de ese código al azar, las probabilidades de adivinar cuál de las dos ejecuciones es la más rápida es de aproximadamente el cincuenta por ciento. Por lo tanto, el contador que calcula cuántas veces el bloque 1 de este ejemplo es más rápido que el bloque 2 debe devolver un número cercano a la mitad del total de pruebas.

Como se ha comentado anteriormente, cada instrucción que es emulada se convierte en decenas o cientos de instrucciones ejecutadas en el procesador real. De forma similar a como los pequeños bucles ayudan a estabilizar los resultados de la prueba en la mayoría de los ejemplos de detección presentados, la ejecución de una gran cantidad de instrucciones para emular una única instrucción tiene ahora un efecto negativo, puesto que hacer que el resultado de la emulación sea mucho más determinista que la ejecución real. De esta forma, en la emulación se puede apreciar que, en las pruebas, la probabilidad de que uno de los bloques sea más rápido que el otro ya no se encuentra cercado al cincuenta por ciento, sino que se ha escorado.

En el ejemplo presentado, se ha optado por ejecutar un total de 128 pruebas, y se ha tomado el punto de corte alrededor del 20%. Los límites usados han sido exactamente 25 y 95. Si un bloque ha resultado ser el más rápido 95 veces o más, o el más lento 25 veces o menos, entonces se asume que el código está siendo emulado. En caso contrario, asumimos que el código está siendo ejecutado en un procesador real.

El ratio de error de este ejemplo es ligeramente superior, comparado con la mayoría de ejemplos expuestos hasta el momento. Se sitúa, aproximadamente, en el 3,5%. Sin embargo, no se ha trabajado extensivamente en el cálculo de los límites y del número de pruebas idóneo, por lo que se asume que la fiabilidad del método se puede incrementar sustancialmente si se lleva a cabo un análisis más detallado. El objetivo, demostrar que se puede detectar que el código está siendo emulado detectando un comportamiento más determinista de lo habitual, queda sobradamente cumplido.

6.13. Ejemplo 10: Micro Fusión

La micro fusión es otra de las optimizaciones que se puede utilizar como base para llevar a cabo una detección. La micro fusión es el ejemplo contrario al que se ha presentado con el nombre de delaminación.

En el caso de la micro fusión, dos μ*ops* de la misma instrucción se funden en un solo μ*op*. Las funciones que se benefician de la micro fusión son funciones que llevan a cabo un cálculo y en las que uno de sus parámetros es un acceso de lectura a memoria. En caso de que la micro fusión no sea posible, el rendimiento del *front-end* se ve afectado ya que las instrucciones a las que no puede aplicársele la micro fusión tendrán que gestionarse generando dos μ*ops*, lo que imposibilita el uso de los descodificadores simples.

La documentación indica que el direccionamiento basado en RIP inhibe la micro fusión en varios casos. Uno de ellos, por ejemplo, es si se utiliza con un valor inmediato adicional.

```
times 64 add rax, [rel _detect]
```

```
times 64 add rax, [rdi]
```

Fig 18: Micro Fusión

La figura 18 muestra los dos bloques usado para esta detección. El primer bloque, mostrado a la izquierda, utiliza un direccionamiento basado en RIP con un valor inmediato adicional, imitando uno de los ejemplos utilizados en la documentación como caso de inhibición de la micro fusión. El segundo bloque muestra una instrucción similar pero cuyo direccionamiento es indirecto no basado en RIP, por los que sí se beneficiaría de las posibilidades ofrecidas por la Micro Fusión. La dirección efectiva a la que acceden las instrucciones presentes en ambos bloques es la misma. Se trata de la dirección de comienzo de la función *detect.*

Es importante notar que no se han usado bucles en este caso. Se generan las instrucciones en una secuencia lineal. Esto es así dado que en caso contrario las instrucciones extra aprovecharían los otros descodificadores, haciendo que la diferencia de tiempos de ejecución fuese menos notable, y el *LSD* podría entrar en acción evitando el problema al no tener que descodificar la instrucción múltiples veces.

El comportamiento de este ejemplo es ligeramente menos estable que en otros casos. El emulador rara vez reporta el primer bloque como más lento que el segundo[50]. En cambio, hay casos en los que el procesador es capaz de ejecutar el primer bloque más rápidamente que el segundo. La probabilidad de error se sitúa en torno al 2,5%.

6.14. Otras pruebas

A lo largo de la investigación que ha llevado a los ejemplos de detección de emulación, se han llevado a cabo un número considerable de pruebas que no se han consolidado en una rutina de detección. En algunos casos el comportamiento dentro del emulador era similar al comportamiento de la rutina de detección ejecutada en el procesador real. En otros casos, la probabilidad de errar la detección era demasiado alta. Por último, en algunos casos el comportamiento esperado no era observable en el procesador, de forma que no era posible implementar una función de detección basada en el concepto analizado.

En este aparatado se mencionan algunos de los ejemplos que se han investigado infructuosamente. La lista no pretende ser una enumeración completa de los casos investigados, sino que solo se mencionan los ejemplos que resultaron más interesante, seguramente por ser lo que parecían más prometedores a priori.

Una de las ideas que a priori era muy prometedora estaba relacionada con la predicción de saltos. Se llevaron a cabo tres tipos de pruebas en esta dirección.

Se intento detectar una diferencia de comportamiento basándose en la predicción estática de las instrucciones de salto[51]. Uno de los problemas asociados a esta prueba era que una vez que la instrucción se ejecuta, el procesador cuenta con información dinámica para predecir el resultado, por lo que el diseño de la prueba requería que las instrucciones de salto fuesen ejecutadas una única vez. Otro de los problemas asociados es que no todas las microarquitecturas modernas cuentan con predicción estática de saltos. Tras efectuarse varias pruebas, no fue posible implementar una función en la que se observase el comportamiento esperado.

---

[50] Aunque no está claro cuál es la razón para este comportamiento.

[51] La predicción estática de saltos es relativamente simple. Los saltos incondicionales siempre se ejecutan. Cuando un salto condicional se dirige hacía una posición de memoria superior a la dirección en la que se encuentra, la predicción estática asume que dicho salto no se tomará. Cuando un salto condicional se dirige hacía una dirección de memoria inferior a la dirección en la cual la instrucción se encuentra, la predicción estática asume que dicho salto si se tomará.

Se intentó detectar una diferencia de comportamiento usando la información de pista asociada al salto condicional[52]. Se llevaron a cabo diversas pruebas sin que, al igual que en el caso anterior, se pudiese observar el comportamiento esperado. En este caso las expectativas de éxito eran bajas [Fo18].

Tras revisar en detalle los contenidos de [In16], [In16-2] y [Fo18], se llega a la conclusión de que es muy probable que las microarquitecturas modernas no tengan ni predicción estática de código ni usen las pistas de los saltos condicionales. Desde un punto de vista microarquitectónico, la razón podría venir dada por el diseño del *front-end*. Las predicciones de saltos que se mencionan requieren la descodificación de la instrucción, mientras que un BPU que solo implementase predicción dinámica no requeriría la descodificación de la instrucción.

```
        xor r8, r8
        mov rax, 1
        mov rcx, 16
        jmp .loop1
.loop1m:
        inc r8
        mfence
        rdtsc
        sal rdx, 32
        mov rsi, rdx
        add rsi, rax

        xor rax, rax

.loop1:
        cmp rax, 0
        je .loop1e
        sub rcx, 1
        jnz .loop1

        cmp r8, r8
        je .loop1m

.loop1e:

        mfence
        rdtsc
        sal rdx, 32
        add rdx, rax
        sub rdx, rsi
        mov r9, rdx
```

Fig 19: Predicción errónea en el salto condicional

Se intentó detectar una diferencia de comportamiento basándose en la predicción dinámica de saltos. La primera idea fue forzar una predicción errónea de un salto. La figura 19 muestra el intento de medir el tiempo de ejecución de un salto cuya predicción es errónea (etiquetado como *JE .loop1e*). La diferencia entre éste y el tiempo de ejecución de saltos condicionales con una predicción acertada es evidente. Este mecanismo no pudo usarse como método de detección, puesto que el comportamiento observado en el emulador coincide con el comportamiento en el procesador real.

Cuando el intento anterior falló, se intentó llevar a cabo una prueba más elaborada. La idea era, de nuevo, forzar una predicción errónea en un salto condicional, pero en vez de medir el tiempo asociado al salto, medir el tiempo asociado a algún cambio microarquitectural provocado por la ejecución especulativa de código. Se optó por un acceso a una zona de memoria que no estuviese previamente en la caché. La ejecución especulativa de código ejecutaría el acceso a memoria provocando el fallo de caché, lo cual provocaría una diferencia en el tiempo de acceso a la zona de memoria en cuestión tras la predicción incorrecta. Este mecanismo parece funcionar correctamente, pero el porcentaje de detecciones incorrectas es demasiado alto, superando el 20%. El mecanismo, al menos conceptualmente, parece sólido y los resultados obtenidos no lo descartan totalmente. Un trabajo más exhaustivo en esta dirección podría dar como resultado un sistema de detección de emulación increíblemente eficaz. Este tipo de análisis no se encuentra dentro de los objetivos de este trabajo.

Otra prueba de concepto que no llego a traducirse en un mecanismo de detección fueron las pruebas con el *LSD*. Se han utilizado bucles pequeños en muchos de los ejemplos para poder aislar el ejemplo de posibles problemas en la descodificación. Sin embargo, al intentar generar un ejemplo de detección basándose en el *LSD*, se encuentra que el observar directamente los beneficios de esta optimización es extremadamente difícil, de forma que no se dispone de un código razonable que permita llevar a cabo la detección.

---

[52] Los saltos condicionales aceptan un prefijo dando una pista sobre si es probable que el salto se ejecute o no.

Se han llevado a cabo varias pruebas con las dependencias de datos, tanto dependencias reales como falsas dependencias provocadas por la reutilización de registros. Uno de los ejemplos, basado en la combinación la instrucción *DEC* para decrementar el contador de un bucle, junto con la instrucción *SHL* fue obtenido directamente de [In16]. En este ejemplo, la dependencia de datos venía dada por la actualización parcial del registro de *flags* por parte de la instrucción *DEC*, que tenía como efecto negativo que la instrucción *SHL* fuese ejecutada más lentamente[53]. Diversas pruebas realizadas no mostraron una diferencia de tiempos suficientemente consistente como para poder implementar una función de detección.

Se han llevado a cabo varias pruebas con los prefijos de cambio de tamaño (0x66, 0x67). Estos prefijos provocan una bajada de rendimiento por parte del pre descodificador del *front-end*. Esta bajada de rendimiento podría usarse para detectar la emulación. El problema mencionado es difícil de observar, y todas las funciones de detección razonables implementadas basándose en este concepto no ofrecían un comportamiento diferente en el emulador.

En muchos casos, diversas optimizaciones dependen de la correcta alineación de los datos. El tiempo necesario para ejecutar una instrucción que accede a memoria también depende de la adecuada alineación de los accesos a memoria. Lo mismo pasa con el *front-end*, cuyo rendimiento depende de la alineación de las instrucciones en memoria.

Se han realizado diversas pruebas relativas a la alineación de las instrucciones en memoria. La más interesante de todas ellas se hizo en combinación con las pruebas de prefijos de cambio de tamaño mencionados anteriormente. Existe una situación que afecta negativamente al *front-end* de forma notable, y se da cuando una instrucción con un prefijo de cambio de tamaño cruza el límite de una línea de datos del pre descodificador[54]. Si la instrucción comienza en el decimotercer byte de la línea, usa un direccionamiento con un registro y un valor inmediato, y usa un prefijo de cambio de tamaño, entonces el *front-end* sufre lo que se conoce como una doble detención[55], lo que causa un retraso notable en el *front-end*. Desafortunadamente, no se ha conseguido implementar una detección de emulación basada en este detalle microarquitectónico.

Se han llevado a cabo una serie de pruebas con las instrucciones no temporales. Un aspecto interesante de las instrucciones no temporales que realizan una escritura en memoria es que las peticiones no requieren las peticiones asociadas al protocolo de caché. En escrituras de memoria normales, estas peticiones comparten el bus con la escritura. Por lo tanto, las instrucciones no temporales, al no requerir las peticiones asociadas al protocolo de caché, disfrutan de más ancho de banda, pudiendo escribir 64 bytes por ciclo siempre que se consiga que los datos a escribir se combinen en una única transacción en el bus de datos. Este detalle parecía poder usarse como base para una detección.

No fue posible diseñar una función en la que se pudiese observar el comportamiento descrito. [Fo18] describe posibles efectos negativo ante la presencia de muchas instrucciones no temporales, comportamiento que se acercaba mucho más al observado que el descrito en [In16]. Los datos obtenidos de las pruebas realizadas no son lo suficientemente concluyentes, por lo que no se dispone de una explicación razonable para el comportamiento observado.

El resto de pruebas que han sido infructuosas resultan menos interesantes, puesto que de alguna manera el efecto deseado estaba cubierto por algunos de los ejemplos expuestos, o estaban relacionadas con alguna de las pruebas mencionadas en este apartado.

---

[53] La instrucción *SHL* necesita tres μops. Si se detecta que la actualización del registro de *flags* no se requiere, entonces uno de los μops será descartado. En este caso, dado que *DEC* no actualiza todos los *flags*, su actualización por parte de *SHL* no podía ser descartada.

[54] Recuérdese que el *front-end* lee de la caché líneas de 16 bytes para comenzar calculando el tamaño de las instrucciones y alimentar a los descodificadores.

[55] Éste no es en único escenario en el se da este tipo de retraso, pero es el que se ha usado para llevar a cabo la prueba.

7. Evitando la Detección

7.1. Introducción

Los ejemplos de detección mostrados hasta ahora se desarrollaron tratando el emulador seleccionado como una caja negra. Así pues, ninguno de los ejemplos se basa en aspectos específicos de la implementación del emulador. En su lugar, se eligieron diferentes aspectos de la microarquitectura que pueden tener un impacto visible en los tiempos de ejecución, se implementaron los ejemplos que intentan observar dicho impacto, y se compararon los resultados obtenidos en la ejecución real y en la emulación.

Una vez que se dispone de los ejemplos de detección, se puede desarrollar una prueba de concepto que permita evaluar la viabilidad de una emulación que evite las detecciones. Se pueden plantear dos posibles líneas de investigación. La primera de ellas se basa en las posibilidades ofrecidas por un sistema de emulación con traducción dinámica de código que lleve a cabo una traducción más exacta del código emulado. Esta línea de investigación se descarta porque requiere que el sistema emulado se corresponda exactamente con el sistema real o, en su defecto, un sistema de traducción dinámica de código considerablemente elaborado. La línea de investigación seguida en el trabajo se basa en crear un sistema de emulación que emule una microarquitectura, en vez de emular solamente la arquitectura del juego de instrucciones. Existen dos posibles líneas de trabajo. La primera, creando una nueva rama del emulador y modificándolo en la dirección expuesta, y la segunda, utilizando la interfaz proporcionada por el emulador para implementar la prueba de concepto. Se prosigue con la segunda línea de trabajo, puesto que la primera requiere un mayor esfuerzo sin aportar un beneficio notable a la hora de implementar una prueba de concepto[56].

Un estudio inicial del emulador, llevando a cabo una revisión sucinta del código, revelan dos aspectos negativos que tienen un impacto en el coste de la implementación. El primero de ellos es que el emulador no implementa la emulación de los contadores de rendimiento, ni la instrucción *RDPMC* que permite su lectura. El segundo aspecto negativo es que el emulador no lleva a cabo una emulación real de la instrucción *RDTSC,* de forma que no cuenta con una representación interna del registro asociado a la instrucción, y emula la instrucción reportando información sobre el tiempo de ejecución de la emulación misma. Este detalle de implementación plantea algunos problemas:

- No existe una forma sencilla de incidir en la emulación de la instrucción *RDTSC*

- La instrumentalización del emulador tiene un impacto en la emulación.

El segundo problema es especialmente importante, puesto que se pueden observar cambios en la probabilidad de éxito de las rutinas de detección expuestas al utilizar la interfaz del emulador para incidir en el comportamiento de la instrucción *RDTSC.* El ejemplo más evidente es el ejemplo número 9, donde no se ha implementado ninguna lógica explícita en el fichero del emulador, pero pasa a reportar la detección de emulación menos de un 10% de las veces.

El emulador no ofrece de una interfaz adecuada para el desensamblado e interpretación de código binario. Esta funcionalidad es necesaria para la implementación de la prueba de concepto. Se ha optado por utilizar la librería udis86[57] para llevar a cabo las tareas de desensamblado.

Al compilar los ejemplos de detección dentro de estos nuevos ejecutables, se puede ver como las probabilidades de error varían. Esto se debe a que se ha minimizado el número de ejecuciones de cada test, intentando simplificar los ejemplos. La variación experimentada no invalida los ejemplos. Este escenario es similar a ejecutar los ejemplos de detección con

---

[56] Esto no sería cierto si el objetivo fuese implementar una solución definitiva. En este caso, el objetivo es analizar la viabilidad, no el desarrollo de software con calidad comercial.

[57] http://udis86.sourceforge.net

diferentes cargas de CPU. Se puede observar como las probabilidades de error varían. Las modificaciones necesarias para mejorar las probabilidades de los ejemplos no dependen del concepto explotado en el ejemplo, sino en aspectos cuantitativos como el número de pruebas efectuadas, el tamaño de dichas pruebas, etc. Otro aspecto importante a tener en cuenta es que esta variación se da en la ejecución real (detección errónea), siendo el comportamiento de la emulación mucho más estable. Por lo tanto, no es necesario rediseñar o mejorar los ejemplos de detección.

7.2. Organización

El formato y organización del trabajo es parecida al utilizado en el trabajo con los ejemplos de detección. Se generan un total de diez ficheros ejecutables, que ejecutan y emulan las funciones de detección, reportando los resultados en ambos casos. El formato de salida es similar al mostrado en los ejemplos de detección. Cada uno de los ejecutables se relaciona directamente con uno de los ejemplos de detección expuesto, compilando exactamente ella misma función de detección.

Todos los ejemplos hacen uso de la misma implementación para llevar a cabo la emulación. La función emulación común a todos ellos se declara en un fichero *emulator.h* y se implementa en un fichero *emulator.c.* Ambos ficheros se almacenan en la carpeta *emulator*. El fichero *emulator.c* contiene la prueba de concepto expuesta. El resto de apartados se dedican a exponer su contenido.

También, al igual que en el caso de los ejemplos de detección, los ficheros *Makefile* que acompañan el código fuente soportan MacOs 64 bits y Linux 64 bits, habiendo sido probados en Debian 9.6.

7.3. Ajuste del tiempo de ejecución

El emulador proporciona varios mecanismos para interactuar con el código emulado. La prueba de concepto aprovecha dos de ellos.

El primero es la posibilidad de recibir una notificación antes de la ejecución de cada instrucción. Cuando se recibe dicha notificación, se indica la dirección de memoria y el tamaño de la instrucción. El segundo es la posibilidad de recibir una notificación cuando se produce un acceso a memoria. Cuando se recibe dicha notificación, se indica la dirección de memoria a la que se accede, así como el número de bytes a los que se accede y el tipo de acceso, que puede ser lectura o escritura.

```
if(!macro)
{
    ctrl->cycles_extra += PENALTY;
}
```

Fig 20: Actualización de los ciclos penalizados

Estas notificaciones son utilizadas para implementar una emulación muy rudimentaria de la microarquitectura. Cada vez que se detecta un escenario donde una optimización no se puede aplicar, incrementa un contador de penalización. Este contador nos indica en todo momento el número de ciclos extra que se deben añadir a los ciclos calculados por el emulador para reflejar los efectos de la emulación de la microarquitectura implementada en la prueba de concepto. La figura número 20 muestra una de las actualizaciones de dicho contador.

Como ya se ha mencionado, el tiempo de ejecución reportado por el emulador depende del tiempo de emulación. A su vez, dicho tiempo depende de la complejidad de la emulación de la microarquitectura. Por lo tanto,

```
/* penalties */
#define PENALTY             2048
#define CACHE_MISS_PENALTY  25000
```

Fig 21: Penalizaciones definidas en la prueba de concepto

el número de ciclos extra a sumar en cada escenario varía con la implementación de la prueba de concepto. Se ha decidido utilizar una solución simplista en este punto, de forma que la penalización es una constante suficientemente grande. Se ha diferenciado entre dos escenarios. Las penalizaciones relativas a los fallos de caché, y el resto de penalizaciones. La figura 21 muestra los valores usados en ambos escenarios. Es interesante notar que las penalizaciones presentes en la prueba de concepto funcionan en los dos entornos reales, pero no en la máquina virtual con la que se ha trabajado. Dentro de la máquina virtual los tiempos para las penalizaciones tendrían que aumentar considerablemente. Por ejemplo, la penalización estándar, situada en 2048 en el ejemplo, debe situarse en torno a 10000 para la máquina virtual. Es también interesante hacer notar que las estadísticas de algunas de las detecciones varían al probarse dentro de la máquina virtual, hasta el punto de no resultar útiles. De la misma forma, en el sistema virtualizado la prueba de concepto implementada falla al intentar evitar la detección en el ejemplo 5. No se ha analizado en detalle este comportamiento.

7.4. La instrucción *RDTSC*

En una máquina real, la instrucción *RDTSC* lee el contenido de un registro que es incrementado con caca ciclo del procesador. Este registro está dentro del grupo de registros llamado registros específicos del modelo[58], y a los que se suele acceder a través de instrucciones especiales. Dado que la interfaz del emulador permite leer y escribir este tipo de registros, la idea inicial era utilizarla interfaz del emulador para ajustar el contenido del registro. Como se ha comentado, el emulador no implementa una emulación real de este aspecto del procesador por lo que ha sido necesario buscar una alternativa.

La solución adoptada utiliza la notificación enviada antes de la ejecución de cada instrucción para detectar la ejecución de la instrucción *RDTSC,* de forma que en el siguiente ciclo se ajustan los efectos de la emulación sumando la penalización acumulada hasta el momento. La figura 22 muestra la implementación de la actualización de los efectos de la instrucción.

```
// if previous instrucion was rdtsc, then adjsut timing
if(ctrl->after_rdtsc)
{
    uint32_t eax, edx;
    uc_reg_read(uc, UC_X86_REG_EAX, &eax);
    uc_reg_read(uc, UC_X86_REG_EDX, &edx);

    uint64_t timestamp = ((uint64_t)edx << 32) + eax;
    timestamp += ctrl->cycles_extra;

    eax = (uint32_t)timestamp;
    edx = (uint32_t) (timestamp >> 32);

    uc_reg_write(uc, UC_X86_REG_EAX, &eax);
    uc_reg_write(uc, UC_X86_REG_EDX, &edx);

    ctrl->after_rdtsc = false;
}
```

Fig 22: Actualización de los efectos de *RDTSC*

7.5. Latencia y rendimiento de las instrucciones

El tiempo de ejecución de una instrucción depende de su latencia y su rendimiento, así como de sus dependencias. A la hora de modelar el tiempo de ejecución de una instrucción, el tiempo aportado por diferentes aspectos de la instrucción se pueden modelar de forma independiente. Por ejemplo, el tiempo de ejecución de una instrucción que accede a una posición de memoria se ve afectado por del tiempo de ejecución de la instrucción tanto como por el posible fallo de caché en el acceso a memoria. Estos dos aspectos del tiempo de ejecución son modelados de forma independiente.

En lo que concierne a la latencia y al rendimiento, el emulador seleccionado impone restricciones importantes a la implementación de una lógica adecuada. No es fácil controlar el tiempo de ejecución reportado por el emulador, por lo que implementar una gestión adecuada de la latencia

[58] MSR por sus siglas en inglés

y rendimiento de instrucciones individuales resulta difícil. Por lo tanto, la prueba de concepto solo implementa en este ámbito los detalles mínimos necesarios para evitar la detección de emulación llevada a cabo por los ejemplos presentados. Se añade una penalización a la instrucción *MFENCE* dado que sabemos que la latencia reportada por el emulador es incorrecta. Y se implementa una versión muy simplificada de control de la instrucción *MOV,* ignorando el ancho de banda del procesador y las dependencias de datos. La gestión del rendimiento de la instrucción *MOV* para las versiones de *MOV* que acceden a memoria puede verse en la figura 23. Como se puede apreciar, una lectura de memoria que se ejecuta cuando ya hay otra lectura en curso se solapa con la primera al usar el puerto de ejecución disponible, por lo que no incurre en una penalización de tiempo. Cuando *MOV* no accede a memoria, entonces la versión que no tiene latencia cero[59] es la que resulta penalizada.

```
/* handle latency and throughput for mov
   with memory access, assuming no data dependencies */
if(ctrl->ud_obj.operand[0].type != UD_OP_REG)
{
    // store operation
    if(ctrl->writes == 0)
    {
        ctrl->cycles_extra += PENALTY;
    }
    ctrl->writes = 2;
}
else
{
    // load operation
    if(ctrl->reads == 0)
    {
        ctrl->cycles_extra += PENALTY;
    }
    ctrl->reads = 2;
}
```

Fig 23: Control del rendimiento de MOV con acceso a memoria.

7.6. Emulación de la caché

La prueba de concepto emula una caché de datos estándar, de 32Kb, ocho vías, y con una línea de caché de 64 bytes. Se han llevado a cabo las siguientes simplificaciones:

- Solo se emula un nivel de caché, y no se emula la caché de instrucciones.

- No se emula el *prefetcher hardware*.

- Se emula un *prefetcher* software simplificado, y se da soporte parcial para las instrucciones no temporales.

La simplificación en las instrucciones no temporales consiste en asumir que no usan la caché. La gestión de la caché se lleva a cabo con cada acceso a memoria, a no ser que se haya detectado una instrucción no temporal. La figura 24 muestra la detección de las instrucciones no

```
case UD_Imovntdq:
case UD_Imovnti:
case UD_Imovntpd:
case UD_Imovntps:
case UD_Imovntq:
    ctrl->nonTemporal = true;
    break;
```

```
// cache
if(!ctrl->nonTemporal)
{
    if(!update_cache(ctrl, address))
    {
        // cache miss
        ctrl->cycles_extra += CACHE_MISS_PENALTY;
    }

    if(lineb != linee)
    {
        if(!update_cache(ctrl, address + size))
        {
            // cache miss
            ctrl->cycles_extra += CACHE_MISS_PENALTY;
        }
    }
}
```

Fig 24: Lógica para la actualización de la caché

[59] Cuando ambos argumentos son el mismo registro

temporales, y la actualización de la caché, cuidando el caso en el que un acceso no alineado toca dos páginas de caché diferentes.
En el caso del *prefetcher software,* cuando una instrucción de *prefetch* es detectada, se calcula la dirección efectiva, y la línea de caché que la contiene. Esta línea será cargada en la caché unos ciclos después de la ejecución de la instrucción. Si, tras ejecutar la instrucción *prefetch*, la dirección de memoria es accedida demasiado pronto, entonces se incurrirá en un fallo de caché. La implementación actual permite tener un total de 8 instrucciones *prefetch* esperado a ser completadas. Si se supera este límite, se incurrirá en una penalización. Las instrucciones esperando a ser completadas tienen un impacto en el tiempo que tarda una nueva instrucción *prefetch* en completarse. La nueva instrucción se completa tras la ejecución de 20 instrucciones, una vez que todas las instrucciones *prefetch* pendientes se han completado. Si se ejecuta una instrucción *prefetch* indicando una posición de memoria presente en la caché, la línea de caché se actualiza sin incurrir en ninguna penalización.

Aunque hay algunas simplificaciones importantes, y algunos de los valores utilizados en la prueba de concepto no se corresponden con parámetros reales del procesador[60], esta emulación de caché es razonable como demostración de viabilidad.

7.7. Emulación de reenvío de datos

El procesador es capaz de optimizar una lectura de memoria que sigue a una escritura en la misma dirección. La prueba de concepto implementada una simplificación de dicha funcionalidad, de forma que penaliza cualquier lectura de memoria que no esté precedida por una escritura a la misma posición. Esta comprobación se hace para todos los accesos a memoria. La emulación no tiene en cuenta que esta optimización también puede aplicarse en otros casos, como la lectura parcial de una escritura previa. Esta simplificación no conlleva una pérdida de generalidad, y es suficiente para cubrir los casos de uso utilizados en los ejemplos de detección. Este caso es especialmente complejo, puesto que añade una penalización siempre que la optimización no se puede aprovechar. En una aplicación estándar, el 40% de sus instrucciones son accesos a memoria [In16] y muy pocos de esos accesos aprovechan la optimización mencionada. Aunque la solución adoptada funciona, es claramente problemática dado que se podría implementar una detección de emulación basada en este nuevo comportamiento. En este caso, los problemas asociados a la solución vienen dados por las limitaciones de la interfaz con el emulador, por lo que no se considera que suponga una limitación para la solución propuesta.

```
// store forwarding
switch(type) {
    case UC_MEM_READ:
        // simplified version. forwarding happens if read matches previous write
        if(address != ctrl->last_write || size != ctrl->last_write_size)
        {
            // not forwarded
            ctrl->cycles_extra += PENALTY;
        }
        ctrl->last_write = 0;
        ctrl->last_write_size = 0;
        break;
    case UC_MEM_WRITE:
        ctrl->last_write = address;
        ctrl->last_write_size = size;
        break;
    default:
        break;
}
```

Fig 25: Envío de datos de escritura a lectura en memoria

7.8. Emulación de macro fusión

La prueba de concepto implementa una versión relativamente completa de emulación de la macro fusión. Para ello, el emulador siempre almacena la información sobre la última instrucción ejecutada. Cuando se detecta que la instrucción en curso es un salto condicional, se comprueba si el alineamiento permite la macro fusión, y en caso afirmativo se procede a analizar la instrucción anterior para decidir si se habilita la macro fusión. Si la instrucción anterior habilita la macro fusión, se comprueba en último lugar si la combinación de la instrucción anterior y el salto

[60] Por ejemplo, el tener un máximo de ocho instrucciones Prefetch completando en un momento dado.

condicional se pueden combinar en la macro fusión. En caso negativo, se añade una penalización al tiempo de ejecución del salto condicional. Al igual que en el caso anterior, esta implementación obedece a limitaciones en la interfaz del emulador.

La emulación implementada coincide con la documentada por Intel para la arquitectura Sandy Bridge.

```
else if(ctrl->ud_obj.mnemonic == UD_Iand
|| ctrl->ud_obj.mnemonic == UD_Iadd
|| ctrl->ud_obj.mnemonic == UD_Isub)
{
    // first operand always register?
    if(ctrl->ud_obj.operand[0].type == UD_OP_REG)
    {
        // rip based addressing not supported
        if(ctrl->ud_obj.operand[1].type == UD_OP_REG
        || ctrl->ud_obj.operand[1].type == UD_OP_IMM
        || (ctrl->ud_obj.operand[1].type == UD_OP_MEM && ctrl->ud_obj.operand[1].base != UD_R_RIP))
        {
            macro_possible = true;
        }
    }
}
```

Fig 26: Macro fusión

La figura 26 muestra como ejemplo el código que comprueba el control de habilitación de macro fusión para las instrucciones *ADD*, *SUB* y *AND*. En el caso de *SUB* y *ADD*, aunque habiliten la macro fusión, está no se efectuará si el salto condicional comprueba explícitamente el *flag* de desborde, signo, o paridad[61].

7.9. Emulación de la Micro fusión, y la delaminación

La micro fusión y la delaminación son, de alguna manera, dos caras de la misma moneda. Las dos afectan a instrucciones complejas, y en especial a instrucciones que llevan a cabo una operación cuando alguno de sus parámetros es un acceso a memoria. En algunos casos, cuando una instrucción es demasiado compleja, ésta se divide en dos μ*ops* diferentes. En otros casos, la instrucción se gestiona como una única instrucción en el *front-end*, pero se divide en dos μ*ops* diferentes para su ejecución. Por lo tanto, la diferencia entre una instrucción que habilita la micro fusión pero que es delaminada, es una mejora de rendimiento en el *front-end*. En la implementación propuesta, estos dos conceptos se fusionan, de forma que las instrucciones que tienen demasiadas dependencias de otros registros, y las instrucciones que usan el direccionamiento relativo a *RIP* se penalizan, independientemente de cual haya sido laoptimización concreta que la instrucción no ha podido aprovechar. En este caso, se considera que una instrucción tiene demasiadas dependencias de otros registros cuando referencia a tres o más registros. En el procesador real, no todas las instrucciones con direccionamiento relativo a *RIP* inhiben estas optimizaciones. Se ha optado por simplificar esta lógica, puesto que añadiría complejidad a la solución propuesta sin aportar un gran valor. La figura 27 muestra la implementación de la emulación de la micro fusión.

```
// microfusion and unlamination
static void handle_microfusion(instruction_control* ctrl)
{
    if(ctrl->ud_obj.operand[0].type != UD_NONE
    && ctrl->ud_obj.operand[1].type != UD_NONE
    && ctrl->ud_obj.operand[2].type == UD_NONE)
    {
        int regs_used = register_count(ctrl, 0) + register_count(ctrl, 1);
        if(regs_used >= 3)
        {
            // too many dependencies. not microfused, or unlaminated
            ctrl->cycles_extra += PENALTY;
        }
        else
        {
            // simplification: lets assume rip relative addressing does not fuse
            if((ctrl->ud_obj.operand[0].type == UD_OP_MEM  && ctrl->ud_obj.operand[0].base == UD_R_RIP)
             ||(ctrl->ud_obj.operand[1].type == UD_OP_MEM  && ctrl->ud_obj.operand[1].base == UD_R_RIP))
             {
                ctrl->cycles_extra += PENALTY;
             }
        }
    }
}
```

Fig 27: Implementación de la micro fusión

[61] Esto es, si es alguno de los siguientes saltos condicionales: *JO, JNO, JS, JNS, JP, JNP*

7.10. Emulación de la predicción de saltos

Aunque la predicción de saltos no se ha utilizado como mecanismo de detección de emulación, la prueba de concepto implementa una versión simplificada de una predicción de saltos dinámica, sin emulación de predicción estática. La predicción de saltos implementada tiene como principales limitaciones que solo predice los saltos condicionales, usa la dirección de memoria exacta en la que se encuentra la instrucción de salto, que el buffer de predicción de saltos almacena solo un máximo de ocho saltos[62], y que para los saltos condicionales de los que no se tiene información histórica, se asume que son tomados[63].

Desde el punto de vista de su implementación, se evita calcular la dirección efectiva del salto llevando el control de los saltos condicionales encontrados y cuál es la dirección de la siguiente instrucción a ser ejecutada en caso de no tomar el salto. En el siguiente ciclo se comprueba si el salto ha sido tomado o no, ajustándose el estado de la información histórica, y añadiendo una penalización en caso de que la predicción sea incorrecta.

```
// check branch prediction mismatch.
// problematic if jmp target is next instruction
if(ctrl->not_taken_address != 0)
{
    if(address == ctrl->not_taken_address)
    {
        // not taken
        if(ctrl->btb[ctrl->btb_index].taken >= 0)
        {
            ctrl->cycles_extra += PENALTY;
        }
        if(ctrl->btb[ctrl->btb_index].taken > -2)
        {
            --ctrl->btb[ctrl->btb_index].taken;
        }
    }
    else
    {
        // taken
        if(ctrl->btb[ctrl->btb_index].taken < 0)
        {
            ctrl->cycles_extra += PENALTY;
        }
        if(ctrl->btb[ctrl->btb_index].taken < 1)
        {
            ++ctrl->btb[ctrl->btb_index].taken;
        }
    }

    ctrl->not_taken_address = 0;
}
```

Fig 28: Control de predicción de saltos condicionales

La figura 28 muestra la comprobación del salto condicional, justo tras la ejecución de la instrucción. Los saltos condicionales tienen cuatro estados en la predicción. Tomado fuerte, tomado débil, no tomado débil y no tomado fuerte. En este caso, el campo *taken* es el encargado de gestionar el estado del salto condicional en el sistema de predicción. Los valores negativos se asocian a la predicción de no tomar el salto, siendo -2 el caso fuerte y -1 el caso débil. La predicción de salto tomado usa los valores 0 para débil y 1 para fuerte.

8. Conclusiones

Se ha proporcionado una descripción general de la microarquitectura de los procesadores x64, y se han proporcionado diversos mecanismos para la detección de emulación basados en los detalles expuestos. Se ha intentado que los diferentes ejemplos cubran los diferentes bloques lógicos y optimizaciones conocidas en la microarquitectura.

La conclusión, tras el análisis realizado, es que es factible el llevar a cabo una detección de emulación basada en el control de los tiempos de ejecución de código binario. Multitud de aspectos relativos a la microarquitectura del procesador son observables en las aplicaciones, y

---

[62] Todos los detalles mencionados son desviaciones respecto al comportamiento esperado por la predicción de saltos en la microarquitectura real. La más importante de todas ellas es el no implementar una predicción para otros tipos de saltos, como llamada a, y retorno de función, salto incondicional indirecto, etc.

[63] No se ha comprobado si este detalle está documentado, pero la asunción en la implementación concuerda con otras microarquitecturas, como se puede ver, por ejemplo, en [Fr09].

no parece que los emuladores tengan en cuenta la microarquitectura, sino que se centran en la implementación de la arquitectura del juego de instrucciones.

También es importante hacer notar que muchos de los ejemplos se podrían aplicar a otras arquitecturas. Si se toma como ejemplo un procesador totalmente diferente cómo puede ser el PowerPC e200, descrito en [Fr09], procesador utilizado normalmente en automoción, se puede apreciar que los ejemplos 1-3, 6, y 9 se podrían aplicar más o menos directamente, y aunque los ejemplos 4 y 5 son muy específicos para la microarquitectura de Intel, la idea subyacente puede ser usada para llevar a cabo una detección en este sistema.

Además, también parece factible el diseñar mecanismos de detección de emulación para los cuales no exista una solución sencilla.

Se ha implementado una emulación simplista del comportamiento de la microarquitectura. Aunque se han llevado a cabo diversas simplificaciones, ninguna de ellas conlleva una pérdida de generalidad. Desde el punto de vista de los resultados obtenidos, la emulación proporcionada invalida todos los ejemplos de detección expuestos.

Basándose en lo comentado en el párrafo anterior, se puede afirmar que un emulador centrado en la emulación de la microarquitectura puede evitar las detecciones basadas en tiempos. Existen, de todas formas, tres aspectos fundamentales que dificultan dicha implementación.

La primera dificultad es relativa a la información disponible sobre la microarquitectura. Revisando la documentación disponible en [In16], [In16-2] y [Fr09], se puede afirmar que la documentación es incompleta y poco clara. Una implementación basada en documentación oficial puede exponer comportamientos diferentes de la máquina real. Documentación no oficial como [Fo18] tampoco proporcionan el grado de detalle necesario. Por lo tanto, la obtención de la información necesaria para dicha implementación requeriría de unir la información documentada con pruebas sistemáticas y automatizadas que permitiesen completar la información necesaria.

Una segunda dificultad está relacionada con la pérdida de rendimiento del emulador. Los emuladores tienen, por lo general, un rendimiento muy por debajo de la ejecución real o de la virtualización. Un emulador que implementa la lógica asociada a la microarquitectura podría incurrir en penalizaciones de tiempo difíciles de asumir. Los ejemplos propuestos en este proyecto son demasiado pequeños, y la prueba de concepto propuesta es demasiado experimental como para poder hacer una estimación de la penalización asociada a la implementación propuesta. Es probable que el coste adicional de dicha emulación se pueda aceptar en entornos específicos, como por ejemplo en laboratorios de análisis de *malware*, pero no en un producto de seguridad instalado en equipos cuyo rendimiento global se vería afectado.

Por último, es importante no menospreciar el esfuerzo en términos de implementación y validación. Si tomamos como ejemplo un emulador como el usado como base para este proyecto, nos encontramos con que puede emular diversas arquitecturas. Una implementación basada en las microarquitecturas asociadas sería más costosa, añadiría problemas de diseño en caso de microarquitecturas muy dispares, añadiría más variaciones en el caso de querer soportar diversas microarquitecturas para un procesador concreto, y seguramente sería mucho más difícil de probar.

Los problemas expuestos dificultan la implementación propuesta, pero no la impiden. La disponibilidad de dicha implementación podría aumentar la eficacia en el análisis de malware, especialmente en los laboratorios de análisis. En caso de poder implementarse la solución propuesta con un rendimiento aceptable, la calidad y capacidad de los motores antivirus también se vería mejorada sustancialmente.

Cómo se puede apreciar en [So18], *malware* como *WannaCry* y *Cerber* representan un porcentaje muy importante de las infecciones a nivel mundial. Ambos usan diversas técnicas anti-emulación. Un emulador capaz de llevar a cabo una emulación más sofisticada y evitar la detección incidiría positivamente en los tiempos de análisis del *malware*, en el tiempo necesario para implementar desinfecciones, y mejoraría en general la seguridad de los sistemas informáticos.

9. Posibles direcciones futuras

Este trabajo se ha centrado en la detección de emulación basada en el análisis de tiempos de ejecución. También se ha centrado en la emulación del procesador, dejando de lado otros aspectos como los periféricos o el sistema operativo. Un estudio centrado en las técnicas de detección basadas en otros mecanismos, así como la detección de inconsistencias en la emulación del sistema operativo y/o periféricos completaría este trabajo y daría una visión general del estado y capacidad de la emulación en el contexto del análisis de *malware*, así como de su futuro.

Otra posible línea de trabajo se basa en avanzar en la línea de la prueba de concepto expuesta, contestando algunas preguntas abiertas importantes, como el rendimiento esperado de una implementación real emulando la microarquitectura del procesador, así como analizando su fiabilidad.

Una de las ideas que se bosquejan en este trabajo es que una implementación de un sistema de traducción dinámica generando una ejecución más cercana al código emulado podría ser una alternativa interesante para obtener una emulación más fiable. El presente trabajo no ha ahondado en esta idea, pero es creíble que esta línea de investigación pueda dar frutos interesantes.

Una de las premisas de este trabajo es que la emulación es preferible a la virtualización dado que la primera ofrece un control superior a la segunda. Plantear un sistema mixto, intentando obtener las ventajas de uno y otro también podría ser una línea de trabajo interesante. Dado que la desviación de la virtualización respecto de la ejecución real es mucho menor que en el caso de la emulación, se podrían obtener resultados interesantes en lo relativo a la fiabilidad de la emulación.

10. Referencias

[As17] B. Asvija, R. Eswari, and M. B. Bijoy, "Virtualization detection strategies and their outcome in public clouds," ed, 2017.

[Bl16] J. Blackthorne, A. Bulazel, A. Fasano, P. Biernat, and B. Yener, "AVLeak: Fingerprinting Antivirus Emulators Through Black-Box Testing," ed, 2016

[Eg09] M. Egele, et al, "A Survey on Automated Dynamic Malware Analysis Techniques and Tools," ed, 2009

[Fo18] A. Fog, "The microarchitecture of Intel, AMD and VIA CPUs," ed, 2018.

[Fr08] J. Franklin, M. Luk, J. McCune, A. Seshadri, A. Perring, and L. Dorn, "Towards sound detection of virtual machines," ed, 2008.

[Ga04] T. Garfinkel, J. Chow, B. Pfaff, K. Christopher, and M. Rosenblum, "Understanding Data Lifetime via Whole System Simulation", ed, 2004.

[Gy09] M. Gyung Kan, et all., "Emulating Emulation-Resistant Malware," ed, 2009.

[In16] Intel Corporation, "Intel 64 and IA-32 Architectures Optimization Reference Manual," ed, 2016.

[In16-2] Intel Corporations, "Intel 64 and IA32 Architectures Software Developer's Manual," ed, 2016.

[Ko18] P. e. a. Kocher, "Spectre Attacks: Exploiting Speculative Execution," ed, 2018.

[Li18] M. e. a. Lipp, "Meltdown: Reading Kernel Memory from User Space," ed, 2018.

[Na14] E. Nasi, "Bypass Antivirus Dynamic Analysis," in Limitations of the AV model and how to exploit them, ed, 2014.

[Po07] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, "Emulation-based Detection of Non-self-contained Polymorphic Shellcode," ed, 2007.

[Po74] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," Communications of the ACM, vol. 17, no. 7, pp. 412-421, 1974.

[Ra07] T. Raffetseder, C. Kruegel, and E. Kira, "Detecting System Emulators," ed, 2007.

[Ru05] J. Rutkowska, "Red Pill... or how to detect VMM using (almost) one CPU instruction," ed, 2005.

[Th10] C. Thompson, M. Huntley, and L. Chad, "Virtualization Detection: New Strategies and Their Effectiveness," ed, 2010.

[Wa07] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," ed, 2007.

[Ya16] Y. Yarom, "Microarchitectural Side-Channel Attacks," ed, 2016.

[fr09] Freescale Semiconductor, "e200Z4 Power Architecture Core Reference Manual," ed, 2009.

[So18] Sophos Antivirus, "Sophos Labs 2018 Malware Forecats," ed, 2018.

[Sy18] Symantec Corporation, "Executive Summary 2018 Internet Security Threat Report," ed, 2018

[Mc00] McGraw, G. Y Morriset, G., "Attacking Malicious Code: A Report to the Infosec Research Council," ed, 2000.

[Le13] Lee, Alan, Varadharajan, Vijay y Tupakula, Udaya, "On Malware Characterization and Attack Classification," ed, 2013

[Ti00] Tijms, Arjan, "Binary translation: Classification of emulators," ed, 2000

[Wa15] Wagner, M., Fischer, F., Luh, R., Haberson, A., Rind, A. Keim, D. A. Y Aigner, W., "A Survey of Visualization Systems for Malware Analysis," ed, 2015

[Sz01] Szor, P. Y Ferrie, P., “Hunting for Metamorphic,” ed, 2001

[Kr14] Kruegel, Christopher, “Full System Emulation: Achieving Successful Automated Dynamic Analysis of Evasive Malware,” ed, 2014

[St05] Stepan, Adrian E., “Defeating Polymorphism: Beyond Emulation,” ed, 2005