UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Instrumentación del emulador QEMU para análisis dinámico de software de sistema

Trabajo fin de grado

Grado en Ingeniería Informática

Autor: Horia Mihai David

Tutor: José Ismael Ripoll Ripoll

Curso 2019-2020

Resum

En els últims anys les tècniques d'anàlisi dinàmica com el *fuzzing* han demostrat ser maneres efectives de descobrir fallades crítiques en els sistemes programari moderns. La implementació d'estes tècniques requerix de mecanismes d'instrumentació, que permeten analitzar el comportamineto de programes així com modificar-los per a realitzar experiments.

El present treball exposarà les tècniques d'instrumentació de programari existents i farà un breu estudi de l'estat de l'art sobre la seua aplicació en el descobriment de fallades de seguretat en sistemes programari. Es recalcarà en la importància de QEMU per a la instrumentació de programari de sistema (SOTA, firmware…) en oposició a la instrumentació de processos comuns en un sistema operatiu.

A manera d'exercici pràctic, es modificarà el codi de l'emulador de sistema complet QEMU per a implementar una tècnica d'instrumentació del flux de codi i dels accessos a memòria, que servirà a manera de prova de concepte per a demostrar el tipus d'instrumentació requerit per a aplicar tècniques d'anàlisi dinàmica al programari de sistema.

Les modificacions realitzades s'explicaran detalladament, així com aquells detalls interns d'implementació d'una versió recent de QEMU que són necessaris per a la seua compressió. Estos detalls no estan documentats, i s'han obtenidomediante l'estudi del codi font de QEMU, perquè és un projecte de codi obert.

Paraules clau: Instrumentació dinàmica, QEMU, Emulació, Anàlisi de software

Resumen

En los últimos años las técnicas de análisis dinámico como el *fuzzing* han demostrado ser maneras efectivas de descubrir fallos críticos en los sistemas software modernos. La implementación de estas técnicas requiere de mecanismos de instrumentación, que permitan analizar el comportamineto de programas así como modificarlos para realizar experimentos.

El presente trabajo expondrá las técnicas de instrumentación de software existentes y hará un breve estudio del estado del arte sobre su aplicación en el descubrimiento de fallos de seguridad en sistemas software. Se hará incapié en la importancia de *QEMU* para la instrumentación de software de sistema (SO, firmware…) en oposición a la instrumentación de procesos comunes en un sistema operativo.

A modo de ejercicio práctico, se modificará el código del emulador de sistema completo *QEMU* para implementar una técnica *inline* de instrumentación del flujo de código y de los accesos a memoria, que servirá a modo de prueba de concepto para demostrar el tipo de instrumentación requerido para aplicar técnicas de análisis dinámico al software de sistema.

Las modificaciones realizadas se explicarán detalladamente, así como aquellos detalles internos de implementación de una versión reciente de *QEMU* que son necesarios para su compresión. Estos detalles no están documentados, y se han obtenido mediante el estudio del código fuente de *QEMU*, pues es un proyecto de código abierto.

Palabras clave: Instrumentación Dinámica, QEMU, Emulación, Análisis de software

Abstract

In recent years, dynamic program analysis techniques like fuzzing have proven to be effective for discovering critical security flaws in modern software systems. The implementation of these techniques require software instrumentation mechanisms for the analysis and modification of software behaviour.

This paper outlines current software instrumentation techniques and performs a brief state of the art analysis of their application in the discovery of security-critical software flaws. It is also the aim of this paper to highlight the importance of the *QEMU* emulator in the instrumentation of system software (OS, firmware...), as opposed to that of common prcesses running on top of an operating system.

As a practical exercise, the *QEMU* full system emulator is patched in order to implement inline instrumentation of the execution flow and memory accesses. This serves as a proof of concept for how to implement the instrumentation required for full-system dynamic binary analysis.

The modifications performed to a recent verstion of the *QEMU* codebase (v 4.2.0) are explained in detail, together with the required background knowledge about *QEMU* internal implementation details, most of which are undocumented.

Key words: Dynamic Binary Analysis, Instrumentation, QEMU, Program Analysis

Índice general

Índice de figuras

CAPÍTULO 1
Introducción

Con el auge de la computación en la nube, la virtualización forma ya parte de las pilas de software y hardware que usamos día a día. La virtualización permite abstraer los recursos de un computador físico (el anfitrión) y ofrecer la visión de que existen múltiples máquinas lógicas (los huéspedes), cada una ejecutando su propio sistema operativo.

En este punto es pertinente aclarar la diferencia entre virtualización y emulación. Cuando hablamos de virtualización, los huéspedes ejecutan su código máquina directamente en el procesador del anfitrión, por lo que huésped y anfitrión deben ser de la misma arquitectura. Es la tarea de un componente software llamado hipervisor la de actuar como una especie de sistema operativo maestro, para arbitrar el acceso a los recursos hardware del anfitrión y hacer pensar a cada huésped que se está ejecutando en su propia máquina física.

El hipervisor realiza esta tarea bien con ayuda del hardware especialmente diseñado para tal fin, o con una serie de modificaciones al software del sistema operativo de los huéspedes. En ambos casos intercepta los accesos al hardware realizados por los huéspedes para multiplexar el hardware físico.

Por otro lado, la emulación es una tecnología que pretende conseguir el mismo objetivo pero mediante un mecanismo distinto: implementar completamente en software el comportamiento de una máquina física, hasta tal punto que un sistema operativo moderno pueda ejecutarse pensando que lo hace sobre una máquina física.

La emulación tiene dos ventajas importantes. Primero, nos permite examinar por completo el comportamiento de la máquina emulada, pues todo lo que hace está implementado en software. Se puede modificar un emulador para realizar todo tipo de experimentos de análisis sobre un sistema completo.

La segunda ventaja es que podemos emular una máquina de una arquitectura en un anfitrión de otra arquitectura, muy útil para hacer desarrollo cruzado por ejemplo, en sistemas empotrados. Esto no es posible con la tecnología de virtualización pues esta multiplexa el hardware físico entre máquinas virtuales de la misma arquitectura que el anfitrión.

Sin embargo, como es común en ingeniería, no existen las soluciones, solo los compromisos. La emulación tiene un coste: bajas prestaciones. Emular un procesador en software es mucho más ineficiente que ejecutar de manera nativa código de un huésped.

Para paliar este efecto, los emuladores de sistemas modernos utilizan la misma técnica que las implementaciones de la máquina virtual Java: traducción dinámica o en inglés, “just-in-time compiling”. Esta técnica consiste en traducir un bloque de código máquina del huésped a la arquitectura del anfitrión cuando se necesite ejecutarlo por primera vez. Cada bloque traducido es insertado en una caché de código traducido, y es reutilizado las próximas veces, evitando la sobrecarga excesiva de sucesivas traducciones.

En este trabajo vamos a explorar el emulador *QEMU*, que implementa precisamente esta técnica y permite emular prácticamente la totalidad de los sistemas de computadores modernos dentro de un ordenador portátil común y corriente.

La razón por la que exploraremos el emulador no es para realizar desarrollo cruzado sino para instrumentarlo con el objetivo de hacer análisis dinámico del software de un sistema *x86_64* moderno. Al emular todo un sistema completo, se puede analizar dinámicamente el código de bajo nivel como la secuencia de arranque, el firmware del sistema o el sistema operativo.

1.1 Motivación

En el año 2016, la agencia *DARPA* del Departamento de Defensa de los Estados Unidos (la misma que dio luz a proyectos como el sistema GPS o el precursor de Internet: ARPANET) organizó la competición *Cyber Grand Challenge* (CGC).

En este evento, varios equipos integrados por investigadores de seguridad competían para desarrollar un sistema automática capaz de encontrar vulnerabilidades en un banco de prueba.

Este concurso, con premios millonarios, impulsó el interés de la comunidad científica por el desarrollo de técnicas de análisis de programas de carácter práctico y orientadas a la explotación de vulnerabilidades.

El desarrollo de estas herramientas de análisis requiere de técnicas de instrumentación, mecanismos por los cuáles se puede obtener información sobre el comportamiento de un sistema y modificarlo cuando sea necesario. *QEMU* resultó ser crucial en el desarrollo de estás herramientas de análisis dinámico como S2E, debido a la flexibilidad que ofrece al tratarse de un emulador y a que soporta arquitecturas modernas.

Más allá de de su importancia en el desarrollo de herramientas de análisis de seguridad, *QEMU*, junto con KVM[1] es la base de las pilas de software de virtualización en servidores y centros de datos, formando la parte fundamental de la infraestructura de la computación en la nube moderna.

Además, es una herramienta fundamental del flujo de desarrollo del núcleo de Linux, debido a su funcionalidad de depurar la máquina huésped. Es también la base del emulador de dispositivos móviles Android del SDK oficial para la plataforma.

QEMU es un proyecto de software muy grande, y que tiene un ciclo de desarrollo muy rápido, activo desde hace casi 20 años. *QEMU* se mantiene actualizado con las últimas características de casi todos los sistemas hardware modernos. Es quizás debido a esto que tiene una muy escasa documentación técnica sobre los detalles del funcionamiento interno, resultando muchas veces ser no oficial y tratando versiones antiguas. En este trabajo se documentará en castellano una pequeña parte del código interno de *QEMU*, el suficiente para entender las modificaciones de instrumentación que se realizarán. Muchos de los detalles que se verán no están oficialmente documentados maś allá del código fuente.

1.2 Objetivos

Los objetivos del presente trabajo son los siguientes:

1. Introducir las técnicas más comunes de instrumentación de software, haciendo hincapié en sus aplicaciones en el análisis de software encaminado a la seguridad informática.

[1] Tal y como veremos en sucesivos capítulos, *QEMU* no es solo un emulador, sino que combinado con la tecnología KVM es un potente hipervisor multiplataforma.

2. Realizar una introducción a la arquitectura y los modos de funcionamiento del emulador de sistema completo *QEMU* y su importancia en la industria.

3. Enumerar algunas herramientas existentes que implementan técnicas avanzadas de análisis dinámico e instrumentación de sistema completo o *Full System* con el objetivo de recalcar la importancia de *QEMU* en el desarrollo de estas, ya que *QEMU* es la base de las herramientas actuales de este tipo.

4. Documentar en suficiente detalle el funcionamiento interno de algunos de los componentes de *QEMU*, centrándonos en el flujo de ejecución de un CPU virtual y en el proceso de generación de código intermedio.

5. Detallar la implementación de un parche al código de *QEMU* para demostrar dos técnicas de instrumentación a nivel de sistema.

1.3 Estructura de la memoria

En el capítulo 2 se introducirá el concepto de instrumentación de software y se tratarán los diferentes tipos que existen. Se hará especial hincapié en las aplicaciones de estas técnicas a la seguridad, concretamente al análisis automático de software.

En el capítulo 3 se introducirá el emulador *QEMU*, su arquitectura y modos de funcionamiento y también detalles técnicos internos que son requisito para la comprensión del siguiente capítulo.

En el capítulo 4 se explicará en detalle una serie de cambios a realizar a la versión 4.2.0 de *QEMU* (publicada en diciembre de 2019) para implementar una prueba de concepto de dos técnicas de instrumentación de sistema completo.

Finalmente, en el capítulo 5 se presentarán las conclusiones finales del trabajo.

CAPÍTULO 2
Instrumentación de software

2.1 Introducción a la instrumentación de software

La instrumentación de software consiste en añadir instrucciones a un programa para realizar análisis de diversa índole que requieran de información en tiempo de ejecución. Los motivos por los que se hace, así cómo las técnicas para conseguirlo son varios.

La razón más común es quizás el monitoreo de las prestaciones y el consumo de recursos de un sistema. Por ejemplo, puede resultar útil saber cuánto tiempo pasa el procesador ejecutando diferentes funciones de un programa para detectar dónde puede estar un cuello de botella. También sería útil obtener, por ejemplo, información del consumo de memoria de un servidor web. Veremos además que tiene aplicaciones interesantes en seguridad informática.

Para conseguirlo se puede añadir código a un programa antes de o durante la compilación (instrumentación estática), o se puede añadir código al vuelo durante la ejecución de un programa. En las siguientes secciones se hablará de estas técnicas y algunos ejemplos y aplicaciones prácticas.

2.2 Instrumentación estática a nivel de código fuente

Añadir, por parte de un programador, líneas de código para por ejemplo trazar la ejecución o medir tiempos se suele considerar más una técnica de programación que un mecanismo de instrumentación.

Cuando hablamos de instrumentación estática a nivel de código fuente nos referimos a pases especiales de compilado que añaden funcionalidad de instrumentación, habitualmente para realizar un propósito específico.

Los compiladores desarrollados con la tecnología *LLVM* [3] están diseñados de manera modular para permitir añadir estos pases (*passes* in inglés) de compilado a un compilador existente sin demasiado esfuerzo (comparado con un compilador tradicional, más monolítico).

Un ejemplo muy notable de este tipo de instrumentación son los *sanitizers* [23] de Google. Son pases de LLVM para el compilador CLang[1] . Estos pases añaden código necesario en al programa que hace que se detecte en tiempo de ejecución diversas condiciones de error y genera un reporte con información de depurado.

El más notable es el pase de `AddressSanitizer`[4], que instrumenta las reservas y accesos de memoria tanto en la pila como en el *heap* y detecta un amplio abanico de errores de memoria.

[1] Compilador basado en LLVM para los lenguajes C, C++, Objective C/C++, OpenCL, CUDA y RenderScript

2.3 Instrumentación estática a nivel binario

Consiste en procesar un archivo ejecutable ya compilado y enlazado para extenderlo añadiéndole las instrucciones necesaria. Es una técnica relativamente poco común, pero la menciono por completitud.

2.4 Instrumentación dinámica

Conocida por su término en ingles *Dynamic Binary Instrumentation*, consiste en modificar el comportamiento de un programa mientras se está ejecutando.

Se pueden clasificar en dos grupos, aquellas herramientas que instrumentan un proceso dentro de un sistema operativo, y aquellas que sirven para instrumentar un sistema completo, incluyendo el SO y otro software de sistema.

Tanto las herramientas de instrumentación de procesos y de sistemas completos suelen basarse en técnicas similares a las de un compilador *Just-In-Time*. La técnica general consiste en no parchear el código en la memoria directamente, sino en copiar un bloque de código[2] según se necesite ejecutar a una *caché* de código, y en esta caché inyectar instrucciones de instrumentación. El programa original ejecutará este nuevo código en lugar del original.

Algunas de las herramientas de este tipo traducen el código máquina del programa a instrumentar a un código máquina intermedio, lo que permite mayor flexibilidad en el análisis o analizar código de otras arquitecturas a la máquina del analista.

2.4.1. Instrumentación dinámica de un proceso

Estas tecnologías habitualmente ofrecen una API que permite la programación de módulos con experimentos dinámicos.

Para ilustrar estas herramientas, veamos un ejemplo. En el listado siguiente podemos observar el código de un módulo de ejemplo para la herramienta *Pin* [2] de *Intel*.

```
FILE * trace;
// Print a memory write record
VOID RecordMemWrite(VOID * ip, VOID * addr, UINT32 size) {
    fprintf(trace,"%p: W %p %d\n", ip, addr, size);
}
// Called for every instruction
VOID Instruction(INS ins, VOID *v) {
    // instruments writes using a predicated call,
    // i.e. the call happens iff the store is
    // actually executed
    if (INS_IsMemoryWrite(ins))
        INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, AFUNPTR(RecordMemWrite),
            IARG_INST_PTR, IARG_MEMORYWRITE_EA,
            IARG_MEMORYWRITE_SIZE, IARG_END);
}
int main(int argc, char *argv[]) {
    PIN_Init(argc, argv);
    trace = fopen("atrace.out", "w");
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_StartProgram(); // Never returns
    return 0;
}
```

[2] En teoría de compiladores, un bloque básico de código es una secuencia lineal de instrucciones a la que se entra solo por la primera instrucción, y de la que se sale únicamente por la última instrucción.

Listing 2.1: Módulo de instrumentación de Pin para trazar los accesos a memoria. Fuente: publicación original de Intel [2]

Este módulo o *pintool* compila a una librería dinámica, que es inyectada junto con el motor de *Pin* en el proceso objetivo. Antes de copiarse una instrucción original de un bloque a la caché de código, se ejecutará `VOID Instruction(...)`, que comprueba si la instrucción es un acceso de memoria y en caso afirmativo inyecta una llamada a la función `VOID RecordMemWrite(...)` antes de dicha instrucción en la caché.

De esta manera, todos los accesos a memoria se ven precedidos por la llamada a `RecordMemWrite` , que registra el acceso en un archivo.

Existen otras herramientas similares pero de código abierto como DynamoRIO [24], Valgrind [25] o Frida [26]. Solo Valgrind utiliza una representación intermedia (VEX).

2.4.2. Instrumentación dinámica *Full System*

Las herramientas de análisis dinámico mencionadas hasta ahora permiten instrumentar procesos dentro de un sistema operativo. Resultaría muy útil poder aplicar las mismas técnicas a todo el software que ejecuta un sistema, desde las primeras instrucciones de la secuencia de arranque.

Para ello se han desarrollado herramientas y proyectos con el fin de permitir realizar análisis dinámico a la pila completa de software de un ordenador, como por ejemplo *BAP* [14], que es una versión modificada de *QEMU*. Se hará referencia a más herramientas de análisis basadas en *QEMU* en la próxima sección.

La herramienta BAP (Carnegie Mellon University Binary Analysis Platform) fue utilizada por el equipo ganador del primer premio en el concurso CGC organizado por el DARPA y mencionado en el primer capítulo de este trabajo. El proyecto es patrocinado por varias empresas y organizaciones que incluyen el Departamento de Defensa de los Estados Unidos, el gobierno de Corea del Sur, Siemens y Boeing.

2.4.3. Unicorn Engine

Mención especial merece el proyecto *Unicorn Engine*[28]. Consiste en una librería que expone una API para emular multitud de CPUs existentes. Se trata de una versión modificada de *QEMU* a la que se le ha eliminado la emulación de todos los componentes del sistema excepto el procesador. Al sistema se le añade mecanismos flexibles de instrumentación y análisis dinámico programables que *QEMU* no soporta por defecto.

Unicorn Engine no es un proyecto que pretenda emular ningún sistema de manera completa, solamente emula el procesador y permite implementar por encima otras herramientas.

Esta librería ha sido utilizada por multitud de proyectos[27], debido a la potencia y flexibilidad que ofrece, y a que soporta multitud de arquitecturas. Todo ello gracias al traductor dinámico de *QEMU*.

Por ejemplo, el proyecto Qiling Framework utiliza la librería Unicorn Engine como base para emular diferentes entornos de ejecución, incluido el entorno de ejecución de los binarios en Windows, Linux y Mac, así como los drivers UEFI[3]. Sobre los programas emulados se pueden realizar multitud de experimentos de análisis dinámico útiles por ejemplo, para el análisis de malware.

[3]Estándar abierto que define el entorno de ejecución del firmware en sistemas modernos. Las aplicaciones o drivers UEFI se ejecutan como parte del firmware del sistema, antes que el SO.

2.5 Aplicaciones en seguridad informática

Los mecanismos de instrumentación de software sirven para implementar técnicas de análisis programa. Estas técnicas han demostrado resultar muy prometedoras para encontrar fallos de software de manera automática y es todavía un campo de investigación abierto en la comunidad académica. A continuación se expondrán brevemente las técnicas principales.

2.5.1. Fuzzing

El fuzzing consiste en ejecutar de forma repetida un programa con entradas (inputs) malformadas. El objetivo es provocar que el sistema falle. El análisis de los fallos revela errores de programación, que pueden convertirse en fallos de seguridad. Con el fuzzing se intenta explorar el máximo número de estados de programa posibles, para aumentar la posibilidad de encontrase con un estado de fallo.

La técnica más relevante de fuzzing en la actualidad implica instrumentar el programa objetivo para obtener una traza de las transferencias de control de flujo, es decir, de las transiciones entre bloques básicos de código (*edge coverage*). Con esta información se puede implementar un algoritmo genético que produzca mutaciones sobre una entrada, y en función de si su ejecución produce nueva información de cobertura (nuevas transiciones), promocionar dicha entrada mutada o penalizarla. Repitiendo el proceso se realiza una exploración heurística que pretende maximizar la cobertura de código, explorando así cuantos más estados de programa lo más rápido posible mejor.

El interés de la comunidad investigadora por esta técnica se despertó con la introducción de la herramienta *American Fuzzy Loop* (AFL) [30], que para realizar la instrumentación implementa un pase especial de compilado.

El proyecto OSS-Fuzz [31] de google utiliza AFL, libFuzzer[4] y hongfuzz [5] para hacer fuzzing en la nube de manera continua de numerosos proyectos de software libre. Hasta la fecha de junio de 2020, encontró más de 20,000 bugs en 300 proyectos de software libre incluido el navegador Chrome, muchos de los cuales resultan ser agujeros de seguridad. Google utiliza los *sanitizers* mencionados en secciones anteriores para poder obtener información relevante sobre el motivo de un fallo inducido por un fuzzer. Esta herramienta es clave para poder reproducir y analizar los fallos.

Si se desea emplear estas técnicas sobre código de sistema se puede recurrir a instrumentar *QEMU*. Para el fuzzing de sistemas operativo ha resultado ser mucho más eficiente instrumentar *QEMU* operando como hipervisor junto con KVM y usar la tecnología Intel PT [6] para obtener información de cobertura [5].

2.5.2. Dynamic Taint Tracking

También conocida como *Taint Analysis* consiste en "seguirle la pista" a ciertos bytes de interés según son movidos por la memoria. Se implementa instrumentando un programa para interceptar las instrucciones de transferencia de datos. Según la memoria marcada como "manchada" es movida, el destinatario de ese movimiento es marcado a su vez como "manchado".

[4] Similar a AFL pero requiere de la programación de pequeñas funciones "target" dentro del código fuente del programa objetivo. Sobre estas funciones se realiza el fuzzing guiado por cobertura, en lugar de sobre un proceso entero. Forma parte del proyecto LLVM.

[5] Fuzzer guiado por cobertura de código abierto hecho por Google. Similar a AFL.

[6] *Intel Processor Trace* es una tecnología hardware implementada en los procesadores Intel que permite obtener información sobre las transferencias de control de flujo en tiempo de ejecución con una sobrecarga de tan solo 5-10 %

En el siguiente artículo [33] del blog del creador de la herramienta de análisis Triton se detalla una implementación de una prueba de concepto de esta técnica usando la herramienta de instrumentación Pin, mencionada anteriormente en este trabajo. Se demuestra también como usar esta técnica para detectar en tiempo de ejecución si el programa incurre en un error de memoria de tipo *"Use-after-free"* [7] el cual suele resultar ser una vulnerabilidad explotable.

Esta técnica permite implementar sistemas que detectan condiciones explotables y resulta especialmente útil cuando se combina con la ejecución concólica para conseguir mayor cobertura de código [8].

Esta técnica se ha empleado satisfactoriamente para mejorar la efectividad del fuzzing guiado por cobertura, el cual, si no se modifica el código fuente del programa a instrumentar, sufre la limitación de que es poco probable que se genere una mutación que satisfaga por ejemplo, una comparación de cadenas o de palabras de memoria. Hacer *Taint Tracking* del *input* de un programa permite realizar las mutaciones de manera más localizada y efectiva [6] [7] y mejorar así la cobertura de código.

2.5.3. Ejecución simbólica y concólica

La ejecución simbólica es una técnica de verificación formal desarrollada en los años 70 y que consiste en simular la ejecución del programa tratando las variables como simbólicas. Las operaciones sobre las variables dan lugar a fórmulas matemáticas en lugar de valores concretos. Cuando se tiene que tomar una decisión de control de flujo se abren ramas de posibilidades. Cada nodo es un estado del programa y se le asocia las restricciones que las variables deben cumplir para llegarse a él ("*Path Constraints*"). Ver figura 2.1.

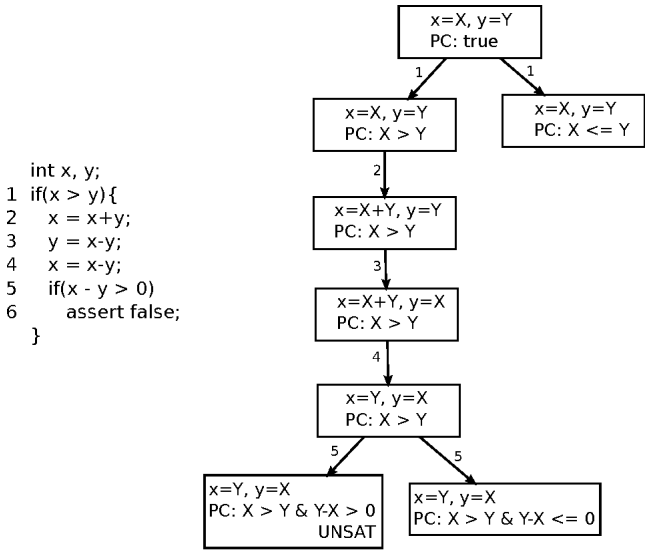

Figura 2.1: Ejecución simbólica. Fuente: [9].

Posteriormente se utiliza el árbol para razonar sobre propiedades del programa, o para generar casos de prueba automáticamente. Para generar casos de prueba se recogen las restricciones de los nodos hoja y se pasan por un *"solver SMT"* [8] que recibe las restricciones y genera valores concretos para cumplirlas o `UNSAT`, que significa que el modelo no es solucionable.

[7] Error de corrupción de memoria que consiste en que un bloque reservado dinámicamente (ej. con `malloc()`) es utilizado después de ser liberado (ej. con `free()`). Es un fallo de seguridad que puede llevar a la ejecución de código arbitrario. [15].

[8] Demostrador automático de teoremas muy usado en la industria para la verificación formal de circuitos integrados pero también de software. Recibe como entrada un conjunto de aserciones matemáticas sobre variables simbólicas y

Esta técnica tal y cómo se describe no es escalable a sistemas reales debido a la explosión de estados, a la dificultad de resolver restricciones no lineales y a la dificultad de interacción con el entorno de ejecución.

Una alternativa a la ejecución simbólica es la **ejecución concólica**. Esta técnica consiste en ejecutar el programa de forma normal, con valores concretos arbitrarios, y recoger una traza de las operaciones sobre las variables que se requieran tratar como simbólicas. Posteriormente se vuelve a ejecutar el programa de manera simbólica pero sin abrirse en anchura, se sigue el mismo camino que la traza concreta. Durante este proceso se recogen las restricciones a lo largo del camino y al final se niega la última restricción y se pasan todas al solver, para generar un nuevo input que explore un nuevo camino. De esta manera el motor simbólico se usa para guiar numerosas ejecuciones concretas que hacen una exploración en profundidad de los estados del programa.

Los frameworks de instrumentación y análisis dinámico Triton [35] y Angr [36] implementan esta técnica. El creador de Triton describe en su blog como implementar una prueba de concepto funcional de esta técnica [34]

Otra alternativa es ejecutar el sistema de manera concreta (normal) y pasar a ejecución simbólica solo en aquellos puntos de interés. Esta técnica, conocida como *"online symbolic execution"* o **"selective symbolic execution"** es la que implementa KLEE [10], una de las primeras herramientas de ejecución simbólica en ganar tracción en la industria y la comunidad investigadora. KLEE se implementa por encima de la infraestructura de compilado LLVM.

Mientras que KLEE permite hacer ejecución simbólica selectiva de programas comunes, el framework de análisis dinámico S2E [12] consiste en una versión de *QEMU* instrumentada para hacer ejecución simbólica selectiva del software de un sistema completo, incluido por ejemplo, el sistema operativo o la BIOS.

El sistema *Mayhem* [11] combina las dos técnicas anteriores mejorando la eficiencia. Este sistema basado en Pin realiza un análisis automático de binarios para encontrar fallos de seguridad.

Implementaciones similares se han realizado con otras herramientas que implementan estas técnicas, siendo interesante mencionar los resultados obtenidos con S2E debido a que puede trabajar con el código del sistema [37].

---

devuelve valores concretos que satisfagan el modelo, o indican que el modelo no se puede satisfacer. Utilizan aproximaciones heurísticas puesto que se trata de un problema NP-completo.

CAPÍTULO 3
El emulador *QEMU*

3.1 Introducción

QEMU [1] es un proyecto de software libre que permite tanto virtualizar como emular eficientemente multitud de plataformas hardware. Cuenta con numerosos desarrolladores activos que lo mantiene al día con las últimas características del hardware moderno. Debido a ello se ha establecido como estándar industrial de virtualización en entorno de servidores y como herramienta de depurado en la programación de software de sistema.

3.2 Modos de operación

QEMU consta de dos modos principales de operación [16]:

User mode emulation En este modo *QEMU* permite ejecutar aplicaciones de usuario compiladas para una arquitectura dentro otra, siempre y cuando huésped y anfitrión compartan el mismo sistema operativo. Lo hace emulando la CPU y traduciendo llamadas al sistema realizadas por la máquina huésped a llamadas al sistema del anfitrión. Consigue así emular el ABI[1] sin necesidad de emular todo el hardware del sistema. Se usa para facilitar la compilación cruzada y el depurado cruzado.[2]

Full system emulation En este modo *QEMU* emula no solo el CPU del huésped sino todo un sistema hardware completo, incluidos diversos periféricos hardware (controlador de interrupciones, timer...). Permite virtualizar un sistema operativo completo con todas sus aplicaciones.

3.3 *Full-system emulation*

La forma de implementar *Full system emulation* puede ser utilizando la virtualización acelerada por hardware o mediante la traducción dinámica.

Si se utiliza la virtualización acelerada por hardware *QEMU* funciona como un hipervisor, mientras que si se utiliza la traducción dinámica, la cual será explicada más adelante, *QEMU* funciona como un emulador.

[1] Application Binary Interface [17]: Entorno de ejecución de un proceso en espacio de usuario.

[2] Es muy común que un compilador para una plataforma solo esté disponible para esa plataforma. Al tratarse del compilador de un programa en espacio de usuario, QEMU user mode nos permite ejecutar el compilador en otra plataforma, para hacer compilación cruzada sin necesidad de portar el compilador.

3.3.1. *QEMU* como hipervisor

QEMU puede trabajar como un hipervisor bare-metal cuando se habilita el modo KVM. En este caso, el procesador y algunos periféricos básicos son emulados usando la tecnología de virtualización acelerada por hardware como Intel VT-x o AMD SVM. Esta virtualización la hace el componente KVM del núcleo de linux.

El trabajo de *QEMU* es emular el resto de periféricos, firmware y BIOS para presentar un sistema hardware virtualizado completo al sistema operativo que se ejecutará en la máquina a emular. Es decir, KVM no es suficiente para virtualizar un sistema, se necesita que trabaje en tándem con *QEMU*. Cuando *QEMU* se utiliza junto con KVM, usando la órden `--enable-kvm`, se denomina *KVM hosting*.

El KVM hosting permite que el anfitrión o máquina virtual en este caso, se ejecute de forma nativa en el procesador del huesped, consiguiendo prestaciones muy similares a las de un sistema no virtualizado. Gracias a esta tecnología de virtualización acelerada por hardware es posible toda la infraestructura de la nube.

Esta tecnología requiere que el huesped soporte tecnologías de virtualización aceleradas por hardware. El *KVM hosting* no funciona cuando *QEMU* se ejecuta bajo el sistema Windows.

3.3.2. *QEMU* como emulador

Cuando *QEMU* funciona como un emulador, no se utilizan tecnologías de virtualización aceleradas por hardware sino que se implementa en software todo un sistema hardware a emular.

EL cuello de botella de la emulación de un sistema es la emulación del CPU, pues hace falta implementar un bucle que lea las instrucciones del anfitrión, las decodifice y emule su comportamiento.

Para hacer más eficiente la emulación del CPU *QEMU* utiliza la técnica de traducción dinámica o traducción binaria al vuelo, una técnica similar a la de los compiladores just-in-time.

La traducción dinámica consiste en traducir el código máquina del huésped a código máquina del anfitrión por bloques o “a trozos”, según se requiera su ejecución (de allí el término “al vuelo”). Estos bloques traducidos son cacheados en memoria para su reutilización, de manera que un bloque de código es traducido una sola vez (la primera vez que se intenta ejecutar) y ejecutado muchas veces ya traducido. Esto supone una mejora muy considerable comparado con los emuladores clásicos que implementan un bucle fetch-execute instrucción a instrucción. El componente que realiza esta traducción es un just-in-time compiler. QEMU lo llama *TCG* (Tiny code generation).

QEMU soporta el modo *Full system* tanto con *KVM hosting* como con TCG, mientras que *user-mode emulation* solo soporta TCG, pues KVM es una tecnología diseñada para emular un sistema completo.

Este trabajo se centra en el modo Full system emulation con *TCG* es decir, traducción dinámica. Por tanto, siempre que se haga referencia a algún detalle de la arquitectura o implementación de *QEMU* se referirá al modo “Full system emulation” asumiendo *TCG* y no KVM, salvo que se indique lo contrario. Además se asumirá la versión 4.2.0 de *QEMU*.

3.4 Arquitectura de QEMU

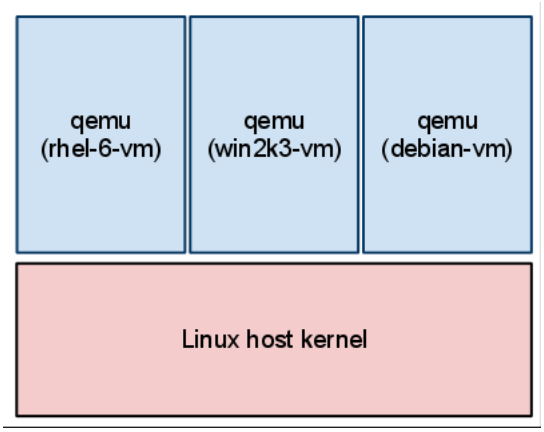

Figura 3.1: Procesos anfitriones *QEMU*. Fuente: [13]

.

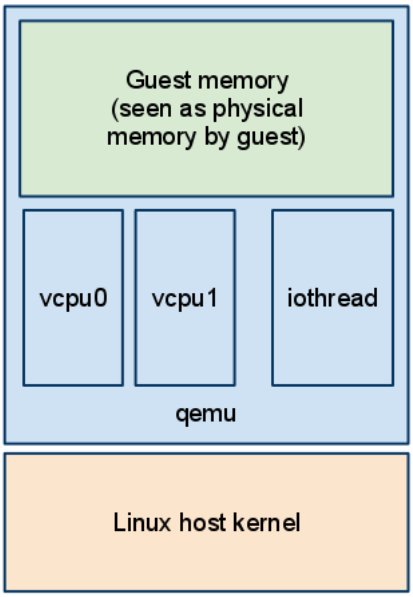

Figura 3.2: Arquitectura de un proceso anfitrión *QEMU*. Fuente: [13]

.

Una máquina virtual *QEMU* consta de un proceso en espacio de usuario que se ejecuta sobre el sistema operativo de la máquina anfitriona. Denominaremos a este proceso como proceso anfitrióno *host* en inglés. Naturalmente, se pueden ejecutar varios procesos*QEMU* independientes si se desea.

El proceso anfitrión consta de varios componentes, los más importantes son la emulación de los cores de procesador del anfitrión y la emulación del hardware. En este trabajo nos centramos especialmente en la emulación de los CPUs.

Esta emulación puede ser relegada a KVM o la puede realizar el propio *QEMU* usando un traductor dinámico que ya hemos introducido y que se llama TCG. Para emular los cores de procesado del anfitrión, en adelante vCPU, *QEMU* lanza un hilo del huesped por cada vCPU.

Cada uno de estos hilos, independiente de los demás, implementa la traducción dinámica del código que necesita ejecutar, y es visto por el sistema operativo del anfitrión como un core físico.

Además, la memoria física del anfitrión es implementada como una o varias regiones de memoria virtual en el proceso del huesped. Los hilos vCPU acceden a esta memoria cuando el código que ejecutan intentan acceder a la memoria física.

Los accesos a la memoria provocados por el huesped pasan por un componente software llamado *softMMU*, y es el encargado de traducir la dirección virtual del acceso a una dirección física emulado, y posteriormente traducir esa dirección física del huesped a una dirección dentro del rango de memoria del proceso anfitrión donde se encuentra la memoria física emulada.

Para implementar otras tareas necesarias para su funcionamiento, como el manejo de conexiones de red, accesos a disco o la interfaz gráfica en la que se puede ver la pantalla del anfitrión, *QEMU* utiliza un modelo orientado a eventos. Se abre uno o más hilos llamados *iothreads* que esperan en un bucle de eventos para resolver peticiones de otros componentes de *QEMU*. Es importante que estos hilos no queden bloqueados por lo que si se desea realizar trabajos computacionalmente intensivos se relega la carga a otros hilos de trabajo.

Esta arquitectura interna que hemos comentado en esta sección se puede ver ilustrada en la figura 3.2.

A continuación se tratará el traductor dinámico de *QEMU*. Este proceso de traducción es invocado por cada hilo vCPU y lo trataremos en detalle en este trabajo.

3.4.1. Arquitectura de Tiny Code Generator

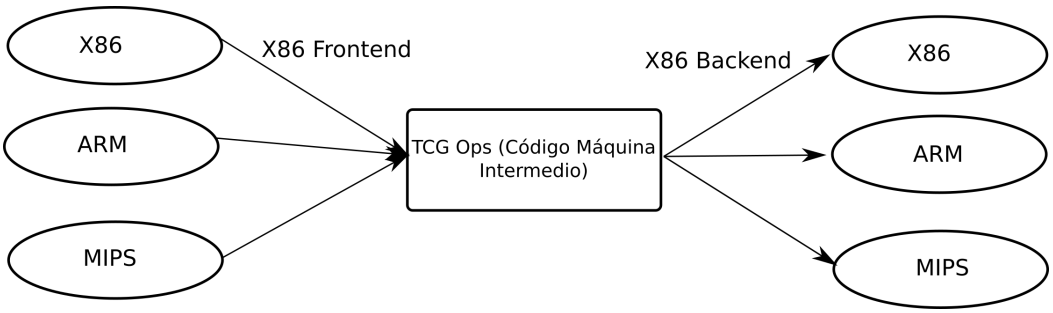

Figura 3.3: Arquitectura de TCG

Algunos traductores binarios traducen código máquina de la arquitectura fuente directamente a la arquitectura destino. Si *QEMU* funcionase así, sería necesario programar un nuevo compilador binario para cada par ordenado de arquitecturas (fuente, destino) que se quiere soportar. Esto no es factible.

El traductor binario de *QEMU* está diseñado para soportar la traducción de código entre múltiples arquitecturas de manera escalable

Para ello, el código de la arquitectura fuente es traducido primero a un código máquina intermedio llamado TCG ops o símplemente código TCG. Este código máquina no corresponde a ninguna arquitectura real. Posteriormente, el código intermedio no es interpretado directamente sino traducido a la arquitectura destino.

El módulo que traduce código máquina de una arquitectura al código intermedio se llama *FRONTEND*, y el módulo que traduce del código intermedio al código máquina de una arquitectura se denomina *BACKEND*.

Este diseño lo podemos apreciar en la figura 3.3. Si quisiéramos añadir soporte para la arquitectura PowerPC se necesita programar únicamente dos módulos de software, el *FRONTEND PowerPC* y el *BACKEND PowerPC*. Esto permitiría traducir código a cualquier otra arquitectura

soportada. Todo esto sin necesidad de programar un traductor binario para cada par de arquitecturas (X, PowerPC) y (PowerPC, X).

Ejemplo de bloque traducido

TCG traduce código bloque a bloque. Un bloque comienza con una instrucción a la que se salta, y termina con una instrucción que cambia el flujo lineal (un salto). También puede terminar en otras situaciones que requieran una salida del huésped al anfitrión, como una instrucción privilegiada, un punto de ruptura...

Si queremos ver el resultado del proceso de traducción, usamos el argumento de línea de órdenes `-d in_asm,op_opt,out_asm` para decirle a *QEMU* que imprima en la salida de error la siguiente información de depurado sobre cada bloque de código:

1. El código máquina del huésped que se va a traducir.
2. El código intermedio generado, después de los pases de optimización.
3. El código máquina que ejecutará el anfitrión, correspondiente al código del huésped ya traducido.

Veamos un ejemplo de un bloque de código. Es importante aclarar que el código máquina huésped y anfitrión son de la misma arquitectura, y que a lo largo del trabajo siempre se usará la arquitectura *x86_64* tanto para el huesped como para el anfitrión.

```
$ qemu-system-x86_64 -d in_asm,op_opt,out_asm 2> trace_tcg.txt
```

Al principio del archivo se encuentra el `PROLOGUE` o prólogo, un fragmento de código en ensamblador al que se transfiere el control para entrar a ejecutar un bloque traducido. Sólo hay un prólogo en todo el sistema. Después de este fragmento parecen los bloques de código propiamente dichos.

Un fragmento del archivo, correspondiente a uno de los bloques de código sería:

```
IN:
0x000fcfd6:  b9 10 00 00 00           movl     $0x10, %ecx
0x000fcfdb:  8e d9                    movl     %ecx, %ds

OP after optimization and liveness analysis:
 ld_i32 tmp11,env,$0xfffffffffffffff0     dead: 1  pref=0xffff
 movi_i32 tmp12,$0x0                      pref=0xffff
 brcond_i32 tmp11,tmp12,lt,$L0            dead: 0 1

 ---- 00000000000fcfd6 0000000000000000
 movi_i64 tmp0,$0x10                      pref=0xffff
 mov_i64 rcx,tmp0                         sync: 0  dead: 1  pref=0xffff

 ---- 00000000000fcfdb 0000000000000000
 mov_i32 tmp5,rcx                         dead: 1  pref=%rdx
 movi_i32 tmp11,$0x3                      pref=%rsi
 call load_seg,$0x0,$0,env,tmp11,tmp5     dead: 0 1 2
 movi_i64 tmp3,$0xfcfdd                   pref=0xffff
 st_i64 tmp3,env,$0x80                    dead: 0 1
 exit_tb $0x0
 set_label $L0
 exit_tb $0x7fea32000c03

OUT: [size=72]
0x7fea32000cc0:  8b 5d f0                movl     -0x10(%rbp), %ebx
0x7fea32000cc3:  85 db                   testl    %ebx, %ebx
```

```
0x7fea32000cc5:  0f 8c 29 00 00 00        jl      0x7fea32000cf4
0x7fea32000ccb:  bb 10 00 00 00           movl    $0x10, %ebx
0x7fea32000cd0:  48 89 5d 08              movq    %rbx, 8(%rbp)
<snip>
```

Listing 3.1: Desensamblado de un bloque de código.

Podemos observar que el bloque consta de dos instrucicónes del huésped. La primera se ha traducido a dos instrucciones intermedias TCG ops, la segunda a 8. Las tres primeras instrucciones son comunes a todos los bloques y se necesitan para el funcionamiento interno de las entradas y salidas al anfitrión. En la sección `OUT` aparecen las instrucciones nativas a las que saltará el hilo de la vCPU del huésped para emular las dos instrucciones originales del bloque.

Los números en hexadecimal en las líneas que comienzan con guiones son la dirección de memoria física emulada en la que se encuentra la instrucción del anfitrión a la que corresponde el código intermedio.

El código TCG es de tipo RISC y opera sobre un modelo de máquina virtual. Las instrucciones TCG ops operan sobre registros de esta máquina que se encuentran almacenados en memoria dentro de la estructura `TCGContext`. Estos registros se llaman *TCG variables*.

Algunas de estas *TCG variables* son globales y corresponden al estado de la máquina. Otras, como `tmp0` o `tmp1` sirven como registros temporales y se reservan y liberan dinámicamente durante el proceso de generación de código intermedio.

En el capítulo 4 se darán más detalles sobre como hacer uso de estas variables durante la generación de código.

Hay una instancia de esta estructura (`TCGContext`) por cada hilo de un vCPU. Se entrará en más detalles de la estructura cuando sea pertinente.

Ciertas instrucciones a emular serían muy costosas de hacer usando código intermedio. En tal caso, dicha funcionalidad se implementa como una función dentro del host. Esta función es invocada con la instrucción TCG `call`, que sirve para provocar una salida del host al anfitrión para invocar esas funciones llamadas *helper*.

Se hará uso de la generación de llamadas a helpers durante el capítulo 4 para añadir código de instrumentación en puntos clave.

3.4.2. Multi-threaded TCG

Como hemos mencionado, versiones recientes de *QEMU* implementan la ejecución del traductor dinámico de cada vCPU en su propio hilo. Versiones anteriores utilizaban un solo hilo junto con un planificador que multiplexaba en el tiempo la ejecución de los diversos vCPU en el mismo hilo.

Esto hacía más fácil la sincronización pero no se aprovechaban las prestaciones que podía otorgar un sistema multinúcleo.

La sincronización dentro de *QEMU* es una tarea complicada sin embargo, durante las modificaciones al proceso de traducción que implementamos en capítulos posteriores solo hacemos uso de estado local a cada hilo, por lo que no necesitamos de mecanismos de sincornización.

De todos modos, si no se le pasa a *QEMU* argumentos de la línea de comandos como `-smp <NUMBER>` el sistema solo cuenta con un núcleo, y por tanto con un hilo ejecutando el traductor dinámico.

Como no necesitamos sincronizarnos con otros hilos auxiliares de *QEMU* la sincronización no resultó ser un problema.

3.5 Entorno de trabajo

El estudio de *QEMU* sería una tarea excesivamente ardua solamente mediante la lectura del código fuente. Es necesario poder depurar el propio proceso anfitrión con un debugger para realizar tareas como poner un breakpoint o sacar una traza de la pila[3], y poder así estudiar el comportamiento en tiempo de ejecución del sistema.

Al ser *QEMU* un proceso en espacio de usuario, podemos depurarlo conectándole el debugger *gdb*. Además, nos resulta útil hacer uso de dos herramientas ofrecidas e implementadas dentro de *QEMU*: el monitor y el *gdbstub*.

El monitor[18] es una línea de órdenes interactiva ofrecida por qemu y que nos permide mandarle comandos como pausar o continuar la ejecución del huésped, así como obtener información de la máquina.

El módulo *gdbstub* expone una interfaz accesible por TCP que nos permite abrir una sesión de depurado remota con *gdb*. Esta sesión de depurado sirve para depurar la ejecución del huésped desde la primera instrucción que ejecuta la máquina. Es como si tuviéramos un debugger hardware [19] conectado a una máquina física real (por ejemplo, un debugger JTAG en un sistema empotrado). Este mecanismo se usa para depurar código del sistema, como el cargador de arranque GRUB o el núcleo de Linux [20].

En la figura 3.4 podemos observar la relación entre los componentes anteriormente mencionados. La sesión de depurado número 1 sirve para depurar el propio proceso de QEMU y a lo largo de este documento la interacción con esta sesión aparecerá precedida por la prompt `(gdb)`. La interacción con la sesión de depurado número dos aparecerá a lo largo del documento precedida por la prompt `(gdbstub)`, en caso de ser necesaria.

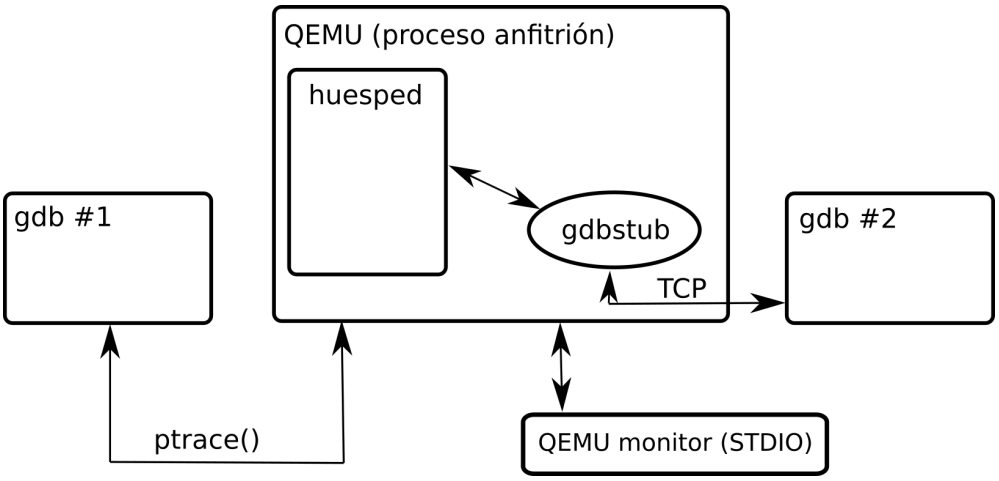

Figura 3.4: Entorno de depurado

Para arrancar un sistema como el de la figura solo necesitamos ejecutar una orden de terminal:

```
$ gdb --args qemu-system-x86_64 -s -S -monitor stdio
```

Este comando abre la sesión de depurado numero 1, con *QEMU* sin arrancar todavía. El flag `-monitor stdio` le indicará a *QEMU* que mientras se ejecuta, abra la consola del monitor en la terminal.

El flag `-s` le indicarán al módulo gdbstub que abra la sesión de depurado numero 2 en la figura en el puerto TCP 1234. La orden `-S` le indicarán a gdbstub que no permita que el huésped

[3] Muy útil para saber la secuancia de llamadas a funciones que nos han llevado hasta el punto de ejecución en el que se encuentra el programa

arranque hasta que reciba la orden `run`. De esta manera, se puede depurar el huésped desde la primera instrucción que ejecuta el procesador.

Desde la sesión 1 ejecutamos la orden `run` para permitir que arranque *QEMU*. Esto permitirá el establecimiento de la sesión 2, a la cual el huésped está esperando bloqueado.

```
(gdb) run
Starting program: qemu-system-x86_64 -s -S -monitor stdio
```

Ahora, desde otra terminal abrimos la sesión de depurado número dos:

```
$ gdb
(gdbstub) target remote localhost:1234
Remote debugging using localhost:1234
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x000000000000fff0 in ?? ()
(gdbstub)
```

El *warning* del debugger podemos ignorarlo, pues no contamos de información simbólica sobre el software de sistema que se va a ejecutar.

3.6 Bucle principal de un vCPU

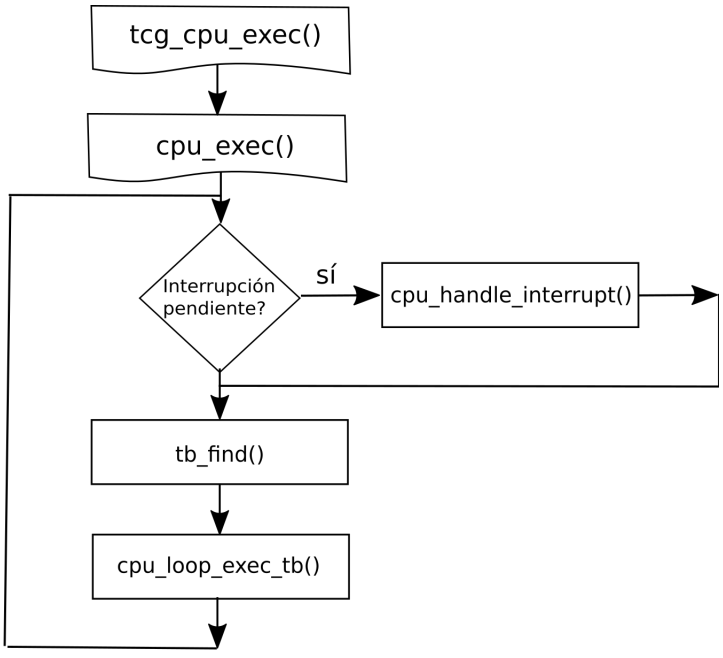

Figura 3.5: Bucle de ejecución principal de un vCPU

En esta sección se pretende exponer el flujo de código de un hilo que implementa un vCPU, es decir, un hilo que realiza la tradución y ejecución de código. El objetivo es situar al lector dentro del código de *QEMU* y así entrar en detalles más concretos en las sucesivas secciones.

Como se puede observar en la figura 3.5, el hilo de un vCPU entra en un bucle. En este bucle obtiene el próximo bloque de código a ejecutar (representado por la estructura `TranslationBlock`) con `tb_find()` y posteriormente lo ejecuta con `cpu_loop_exec_tb()`.

`tb_find()`

`tb_find()` busca el bloque a ejecutar en una caché interna, donde se almacenan los bloques ya traducidos para su reutilización. Si la búsqueda falla, se llama a `tb_gen_code()`, función encargada de generar un bloque a ejecutar. Esta función acabará invocando el proceso de traducción de código y será explicado en más detalle en la siguiente sección. Una vez generado el bloque, este es introducido en la caché anteriormente mencionada.

Aquí se implementa la técnica *Basic Block Chaining* que consiste en que el bloque de código que ha provocado la transferencia de control de flujo al bloque actual es parcheado para saltar directamente al código traducido del bloque actual, en lugar de tener que pasar otra vez por el bucle de ejecución.

La estructura `TranslationBlock`

Un bloque de código traducido y toda la información que le rodea se representa por la estructura `TranslationBlock`.

Entre otros campos incluye el valor del contador de programa de la máquina emulada a la que corresponde el bloque (`target_ulogn pc`) y el puntero al buffer de código ya traducido para ser ejecutado por el anfitrión (`tb_tc.ptr`).

La estructura `CPUState`

Como su nombre indica, representa el estado de un vCPU. Hay una instancia por cada procesador virtual. Contiene campos comunes a todas las arquitecturas y un puntero a una estructura con información específica de la arquitectura emulada (`env_ptr`). Puesto que estamos emulando un *x86_64*, este puntero apunta a un instancia de `CPUX86State`. Uno de sus campos es `regs`, que contiene el valor de los registros de propósito general (RAX, RBX, RCX...).

Cada hilo de un vCPU contiene una variable local[4] con el estado del procesador que repreeenta:

```
__thread CPUState *thread_cpu;
```

3.7 Traducción de un bloque de código

Recordamos que el proceso de traducción de un bloque ocurre cuando `tb_find()` invoca a `tb_gen_code()` debido a un fallo en la caché del traductor[5].

`tb_gen_code()` prepara la estructura `TranslationBlock` [6] e invoca a `gen_intermediate_code()`, que como su nombre indica, traduce las instrucciones de código máquina del anfitrión a código intermedio *TCG ops*.

Finalmente, invoca a `tcg_gen_code()`. Esta función realiza pasos de optimización sobre el código intermedio y genera el código máquina ejecutable por el anfitrión asociado al bloque.

No se entrarán en más detalles sobre la generación de código nativo a partir del código intermedio. Sin embargo, sí que explicaremos el proceso de generación de código intermedio, puesto que la implementación de algunas técnicas de instrumentación que se explicarán en sucesivos capítulos consisten en parchear *QEMU* para inyectar código intermedio en puntos clave.

[4] Se usa el mecanismo Thread Local Storage para obtener memoria local a cada hilo.
[5] A no confundir con la caché de la jerarquía de memoria en los computadores modernos
[6] Entre otras cosas le asigna los campos `pc` y `tc.ptr`

3.8 Traducción a código intermedio

`gen_intermediate_code()` no es más que un *wrapper* alrededor de `translator_loop()`, que se encarga de generar el código intermedio de un bloque.

La implementación de la generación de código requiere de cierta funcionalidad específica de la arquitectura emulada, por lo que `gen_intermediate_code()` le pasa a `translator_loop()` una estructura con punteros a funciones correspondientes.

```
static const TranslatorOps i386_tr_ops = {
    .init_disas_context = i386_tr_init_disas_context,
    .tb_start           = i386_tr_tb_start,
    .insn_start         = i386_tr_insn_start,
    .breakpoint_check   = i386_tr_breakpoint_check,
    .translate_insn     = i386_tr_translate_insn,
    .tb_stop            = i386_tr_tb_stop,
    .disas_log          = i386_tr_disas_log,
};

/* generate intermediate code for basic block 'tb'.  */
void gen_intermediate_code(CPUState *cpu, TranslationBlock *tb, int max_insns)
{
    DisasContext dc;

    translator_loop(&i386_tr_ops, &dc.base, cpu, tb, max_insns);
}
```

Listing 3.2: Fragmento de código de *QEMU*.

Cuando el sistema de compilado genera el binario `qemu-system-x86_64`, usa esta implementación de `gen_intermediate_code()`, y usará una similar pero con otra tabla de punteros para generar, por ejemplo, `qemu-system-arm`. De esta manera, `translator_loop()` es independiente de la arquitectura emulada. Cuando necesite de cierta funcionalidad específica de la arquitectura emulada, llamará a un puntero de dentro de la estructura, el cuál ya ha sido cargado con la función adecuada.

El objeto `DisasContext dc` será inicializado dentro de `translator_loop()` y se usará para desensamblar instrucciónes del huésped, operación necesaria para traducirlas a código intermedio.

3.8.1. Bucle de traducción `translator_loop()`

`translator_loop()` comienza inicializando el objeto del desensamblador:

```
/* Initialize DisasContext */
db->tb = tb;
db->pc_first = tb->pc;
db->pc_next = db->pc_first;
db->is_jmp = DISAS_NEXT;
```

Es suficiente comentar que lo que sigue es un bucle que en cada iteración llama a `ops->translate_insn(db, cpu)` para traducir cada instrucción del bloque. Si la máquina emulada es un *x86_64*, esta función sera `i386_tr_translate_insn()`.

Durante cada iteraciíon del bucle, `db->pc_next` apunta a la próxima instrucción dentro del huésped que se necesita traducir. Cuando `db->is_jmp` sea distinto de `DISAS_NEXT`, se saldrá del bucle y se emitirá el código intermedio necesario para salir de la ejecución de un bloque. Esto lo hace `gen_tb_end()`, que emite una instrucción *TCG ops* de tipo `exit_tb`.

Se puede salir del bucle de traducción porque se ha traducido una instrucción de salto, porque se ha llegado al máximo de instrucción a traducir o el buffer de salida está lleno, o porque se ha detectado la presencia de un *breakpoint* en una instrucción a traducir. Este último caso provoca que el bloque termine en ese punto, para que durante su ejecución, en ese mismo punto se transfiera de nuevo el control a *QEMU*, el cual podrá notificar al debugger conectado al *gdbstub*.

3.8.2. Traducción de una instrucción a código intermedio

Las instrucciones TCG ops se encuentran en una lista enlazada para una manipulación más flexible. La lista enlazada es el campo `ops` de la estructura `TCGContext`.

Esta estructura almacena también las "variables temporales", que son los registros sobre los que operan las instrucciones intermedias. Cada hilo de un vCPU tiene una instancia de esta estructura, pues necesita su propio contexto de ejecución. Cada hilo puede acceder a su instancia a través de la variable:

```
extern __thread TCGContext *tcg_ctx;
```

Las funciones de nombre `tcg_gen_*()` son las que generan una instrucción intermedia. Por ejemplo, para generar una instrucción `movi_i32` se utiliza `tcg_gen_movi_i32()`

Todas estas funciones acaban llamando directa o indirectamente a `tcg_emit_op()` que hace la emisión de la instrucción al añadirla a la lista enlazada:

```
TCGOp *tcg_emit_op(TCGOpcode opc)
{
    TCGOp *op = tcg_op_alloc(opc);
    QTAILQ_INSERT_TAIL(&tcg_ctx->ops, op, link);
    return op;
}
```

Si ponemos un punto de ruptura en `tcg_emit_op()` y sacamos una traza de la pila, podemos observer la secuencia de llamadas que ha llevado a `tcg_emit_op()`:

```
(gdb) break tcg_emit_op
Breakpoint 1 at 0x2fd464: file /qemu-4.2.0/tcg/tcg.c, line 2343.
(gdb) run
Starting program: /qemu-4.2.0/bin/debug/native/x86_64-softmmu/qemu-system-
    x86_64 debian_wheezy_amd64_standard.qcow2

(gdb) bt
#0  0x0000555555851464 in tcg_emit_op (opc=INDEX_op_discard) at /qemu-4.2.0/
    tcg/tcg.c:2343
#1  0x000055555585d131 in tcg_gen_op3 (opc=INDEX_op_ld_i32, a1
    =140736683055408, a2=140736683052776, a3=18446744073709551600) at /qemu
    -4.2.0/tcg/tcg-op.c:60
#2  0x00005555558e03fe in tcg_gen_ldst_op_i32 (opc=INDEX_op_ld_i32, val=0xdd0,
     base=0x388, offset=18446744073709551600) at /qemu-4.2.0/tcg/tcg-op.h:112
#3  0x00005555558e0490 in tcg_gen_ld_i32 (ret=0xdd0, arg2=0x388, offset=-16)
    at /qemu-4.2.0/tcg/tcg-op.h:384
#4  0x00005555558e0668 in gen_tb_start (tb=0x7fffd9bdd040 <code_gen_buffer
    +19>) at /qemu-4.2.0/include/exec/gen-icount.h:46
#5  0x00005555558e08f0 in translator_loop (ops=0x555556422f20 <i386_tr_ops>,
    db=0x7fffd9bd7430, cpu=0x555556af12d0, tb=0x7fffd9bdd040 <code_gen_buffer
    +19>, max_insns=512)
    at /qemu-4.2.0/accel/tcg/translator.c:56
#6  0x00005555559fc417 in gen_intermediate_code (cpu=0x555556af12d0, tb=0
    x7fffd9bdd040 <code_gen_buffer+19>, max_insns=512) at /qemu-4.2.0/target/
    i386/translate.c:8619
#7  0x00005555558dedd8 in tb_gen_code (cpu=0x555556af12d0, pc=4294967280,
    cs_base=4294901760, flags=64, cflags=-16252928) at /qemu-4.2.0/accel/tcg/
    translate-all.c:1734
```

```
#8  0x00005555558db57a in tb_find (cpu=0x555556af12d0, last_tb=0x0, tb_exit=0,
    cf_mask=524288) at /qemu-4.2.0/accel/tcg/cpu-exec.c:406
#9  0x00005555558dbf3d in cpu_exec (cpu=0x555556af12d0) at /qemu-4.2.0/accel/
    tcg/cpu-exec.c:730
#10 0x0000555555892d3b in tcg_cpu_exec (cpu=0x555556af12d0) at /qemu-4.2.0/
    cpus.c:1473
#11 0x00005555558935a9 in qemu_tcg_cpu_thread_fn (arg=0x555556af12d0) at /qemu
    -4.2.0/cpus.c:1781
```

Si continuamos el experimento (**`continue`**, `bt`), veremos para cada instrucción TCG generada, el tipo de instrucción que es, pues podemos ver si por ejemplo se ha ejecutado `tcg_gen_ld_i32()`,`tcg_gen_movi_i32()`...

Como vemos, la invocación viene de `translator_loop()` y en este caso es una de las instrucciones que forman parte del comienzo de cada bloque (`gen_tb_start()`).

Durante el capítulo 4 se comentarán más detalles del traductor dinámico. Aquellos que son necesarios para la realización de la instrumentación propuesta.

CAPÍTULO 4
Instrumentación de *QEMU*

4.1 Introducción

Apoyándonos en los detalles de implementación de *QEMU* que han sido documentados en el capítulo anterior, en este capítulo trataremos el tema de la instrumentación del emulador.

Primero se realiza una discusió sobre los mecanismos que ya ofrece *QEMU* para analizar el comportamiento del sistema emulado.

Posteriormente se detalla con exactitud las modificaciones que se realizan al emulador para conseguir dos objetivos:

1. Interceptar el flujo de código de insturcciones que se encuentran en un rango concreto.
2. Interceptar los accesos a memoria provocado por instrucciones en el mismo rango.

El objetivo no es resolver un problema de análisis de software concreto, sino detallar los detalles necesarios para implementar estas técnicas y que sirve de base para construir plataformas de análisis de código basadas en *QEMU* como aquellas nombradas en el capítulo anteior.

Nuestra instrumentación se basará en modificar el compilador dinámico que usa *QEMU* para inyectar código de instrumentación en puntos de interés. El código de instrumentación imprimirá trazas para demostrar su funcionamiento.

Las mismas técnicas se pueden aplicar para resolver problemas de análisis dinámico.

4.2 Mecanismos de instrumentación existentes en *QEMU*

Antes de empezar a modificar el código de *QEMU* veamos qué mecanismos ofrece ya para realizar instrumentación.

Ya hemos utilizado a lo largo del trabajo la opción `-d`, la cual nos permite decirle al emulador que genera en la salida de error una traza con información sobre determinados eventos. Solamente hemos utilizado `op_opt`, `in_asm` y `out_asm` para obtener un desensamblado de los bloques de código.

Existen más opciones como por ejemplo obtener una traza de las interrupciones o una traza de las traducciones de memoria virtual a memoria física. Para obtener una lista de las opciones posibles ejecute:

```
qemu-system-x86_64 -d H
```

Esta información puede resultar útil en proyectos de instrumentación para obtener información sobre el comportamiento del sistema. Sin embargo esta información es relativamente limitada. Existe un mecanismo dentro de *QEMU* similar pero con una granularidad más fina. El mecanismo de *tracing* [38].

Con este mecanismo *QEMU* nos efrece una lista de eventos, los cuales podemos activar. El emulador nos informará sobre estos eventos, bien en un archivo de texto, en la salida de error, o integrándose con sistemas como SystemTap que no se tratará en este trabajo.

A lo largo de todo el código de *QEMU* hay invocaciones a funciones que registran el suceso de un evento (`trace_*()`). Por defecto el sistema de compilado las ignora si no están activadas. La documentación oficial contiene escasa documentación sobre qué eventos están disponibles, pero se puede obtener un listado completo en archivos `.h` del código fuente del emulador.

Estos mecanismos pueden resultar útiles para proyectos de insturmentación en los que se requiera obtener informacion de profiling o trazas sobre el comportamiento de un sistema. También son útiles para depurar y estudiar el propio emulador, sin embargo, son mecanismos de trazado, no de instrumentación completa.

Si se desear instrumentar el código emulado usando una API similar a la de herramientas como Pin se puede recurrir a los plugins de *QEMU*, o a la modificación del código fuente del emulador, pues es un proyecto de código abierto.

4.2.1. *QEMU* Plugins

A partir de la versión 4.2 de *QEMU*, si se compila *QEMU* con la opción `--enable-plugins` cuando se hace el `configure` se habilitan los plugins. Este mecanismo es reciente y no está completado.

Los plugins son librerías dinámicas que son cargadas por el emulador si así se le indica en la línea de órdenes. Estos plugins, similar a los plugins de Pin que hemos visto en capítulos anteriores, pueden usar una API ofrecida por *QEMU* para, sin tener que preocuparse por los detalles internos del traductor dinámico, introducir código de instrumentación en el flujo de instrucciónes.

La API es muy similar a la de Pin. El punto de entrada de un plugin es llamado por *QEMU* durante la inicialización. En este punto de entrada (la función `qemu_plugin_install()`) se puede registrar una callback propio que es invocada por *QEMU* despues de traducir a código intermedio un bloque.

En este callback se itera sobre las instrucciones de un bloque y se decide si se desea inyectar o no una llamada a una función de instrumentación en mitad del bloque en los puntos deseados. Haciendo esto lo que se consigue es introducir un *call* en puntos deseados del código del sistema.

En la carpeta `qemu/tests/plugin` podemos ver varios ejemplos de estos plugins. La API ofrecida está documentada en el propio código, en `qemu/plugins/api.c` y `qemu/plugins/plugin.h`.

El plugin de ejemplo `bb.c` cuenta los bloques de codigo y el total de instrucciones. El plugin `mem.c` hace uso de la API para instrumentar solo aquellas instrucciones que son accesos a memoria y contarlas.

El mecanismo de plugin no ofrece la flexibilidad necesaria para implementar cualquier tipo de instrumentación, pero sí una API que abstrae los detalles de implementación del emulador. Puesto que el objetivo del trabajo es implementar desde cero la instrumentación, no se usarán los plugins.

4.3 Instrumentación del flujo de código

En esta sección detallaremos una técnica para conseguir instrumentar el flujo de ejecución de un anfitrión corriendo dentro de *QEMU*.

Para demostrar la técnica, nos ponemos como objetivo realizar las modificaciones pertinentes dentro de *QEMU* para obtener una traza en la consola cada vez que se vaya a ejecutar una instrucción cuya dirección se encuentre en un rango de interés, concretamente direcciones en el rango en el que se encuentra el código de la primera etapa del cargador de arranque. Este código será cargado por la BIOS en el rango de memoria física [0x00007c00-0x00007E00) [39] y se trata del primer sector del disco de arranque en un sistema legacy BIOS, como lo es *QEMU* por defecto.

Para conseguirlo aprovechamos el proceso de traducción a código intermedio que realiza *QEMU*. Modificaremos el código de *QEMU* para que, al traducir una instrucción de interés, añada funcionalidad extra al código traducido justo antes de la instrucción de interés.

Esta funcionalidad extra o código de instrumentación lo añadimos en forma de una instrucción de código intermedio de tipo `call`, que hace una llamada a un *helper*. Cuando el anfitrión ejecute este call, el hilo del correspondiente vCPU saldrá del contexto de ejecución del anfitrión temporalmente, para ejecutar la función helper del huésped, definida por nosotros.

Como ya hemos mencionado en capítulos anteriores, estos *helpers* se usan para implementar funcionalidad compleja necesaria para emular una arquitectura, y que sería muy costoso implementar en código intermedio. Nosotros lo usamos para colocar en un *helper* nuestro código de instrumentación, y poder implementarlo como una función C en el huésped.

4.3.1. Añadir un *helper* de instrumentación

Las llamadas a helpers se hacen con la instrucción TCG ops `call` (`INDEX_op_call`):

```
call load_seg,$0x0,$0,env,tmp11,tmp5
```

El primer operando es la función del helper, le sigue un campo de flags, que discutiremos más adelante, el tipo de dato que retorna (`0x0` es **`void`**) y los parámetros que recibe la función helper. Estos parámetros tienen que ser variables TCG, como la variable global `env` o las variables temporales `tmp1, tmp11`. Recordamos que el código intermedio opera sobre variables TCG.

No podemos definir una función dentro de *QEMU*, generar un `call` y esperar a que funcione. Hay que registrar el helper usando macros del preprocesador. De esta manera, *QEMU* añadirá el helper a una tabla interna para poder despachar los calls.

Además, una vez registrado un helper, por ejemplo, `load_seg`, el preprocesador de C pone a nuestra disposición la macro `gen_helper_load_seg()` para generar llamadas al helper. A esta función solo le pasamos los parámetros como variables TCG y de toda la fontanería interna para el paso de parámetros se encarga *QEMU*.

El campo de flags sirve para indicarle a *QEMU* cómo de instrusivo es el helper relativo al acceso al estado de la máquina. Si se le dice a *QEMU* que un helper no accede a las variables TCG o no escribe en ellas la entrada y salida de los helpers es más eficiente. Todos los helpers que generamos nosotros tienen el flag por defecto `0x0`, que nos permite leer y escribir el estado de la máquina durante la ejecución de los mismos.

Vamos a detallar los pasos a seguir para añadir el *helper* de instrumentación del flujo de código que usaremos en la próxima sección.

Primero añadimos en el archivo `accel/tcg/tcg-runtime.h` la siguiente línea:

```
DEF_HELPER_2(tfgmihai_ins_cb, void, env, i64)
```

Esta línea le dice al entorno de compilado que vamos a necesitar un helper de nombre `tfgmihai_ins_cb`, que recibe dos parámetros, retorna `void` (no retorna nada), y los dos parámetros son de tipo `env` y `i64`. El tipo `env` significa que este primer parámetro será siempre el entorno del procesador, para poder acceder al estado de la máquina, aunque para este experimento no lo necesitamos.

Posteriormente debemos definir la función del helper propiamente dicha, la que acabará siendo invocada cuando se ejecuta el respectivo `call`. Para ello, añadimos la siguiente función por ejemplo, en `accel/tcg/tcg-runtime.c`:

```
void HELPER(tfgmihai_ins_cb)(CPUArchState *env, uint64_t insaddr)
{
    qemu_log("[DEBUG]->> TFGMIHAI INSADDR: %p\n", (void*)insaddr );
}
```

Listing 4.1: Helper de instrumentación para el flujo de código.

Elegimos este archivo porque aquí se encuentran algunos de los helpers ya implementados. Cuando el helper es invocado, imprime en el log (salida de error) el valor que recibe como segundo parámetro. Ya veremos lo que significa y cuando será invocado el helper en la próxima sección.

Por si queremos poner un breakpoint, nuestra función helper tiene de nombre `helper_tfgmihai_ins_cb`. A partir de ahora tenemos a nuestra disposición la macro `gen_helper_tfgmihai_ins_cb` que usaremos en la próxima sección durante el proceso de traducción a código intermedio para generar un `call` a nuestro helper.

4.3.2. Inyectar una llamada al *helper* en el código intermedio

Una vez declarado el helper de instrumentación, debemos conseguir inyectar un `call` al helper justo antes de cualquier instrucción que se encuentre en el rango de interés.

Para conseguirlo, modificamos `translator_loop()` en `accel/tcg/translator.c`. Recordamos del capitulo 3 que en esta función se halla un bucle que en cada iteración traduce una instrucción del huesped a código intermedio.

Primera modificación

```
62      while (true) {
63          db->num_insns++;
            /* BEGIN MODIFICACION */
            g_tfgmihai_current_pc = db->pc_next;
            /* END MODIFICACION */
64          ops->insn_start(db, cpu);
65          tcg_debug_assert(db->is_jmp == DISAS_NEXT);
```

Esta modificación (entre las líneas 63 y 64 del archivo original) guardan en una variable global la dirección de la instrucción que se procederá a traducir. Este valor se obtiene de una variable del objeto `DisassemblyContext`. Es importante aclarar que la variable `g_tfgmihai_current_pc` es local a cada hilo, y la tenemos que declarar, por ejemplo, antes de la definición de `translator_loop()`, tal que así:

```
__thread target_ulong g_tfgmihai_current_pc = 0;
```

Siempre que deseemos usar esta variable en un archivo distinto a `accel/tcg/translator.c` hay que incluir, antes del primer uso y fuera de una función:

```
extern __thread target_ulong g_tfgmihai_current_pc;
```

Para la realización del experimento de esta sección no necesitamos que la la dirección de la instrucción se encuentre en una variable global, pues la vamos a pasar como parámetro al helper, sin embargo, sí que necesitamos esta modificación para el experimento que se realizará en la próxima sección.

Segunda modificación

Dentro del mismo bucle, entre las líneas 65 y 66 añadimos:

```
65          tcg_debug_assert(db->is_jmp == DISAS_NEXT);
            /* BEGIN MODIFICACION */
            if (g_tfgmihai_current_pc >= 0x7c00 && g_tfgmihai_current_pc < 0
                x7E00) {
                TCGv_i64 _insaddr = tcg_temp_new_i64();
                tcg_gen_movi_i64(_insaddr, g_tfgmihai_current_pc);
                gen_helper_tfgmihai_ins_cb(cpu_env, _insaddr);
                tcg_temp_free(_insaddr);
            }
            /* END MODIFICACION */
66
67          if (plugin_enabled) {
```

Este código comprueba que la instrucción que se está traduciendo está en el rango de interés y en caso afirmativo genera dos instrucciones de código intermedio que preceden al código intermedio de la instrucción que se está traduciendo.

Podemos ver un ejemplo de estas dos instrucciones:

```
movi_i64 tmp13,$0x7c00
call tfgmihai_ins_cb,$0x0,$0,env,tmp13
```

La segunda es la llamada a nuestro `helper` de instrumentación, que recibe como parámetro el contexto del procesador y una variable temporal.

Recordamos que las instrucciones TCG ops operan sobre variables temporales. En nuestro caso, la variable temporal `_insaddr` es reservada por nosotros y liberada cuando ya no se requiere (`tcg_gen_movi_i64()`,`tcg_temp_free()`).

En la primera instrucción, le asignamos a la variable temporal un valor, la dirección de la instrucción que se va a traducir, y por tanto la instrucción a instrumentar, el cual conocemos cuando generamos esta instrucción y que tendrá marcado cono una constante o valor inmediato.

Habitualmente las instrucciones TCG solo pueden recibir parámetros en forma de variables TCG y no constantes (`movi` es un excepción). Es por esto que necesitamos dos instrucciones y no solo una.

Examinemos el código máquina y su correspondiente código intermedio de los bloques de código usando el mismo mecanismo explicado en el capítulo anterior:

```
$ qemu-system-x86_64 debian_wheezy.qcow2 -d op_opt,in_asm 2> traza.txt
```

Este bloque es el que contiene la primera instrucción del rango, en la dirección 0x7c00.

```
----------------
IN:
0x00007c00:  eb 63                    jmp      0x7c65

OP after optimization and liveness analysis:
 ld_i32 tmp11,env,$0xfffffffffffffff0     dead: 1  pref=0xffff
 movi_i32 tmp12,$0x0                      pref=0xffff
 brcond_i32 tmp11,tmp12,lt,$L0            dead: 0 1
```

```
 ---- 0000000000007c00 0000000000000000
 movi_i64 tmp13,$0x7c00                    <-
 call tfgmihai_ins_cb,$0x0,$0,env,tmp13    <-
 goto_tb $0x0
 movi_i64 tmp3,$0x7c65                     pref=0xffff
 st_i64 tmp3,env,$0x80                     dead: 0 1
 exit_tb $0x7f9aac171180
 set_label $L0
 exit_tb $0x7f9aac171183

[DEBUG]->> TFGMIHAI INSADDR:0x7c00
----------------
```

Listing 4.2: Código producido con las instrucciones de instrumentación.

Las instrucciones que hemos generado con nuestra modificación durante la traducción del bloque aparecen en negrita en el listado y marcadas con <-.

Si seguimos mirando el código intermedio de más bloques, vemos que cada instrucción que esté en el rango de interés tiene estas dos instrucciones TCG como predecesoras en su traducción, con el valor adecuado de la constante. Por ejemplo, el siguiente bloque consta de dos instrucciones fuente, y cada una, en su código intermedio tiene el código extra de instrumentación. El valor de la constante, como se puede observar, es la dirección original de cada instrucción.

```
IN:
0x00007c6d:  f6 c2 70                 testb    $0x70, %dl
0x00007c70:  74 02                    je       0x7c74

OP after optimization and liveness analysis:
 ld_i32 tmp11,env,$0xfffffffffffffff0     dead: 1  pref=0xffff
 movi_i32 tmp12,$0x0                      pref=0xffff
 brcond_i32 tmp11,tmp12,lt,$L0            dead: 0 1

 ---- 0000000000007c6d 0000000000000000
 movi_i64 tmp13,$0x7c6d                   <-
 call tfgmihai_ins_cb,$0x0,$0,env,tmp13   <-
 movi_i64 tmp1,$0x70                      pref=0xffff
 and_i64 cc_dst,rdx,tmp1                  sync: 0  dead: 0 1 2  pref=0xffff
 discard cc_src                           pref=0xff3f
 discard cc_src2                          pref=0xffff
 discard cc_op                            pref=%rsi

 ---- 0000000000007c70 0000000000000016
 movi_i64 tmp13,$0x7c70                   <-
 call tfgmihai_ins_cb,$0x0,$0,env,tmp13   <-
 ext8u_i64 tmp0,cc_dst                    dead: 1  pref=0xffff
 movi_i32 cc_op,$0x16                     sync: 0  dead: 0  pref=0xffff
 movi_i64 tmp13,$0x0                      pref=0xffff
 brcond_i64 tmp0,tmp13,eq,$L1             dead: 0 1
 goto_tb $0x0
 movi_i64 tmp3,$0x7c72                    pref=0xffff
 st_i64 tmp3,env,$0x80                    dead: 0 1
 exit_tb $0x7f9aac171480
 set_label $L1
 goto_tb $0x1
 movi_i64 tmp3,$0x7c74                    pref=0xffff
 st_i64 tmp3,env,$0x80                    dead: 0 1
 exit_tb $0x7f9aac171481
 set_label $L0
 exit_tb $0x7f9aac171483

[DEBUG]->> TFGMIHAI INSADDR:0x7c6d
[DEBUG]->> TFGMIHAI INSADDR:0x7c70
```

Listing 4.3: Código producido con las instrucciones de instrumentación incluidas precediendo los accesos a memoria.

Cuando este código intermedio sea traducido a código nativo y posteriormente ejecutado, contendrá las instrucciones necesarias para salir del contexto del huésped y ejecutar nuestro `helper` justo antes de el código que emula cada instrucción original en el rango de interés.

Cuando el helper es llamado, imprimirá en la salida de error la traza programada:

```
[DEBUG]->> INSADDR:0x7c00
[DEBUG]->> TFGMIHAI INSADDR:0x7c65
<snip>
[DEBUG]->> TFGMIHAI INSADDR:0x7d66
[DEBUG]->> TFGMIHAI INSADDR:0x7d67
```

Hay una traza por cada instrucción ejecutada en el rango de interés. Cada traza contiene la dirección de la instrucción ejecutada.

En los listados de depurado generados con el flag `-d` se ven entremezcladas estas trazas con el desensamblado del código de los bloques, debido a que ambas se imprimen en la salida de error. Si eliminamos el flag `-d` solo se verán las trazas.

4.4 Instrumentación de los accesos a memoria

Ya hemos visto como instrumentar de manera selectiva el flujo de ejecución de la máquina emulada. En esta sección mostraremos cómo instrumentar los accesos a memoria.

Queremos mostrar cómo se implementaría una técnica *inline* de instrumentación de los accesos a memoria. Esta técnica consiste en preceder las instrucciones de acceso a memoria que resultan de nuestro interés con una llamada a un helper de instrumentación de manera similar a como se ha instrumentado el flujo de ejecución en la sección anterior.

Para ello, similar a la sección anterior, nos ponemos como objetivo obtener una traza de todos los accesos a memoria que realiza el código de la primera etapa del cargador de arranque, que se encuentra en el rango de direcciones ya mencionado.

Ya hemos visto en la sección que hay un mecanismo de trazado de *QEMU*. Este mecanismo nos prmite "escuchar" eventos de memoria. Es decir, *QEMU* nos informa de todos los acceso a memoria que realiza la máquina. Posteriormente podemos filtrar estos accesos.

Sin embargo, el mecanismo de trazado no permite ejecutar código de instrumentación antes o después de los accesos, y no es por tanto un mecanismo de instrumentación propiamente dicho. No es programable. No se podría utilizar para filtrar los accesos a memoria en función del estado de la máquina en el momento en el que se producen (como el valor de un registro), ni tampoco realizar acciones arbitrarias antes del acceso como por ejemplo leer memoria.

4.4.1. Helper de instrumentación

De la misma manera que en la sección anterior, definiremos un helper con nuestro código de instrumentación, siguiendo los mismos pasos que en la sección anterior.

Sin embargo, esta vez nuestro helper tomará la siguiente forma:

```
void HELPER(tfgmihai_mem_cb)(CPUArchState *env, uint64_t insaddr, uint64_t
    addr, uint32_t info)
{
    do_tfgmihai_mem_cb(env, insaddr, addr, info);
```

```
}
```

Listing 4.4: Helper de instrumentación para los accesos a memoria.

```
DEF_HELPER_4(tfgmihai_mem_cb, void, env, i64, i64, i32)
```

El código de instrumentación se encuentra en la función llamada por el helper en lugar de en el helper directamente. La razón es que necesitamos llamar al código de instrumentación no solo desde el helper, esto se verá más adelante.

El primer parámetro es el entorno del procesador, el siguiente la dirección de la instrucción de acceso a memoria que se está instrumentando. Le sigue la dirección a la que va a acceder esa instrucción y un campo de bits con información sobre el acceso (si es lectura/escritura, tamaño del acceso...).

La implementación de `do_tfgmihai_mem_cb()` se verá más adelante.

4.4.2. Llamadas al helper

Ahora necesitamos realizar una modificación de naturaleza similar a la realizada en la sección anterior. Debemos conseguir que las instrucciones TCG ops que realizan un acceso a memoria se vean precedidas por un `call` al helper.

Afortunadamente, el código intermedio TCG es de tipo RISC y solo tiene cuatro instrucciones para el acceso a memoria:

```
ld_i32
ld_i64
st_i32
st_i64
```

El resto de instrucciones trabajan sobre valores inmediatos o variables TCG.

Si queremos inyectar una instrucción antes de cualquiera de estas instrucciones, tenemos dos opciones:

- Iterar sobre las instrucciones TCG ops de un bloque una vez emitidas a la lista `tcg_ctx->ops`, detectar las que tienen un código de operación pertinente (`INDEX_op_ld_i64...`) y precederlas con un `call`.
- Encontrar los puntos dentro el código de *QEMU* donde se generan estas instrucciones de acceso a memoria, y parchearlo para generar una llamada al helper con `gen_helper_tfgmihai_mem_cb()`

La primera opción es más complicada de lo que parece, ya que las funciones `gen_helper_*` no pueden generar una instrucción en mitad de la lista de instrucciones sino únicamente en la cola, por lo que habría que inyectar el `call` manualmente con `tcg_emit_op()` o manipular la lista.

Vamos a elegir la segunda opción, pero es pertinente mencionar que la primera opción sería una técnica bastante general de instrumentación.

Para implementar la segunda opción debemos estudiar cómo se generan estas instrucciones. Si buscamos la palabra clave `ld_i64` dentro del código del FRONTEND de *x86_64* [1] usando la orden `grep -r -I ld_i64` vemos llamadas a `tcg_gen_ld_i64()` y `tcg_gen_qemu_ld_i64()`. Obtenemos un resultado simétrico si buscamos `st_i64` y las demás variantes.

[1] Comparte la misma carpeta con el de *i386*, `target/i386`

Si ponemos un breakpoint en estas funciones y sacamos una traza de la pila vemos que son invocadas indirectamente por `translator_loop()`, como parte del proceso de traducción a código intermedio. Observamos en la traza de la pila que las llamadas a `tcg_gen_*` son muy poco frecuentes comparadas con aquellas `tcg_gen_qemu_*`.

Leyendo el código de `tcg_gen_qemu_(ld|st)_(i32|i64)` que se encuentra en `tcg/tcg-op.c` vemos que en estas funciones se encuentra el código de instrumentación del mecanismo de tracing de *QEMU* y del sistema de plugins, por lo que es una buena señal de que podemos colocar en este camino de código nuestra instrumentación.

Sin embargo nos gustaría generar la llamada al call en un solo sitio.

Todas estas funciones (`tcg_gen_qemu_*`) acaban llamando a `gen_ldst_(i64|i32)()`, por ejemplo, en `tcg_gen_qemu_ld_i64()` tenemos:

```
gen_ldst_i64(INDEX_op_qemu_ld_i64, val, addr, memop, idx);
```

La versión de 32 bits solo es llamada cuando el sistema emulado es de 32 bits, por lo que la podemos ignorar. Es decir, hasta los `ld_i32, st_i32` son generados en `gen_ldst_i64`. Recordamos que solo trabajamos con *x86_64*.

Es por tanto en `gen_ldst_i64` donde se van a generar todas las instrucciones de acceso a memoria que nos interesan. Vamos a colocar la generación de la llamada al helper al comienzo de `gen_ldst_i64`

```
static void gen_ldst_i64(TCGOpcode opc, TCGv_i64 val, TCGv addr,
                         MemOp memop, TCGArg idx)
{
    /* BEGIN MODIFICACION */
    if(g_tfgmihai_current_pc >= 0x00007c00
    && g_tfgmihai_current_pc < 0x00007e00) {

        TCGv_i64 _insaddr = tcg_temp_new_i64();
        TCGv_i32 _info = tcg_temp_new_i32();

        tcg_gen_movi_i64(_insaddr, g_tfgmihai_current_pc);
        tcg_gen_movi_i32(_info, make_memop_idx(memop, idx));

        gen_helper_tfgmihai_mem_cb(cpu_env, _insaddr, addr, _info);

        tcg_temp_free_i64(_insaddr);
        tcg_temp_free_i32(_info);
    }
    /* END MODIFICACION */
    TCGMemOpIdx oi = make_memop_idx(memop, idx);
    ... ...
```

`opc` es el *opcode* TCG que se va a generar, `val` es la variable TCG (habitualmente registro temporal) que se ve involucrada en el acceso, es decir, dónde se almacenará el valor leído o de donde se sacará el valor a escribir.

`addr` es la variable TCG que contiene la dirección a la que se va a acceder. `memop` es un campo de bits con información sobre el acceso, nos permitirá saber si se trata de una lectura o una escritura entre otras cosas. Finalmente `idx` no es relevante.

Necesitamos pasarle a nuestro helper de instrumentación, por un lado la dirección donde se encuentra la instrucción que generó el acceso, la cual obtenemos de la variable global que actualizamos en el bucle de traducción. Necesitamos pasarle también `addr` y `memop`.

Aquellos parámetros que no sean variables TCG necesitamos que estén en una variable TCG. Para ello reservamos las dos variables TCG necesarias y generamos antes del `call` dos instrucciones `mov` para llenarlas con el valor adecuado.

Vemos que necesitamos que `g_tfgmihai_current_pc` sea una variable global asignada en el bucle de traducción, para poder utilizarla desde fuera de ese bucle.

Es pertinente comentar que aquellas instrucciones de acceso a memoria que no se encuentren en el rango de interés no tendrán un `call` de instrumentación precedente, gracias a la comprobación realizada al principio de la modificación.

4.4.3. Filtrado de la instrumentación *inline* en base al estado

Es pertinente realizar una discusión sobre el filtrado usado para decidir qué instrucciones son instrumentadas y cuáles no con un `call`.

En el bucle de traducción guardamos la dirección de la instrucción que está siendo traducida de manera que podemos recuperar este valor cuando se está haciendo la generación de un `ld/st`, y así decidir en base a este valor si añadir un `call` de instrumentación o no.

Esto es posible porque ese valor es conocido en tiempo de traducción, análogo al tiempo de compilación en un esquema de compilado más habitual.

Si quisiéramos instrumentar solo aquellos accesos a memoria que cumplan una condición que es resoluble únicamente en tiempo de ejecución pero no de traducción, no podríamos decidir si colocar o no un `call`. Deberíamos colocar un `call` delante de todos los accesos y decidir en el propio helper de instrumentación, con acceso ya a información en tiempo de ejecución, si realizar la instrumentación/trazado o simplemente retornar sin hacer nada.

Por ejemplo, imagínese que se desea instrumentar únicamente los accesos a memoria que cumplan la condición de que cuando se lleven a cabo el registro `RAX` contenga el valor `0xDEADBEEF`, por poner un ejemplo.

Esa condición solo se puede comprobar cuando ya se han ejecutado las instrucciones previas al acceso a memoria. Es decir, al traducir un acceso a memoria de un bloque a código intermedio no podemos conocer el estado de la máquina previo al acceso, pues el bloque está siendo traducido y aún no se ha ejecutado.

Por tanto, lo que habría que hacer es instrumentar todos los accesos, asumiendo el coste de una salida del huésped por culpa del helper. Dentro del helper *sí* que se puede comprobar el valor de `RAX`, pues se han ejecutado las instrucciones previas del bloque.

En el caso de desear realizar instrumentación que dependa del estado de la máquina en puntos concretos de ejecución dentro de un bloque de código sería conveniente analizar si la sobrecarga de la instrumentación inline (salida del huésped provocada por el helper) es asumible, o si resulta más eficiente instrumentar, por ejemplo, los accesos a memoria dentro del código del *softMMU* de *QEMU*.

Si se opta por la instrumentación inline, se podría, durante la ejecución del propio helper y una vez se ha podido decidir si se debe continuar o no, eliminar el `call` que ha provocado la llamada a sí mismo, para que no se vuelva a llamar cuando se ejecute de nuevo el bloque cacheado. Si se opta por la instrumentación inline, se podría, durante la ejecució del propio helper y una vez se ha podido decidir si se debe continuar o no, eliminar el `call` que ha provocado la llamada a sí mismo, para que no se vuelva a llamar cuando se ejecute de nuevo el bloque cacheado.

Esta tarea sería muy complicada, puesto que no se requiere eliminar un `call` del código intermedio, sino del código nativo al que se ha traducido este código intermedio. Sería una tarea dependiente de la arquitectura del anfitrión.

4.4.4. Accesos desde helpers existentes

Hasta ahora hemos conseguido que las instrucciones TCG de acceso a memoria se vean precedidas por un `call` de instrumentación. Sin embargo, existen más accesos a memoria que todavía no hemos instrumentado.

Estos accesos se realizan desde algunas funciones helper ya existentes. Necesitamos encontrar dónde se producen estos accesos e introducir una llamada a `do_tfgmihai_mem_cb()` previo al acceso. Esta es la razón por la que el código de instrumentación lo hemos situado en una función a parte en lugar de directamente en el helper que hemos introducido.

Afortunadamente, el mecanismo de tracing de *QEMU* instrumenta los puntos necesarios dentro de para hacer una traza de los accesos a memoria. Esto incluye los accesos producidos desde helpers existentes.

Investigando el código del mecanismo de tracing, la función invocada es `trace_guest_mem_before_exec()`.

Si buscamos esta función con `grep -r -I trace_guest_mem_before_exec` y obviamos el código de otras arquitecturas y de *QEMU Usermode* vemos que es invocada desde varios helpers en los archivos:

```
accel/tcg/cputlb.c
accel/tcg/atomic_common.inc.c
```

Como es razonable, cada helper no implementa sus propios accesos a memoria, sino que llama a unas funciones para acceder a memoria. En esas funciones se encuentra el código de instrumentación de tracing y es donde debemos colocar nuestra llamada a `do_tfgmihai_mem_cb()`, por ejemplo:

```
// en accel/tcg/cputlb.c
static inline void QEMU_ALWAYS_INLINE
cpu_store_helper(CPUArchState *env, target_ulong addr, uint64_t val,
                 int mmu_idx, uintptr_t retaddr, MemOp op)
{
    TCGMemOpIdx oi;
    uint16_t meminfo;

    meminfo = trace_mem_get_info(op, mmu_idx, true);
    trace_guest_mem_before_exec(env_cpu(env), addr, meminfo);

    oi = make_memop_idx(op, mmu_idx);
    /* BEGIN MODIFICACION */
    CPUState *cs = env_cpu(env);
    CPUX86State *x86_env = &(X86_CPU(cs))->env;

    if(x86_env->eip >= 0x00007c00
    && x86_env->eip <  0x00007e00) {
        do_tfgmihai_mem_cb(env, g_tfgmihai_current_pc, addr, meminfo);
        //qemu_log("***PRUEBA*** eip=%p\n", (void*)x86_env->eip);
    }
    /* END MODIFICACION */
    store_helper(env, addr, val, oi, retaddr, op);

    qemu_plugin_vcpu_mem_cb(env_cpu(env), addr, meminfo);
}
```

Esta función es invocada por todos los helpers cuando quieren hacer un acceso de escritura a la memoria. Retomando la discusión sobre el estado de la máquina, no es correcto hacer la comprobación de la dirección usando `g_tfgmihai_current_pc`.

Cuando hicimos el filtrado a la hora de generar instrucciones nos encontrábamos en tiempo de traducción/compilación, y no ejecución. Esta función se ejecuta en tiempo de ejecución, como parte de la ejecución de un helper. El valor de `g_tfgmihai_current_pc` probablemente corresponda a la dirección de la última instrucción traducida, no de la instrucción que está provocando el helper.

Para obtener la dirección de la instrucción que ha invocado el helper que se está manejando, y que por tanto está intentando realizar el acceso a memoria, podemos acceder al valor del registro del contador de programa en la estructura que mantiene el estado del procesador.

Para no llenar de información esta sección, los detalles de cómo se accede al estado del procesador se explicarán en la siguiente sección. Lo importante es aclara que en tiempo de ejecución podemos acceder a información exacta instrucción a instrucción sobre estado de la máquina, mientras que en tiempo de traducción/compilación no.

Hay que implementar exactamente la misma modificación en `cpu_load_helper()` y también en las funciones de `accel/tcg/atomic_common.inc.c` que invocan a `trace_guest_mem_before_exec()` . Pongo un ejemplo de una de ellas únicamente, pues las demás son idénticas.

```
static inline
void atomic_trace_rmw_pre(CPUArchState *env, target_ulong addr, uint16_t info)
{
    CPUState *cpu = env_cpu(env);

    trace_guest_mem_before_exec(cpu, addr, info);
    trace_guest_mem_before_exec(cpu, addr, info | TRACE_MEM_ST);
    /* BEGIN MODIFICACION */
    CPUState *cs = env_cpu(env);
    CPUX86State *x86_env = &(X86_CPU(cs))->env;

    if(x86_env->eip >= 0x00007c00
    && x86_env->eip <  0x00007e00) {
        do_tfgmihai_mem_cb(env, g_tfgmihai_current_pc, addr, info);
    }
    /* END MODIFICACION */

}
```

4.4.5. Código de instrumentación

Ahora que hemos conseguido que cada acceso de memora se vea precedido por una llamada a `do_tfgmihai_mem_cb()`, expliquemos su implementación.

Esta función la hemos incluido en `accel/tcg/tcg-runtime.c` pero puede estar en cualquier archivo de código fuente incluido en la compilación.

```
void do_tfgmihai_mem_cb(CPUArchState *env, uint64_t insaddr, uint64_t addr,
    uint32_t info);

#define TRACE_MEM_ST (1ULL << 6)     /* store (y/n) */
void do_tfgmihai_mem_cb(CPUArchState *env, uint64_t insaddr, uint64_t addr,
    uint32_t info) {
    TCGMemOpIdx oi = (TCGMemOpIdx)info;
    CPUState *cs = env_cpu(env);

    MemOp memop = get_memop(oi);
    unsigned size = memop_size(memop); // 1 << (op & MO_SIZE);
    bool is_write = !!(info & TRACE_MEM_ST);

    X86CPU *x86 = X86_CPU(cs);
    CPUX86State *x86_env = &x86->env;
```

```
    // prueba leer memoria
    char data[16];
    memset(data, 0x00, 16);

    if(!is_write) {
        cpu_memory_rw_debug(cs, addr, data, size, 0); // read
    }
    qemu_log("[DEBUG]->> TFGMIHAI %s addr:%p insaddr:%p size:%u RAX=%lx "
             "RSP=%lx debug_read=%llx\n",
             (is_write? "WRITE" : "READ"), (void*)addr, (void*)insaddr, size,
             mmu_idx, x86_env->regs[R_EAX], x86_env->regs[R_ESP],
             x86_env->regs[R_ECX], x86_env->regs[R_EBX],
             *((unsigned long long *)data));
}
```

Listing 4.5: Código de instrumentación de los accesos a memoria.

La función de instrumentación lo que hace es imprimir una traza en la salida de error. Esta traza contiene la dirección del acceso que se va a producir, la dirección de la instrucción original que lo produce y el tipo de acceso (lectura o escritura). Además, para hacer una demostración de cómo se puede acceder al estado de la máquina durante la instrumentación, se imprimirá en la traza el valor de algunos registros de propósito general y también el valor de la memoria que se leerá en caso de que se trate de un acceso de lectura.

A partir del campo de bits `info` se extrae la información sobre el acceso a memoria de la misma manera en que lo hace *QEMU* en otras partes del código. Se muestran dos maneras, usando un función auxiliar y extrayendo el bit de interés directamente.

Para acceder al estado de la máquina obtenemos el puntero al objeto `CPUState` a partir de la variable TCG `env`. Posteriormente se fuerza una conversión de tipo a una CPU de la arquitectura de interés, de la misma manera en que lo hace el código del *QEMU Object Model*.

Para leer la memoria se utiliza la función que usa para leer memoria la implementación del *gdbstub* [2].

4.4.6. Ejemplo de traza

Veamos un ejemplo de la traza producida en la salida de error. Se filtra la traza del flujo de código pues ya la hemos visto antes.

```
qemu-system-x86_64 debian_wheezy_amd64.qcow2 2>traza.txt'
```

```
grep 'TFGMIHAI READ\|WRITE' traza.txt | head
```

```
[DEBUG]->> TFGMIHAI READ addr:0x7c64 insaddr:0x7c83 size:1 RAX=0 RSP=2000 debug_read=ff
[DEBUG]->> TFGMIHAI READ addr:0x1ffe insaddr:0x7c8c size:2 RAX=ff RSP=2000 debug_read=0
[DEBUG]->> TFGMIHAI READ addr:0x1ffc insaddr:0x7c90 size:2 RAX=ff RSP=1ffe debug_read=0
[DEBUG]->> TFGMIHAI READ addr:0x7d80 insaddr:0x7daa size:1 RAX=ff RSP=1ffc debug_read=47
[DEBUG]->> TFGMIHAI READ addr:0x40 insaddr:0x7da8 size:1 RAX=e47 RSP=1ffc debug_read=6c
[DEBUG]->> TFGMIHAI READ addr:0x42 insaddr:0x7da8 size:1 RAX=e47 RSP=1ffc debug_read=0
[DEBUG]->> TFGMIHAI WRITE addr:0x1ffa insaddr:0x7da8 size:1 RAX=e47 RSP=1ffc debug_read=0
[DEBUG]->> TFGMIHAI WRITE addr:0x1ff8 insaddr:0x7da8 size:1 RAX=e47 RSP=1ffc debug_read=0
[DEBUG]->> TFGMIHAI WRITE addr:0x1ff6 insaddr:0x7da8 size:1 RAX=e47 RSP=1ffc debug_read=0
[DEBUG]->> TFGMIHAI READ addr:0x7d81 insaddr:0x7daa size:1 RAX=e47 RSP=1ffc debug_read=52
```

```
grep 'TFGMIHAI READ\|WRITE' traza.txt | tail
```

```
[DEBUG]->> TFGMIHAI READ addr:0x81fe insaddr:0x7d63 size:2 RAX=1 RSP=1fec debug_read=0
[DEBUG]->> TFGMIHAI READ addr:0x1fec insaddr:0x7d65 size:2 RAX=1 RSP=1fec debug_read=0
[DEBUG]->> TFGMIHAI READ addr:0x1fee insaddr:0x7d66 size:2 RAX=1 RSP=1fee debug_read=0
[DEBUG]->> TFGMIHAI READ addr:0x1ff0 insaddr:0x7d66 size:2 RAX=1 RSP=1fee debug_read=7c05
[DEBUG]->> TFGMIHAI READ addr:0x1ff2 insaddr:0x7d66 size:2 RAX=1 RSP=1fee debug_read=0
```

[2] Ver `handle_write_mem()` en `gdbstub.c`

```
[DEBUG]->> TFGMIHAI READ addr:0x1ff6 insaddr:0x7d66 size:2 RAX=1 RSP=1fee debug_read=7000
[DEBUG]->> TFGMIHAI READ addr:0x1ff8 insaddr:0x7d66 size:2 mmu_idx:2 RAX=1 RSP=1fee RCX=0 RBX=7000 debug_read=80
[DEBUG]->> TFGMIHAI READ addr:0x1ffa insaddr:0x7d66 size:2 mmu_idx:2 RAX=1 RSP=1fee RCX=0 RBX=7000 debug_read=1
[DEBUG]->> TFGMIHAI READ addr:0x1ffc insaddr:0x7d66 size:2 mmu_idx:2 RAX=1 RSP=1fee RCX=1 RBX=7000 debug_read=1
[DEBUG]->> TFGMIHAI READ addr:0x7c5a insaddr:0x7d67 size:2 mmu_idx:2 RAX=1 RSP=1ffe RCX=1 RBX=7000 debug_read=8000
```

Se pueden ver las trazas de los primeros y últimos accesos. El proceso *QEMU* se dejó en ejecución hasta que empezaron los mensajes de la carga del núcleo de linux. Eso significa que todo el código de la primera etapa del cargador de arranque se ejecutó en su completitud. Como es de esperar, todos los accesos trazados están producidos por instrucciones en el rango de interés (`insaddr`).

CAPÍTULO 5
Conclusiones

A pesar de la escasa documentación técnica para desarrolladores del emulador *QEMU*, se ha conseguido abordar con éxito los objetivos iniciales del trabajo:

1. Presentar una clasificación de las diferentes técnicas de instrumentación de software.
2. Se ha realizado un análisis del estado del arte de las técnicas para el análisis automático de software orientado al descubrimiento de fallos de seguridad, especialmente de corrupción de memoria. Estas técnicas necesitan de herramientas de instrumentación para logras su cometido.
3. Debido a la falta de documentación, se ha tenido que realizar un importante esfuerzo de ingeniería inversa para descubrir el funcionamiento del código interno de *QEMU*. No en su totalidad, pero sí de aquella parte necesaria para la implementación de las técnicas de instrumentación detalladas en el punto siguiente.
4. Se ha modificado *QEMU* (v4.2.0) para implementar una instrumentación inline del flujo de código y los acceso a memoria. Esto se ha hecho aprovechando el proceso de traducción dinámica del emulador.

La instrumentación implementada tiene como objetivo servir como prueba de concepto de los pasos a seguir para realizar dicha instrumentación, la cual puede resultar útil en un futuro en algún proyecto concreto.

La consecución de este trabajo me ha permitido aprender a tratar con el código de proyectos de software grandes y complejos, pues *QEMU* implementa en el mismo código base la emulación de una gran gama de arquitecturas y un traductor/compilador dinámico de código máquina.

El código de *QEMU* está en continua evolución, es por eso que la documentación técnica para desarrolladores es escasa, especialmente para versiones recientes.

Bibliografía

[1] F. Bellard. *QEMU, a fast and portable dynamic translator.* USENIX 2005 Annual Technical Conference, FREENIX Track, 2005.

[2] Luk, Chi-Keung et al. *Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation* In Sigplan Notices - SIGPLAN. 40. 190-200. (2005).

[3] C. Lattner and V. Adve, *LLVM: a compilation framework for lifelong program analysis & transformation* International Symposium on Code Generation and Optimization, 2004. CGO 2004., San Jose, CA, USA, 2004, pp. 75-86, doi: 10.1109/CGO.2004.1281665.

[4] K. Serebryany et al. *AddressSanitizer: A Fast Address Sanity Checker.* USENIX 2012 Annual Technical Conference.

[5] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. *KAFL: hardware-assisted feedback fuzzing for OS kernels.* In Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17). USENIX Association, USA, 167–182.

[6] Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R., and Holz, T. *REDQUEEN: Fuzzing with Input-to-State Correspondence.* NDSS (2019)

[7] P. Chen and H. Chen *Angora: Efficient Fuzzing by Principled Search* 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, 2018, pp. 711-725

[8] E. J. Schwartz, T. Avgerinos, and D. Brumley. *All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask).* Security and Privacy SP 2010 IEEE Symposium on, (7):317–331, 2010

[9] Anand, Saswat. *Techniques to facilitate symbolic execution of real-world programs. (2012).*

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. *KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs* In OSDI '08, pages 209–224. USENIX, 2008.

[11] S. K. Cha, T. Avgerinos, A. Rebert and D. Brumley. *Unleashing Mayhem on Binary Code* 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, 2012, pp. 380-394, doi: 10.1109/SP.2012.31.

[12] Chipounov, Vitaly & Kuznetsov, Volodymyr & Candea, George. (2012). *S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems.* Computer Architecture News. 39. 10.1145/1961295.1950396.

[13] Recurso web. QEMU Internals: Big picture overview http://blog.vmsplice.net/2011/03/qemu-internals-big-picture-overview.html

[14] Carnegie Mellon University Binary Analysis Platform https://github.com/BinaryAnalysisPlatform/bap

[15] Recurso web. CWE-416: Use After Free https://cwe.mitre.org/data/definitions/416.html

[16] QEMU Internals (versión anterior) https://qemu.weilnetz.de/doc/2.7/qemu-tech.pdf

[17] x86-64 psABI version 1.0 https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI

[18] https://qemu.weilnetz.de/doc/qemu-doc.html#pcsys_005fmonitor

[19] https://www.eetimes.com/how-to-choose-an-in-circuit-emulator/

[20] http://files.meetup.com/1590495/debugging-with-qemu.pdf

[21] https://pdos.csail.mit.edu/6.828/2008/readings/i386/s12_02.htm

[22] https://sourceware.org/gdb/current/onlinedocs/gdb/Set-Watchpoints.html

[23] AddressSanitizer, ThreadSanitizer, MemorySanitizer https://github.com/google/sanitizers

[24] DynamoRIO https://dynamorio.org/

[25] Valgrind https://www.valgrind.org/

[26] Frida https://frida.re/

[27] Recurso web. Lista de proyectos basados en *Unicorne Engine* https://www.unicorn-engine.org/showcase/

[28] Unicorn - A lightweight multi-platform, multi-architecture CPU emulator framework. https://www.unicorn-engine.org/

[29] Qiling Framework - Advanced Binary Emulation Fraework https://www.qiling.io/

[30] American Fuzzy Loop https://lcamtuf.coredump.cx/afl/

[31] OSS-Fuzz: Continuous Fuzzing for Open Source Software https://github.com/google/oss-fuzz

[32] Guided in-process fuzzing of Chrome components https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html

[33] Taint analysis and pattern matching with Pin http://shell-storm.org/blog/Taint-analysis-and-pattern-matching-with-Pin/

[34] Binary analysis: Concolic execution with Pin and z3 http://shell-storm.org/blog/Binary-analysis-Concolic-execution-with-Pin-and-z3/

[35] Triton - Dynamic Binary Analysis Framework https://triton.quarkslab.com/

[36] Angr - Next-generation binary analysis framework https://angr.io/

[37] List of some scientific projects done with the S2E platform. http://s2e.systems/showcase/

[38] Tracing - Documentación para desarrolladores de *QEMU*. https://git.qemu.org/?p=qemu.git;a=blob_plain;f=docs/devel/tracing.txt;hb=HEAD

[39] Recurso web. Secuencia de arranque de un PC https://wiki.osdev.org/Boot_Sequence