UNIVERSIDAD NACIONAL DE CÓRDOBA

Fuzzing Multiplataforma Guiado con Ejecución Concólica.

Autor: Aznarez Rojo Gastón

Director: Nicolás Wolovick
Asistentes: Maria Curetti, Daniel Gutson

Córdoba, Argentina, 2023

Fuzzing Multiplataforma Guiado con Ejecución Concólica © 2023 by Gastón Aznarez Rojo is licensed under Creative Commons Attribution 4.0 International

Contenido

1. Introduccíon **4**
1.1. Motivación 4
1.2. Objetivos 5

2. Marco Teórico **6**
2.1. Fuzzing 6
2.2. Ejecución Simbólica 11

3. Not Yet Another Fuzzer **15**
3.1. El fuzzer 15
3.2. Algoritmo de fuzzing 16
3.2.1. Función de Abstracción 16
3.2.2. Generación de Inputs: *Búsqueda Generacional* 16
3.3. Implementación 22
3.3.1. Herramientas 22
3.3.2. Sanitizer 23
3.3.3. Ciclo de Ejecución 26
3.3.4. Paralelismo 29

4. Caso de estudio **30**
4.1. Análisis 31
4.2. Preparaciones 37
4.2.1. Snapshot 37
4.2.2. Configuración 39
4.3. Fuzzing 41

5. Conclusiones **43**
5.1. Análisis de Resultados 43
5.2. Conclusión 46
5.3. Trabajo futuro 46

Appendices **47**

A. NYAF - Repositorio **47**

Resumen

Fuzzing o pruebas de Fuzz, es el proceso de encontrar vulnerabilidades de seguridad en programas, dándole como entrada datos inválidos, inesperados o aleatorios. Como este proceso es altamente dependiente de la interfaz de entrada y del entorno de ejecución del sujeto bajo análisis, realizar Fuzzing sobre programas con interfaces complejas y/o entornos de ejecución poco flexibles, se vuelve más complicado y casi inviable.

En este trabajo desarrollaremos y analizaremos una herramienta para realizar pruebas de Fuzzing guiado sobre binarios compilados para plataformas foráneas y dependientes del entorno de ejecución.

El proceso se lleva a cabo capturando el programa bajo análisis, emulando el mismo y utilizando ejecución simbólica dinámica para generar casos de prueba, con el objetivo de conseguir una mayor cobertura del programa con la menor cantidad de intentos.

A su vez, se mostrarán los desafíos impuestos por la naturaleza del problema, las herramientas utilizadas y la arquitectura del fuzzer. Por último, se puso a prueba la herramienta con un caso de estudio sobre una plataforma específica, para encontrar un error introducido previamente y con el objeto de vulnerar el programa.

Abstract

Fuzzing, or Fuzz testing, is the process of finding security vulnerabilities in programs by providing invalid, unexpected, or random data as input. Since this process is highly dependent on the input interface and the execution environment of the subject under analysis, performing Fuzzing on programs with complex interfaces and/or inflexible execution environments becomes more complicated and almost unfeasible.

In this work, we will develop and analyze a tool to perform guided Fuzzing tests on binaries compiled for foreign platforms and dependent on the execution environment.

The process is carried out by capturing the program under analysis, emulating it, and using dynamic symbolic execution to generate test cases, aiming to achieve greater program coverage with the fewest attempts.

Additionally, the challenges imposed by the nature of the problem, the tools used, and the fuzzer architecture will be presented. Finally, the tool was tested with a case study on a specific platform to find a previously introduced bug and to exploit the program.

Capítulo 1

Introduccíon

1.1. Motivación

Desde el comienzo del año 2000, el Fuzzing se convirtió en una técnica del estado del arte en tareas de seguridad de software. Miles de vulnerabilidades de seguridad fueron encontradas con este método en todo tipo de software, como en programas, sistemas operativos y más.

A lo largo del tiempo, se desarrollaron muchas herramientas para realizar estas pruebas, una de ellas fue SAGE (Scalable, Automated, Guided Execution), implementando un trazado de nivel de instrucciones en x86 y emulación para realizar whitebox fuzzing de aplicaciones de lectura de archivos en Windows. Esta herramienta fue capaz de detectar la vulnerabilidad MS07-017 ANI[7] sin conocimiento específico del formato, mientras que la misma no pudo ser encontrada con análisis estático o blackbox fuzzing.

Aunque esta herramienta fue altamente exitosa en el campo, tenía como objetivo un subconjunto pequeño de los programas: aquellos con arquitectura x86 y que corrían en el sistema operativo Windows.

Este fuzzer tiene la capacidad de hacer una búsqueda sistemática pero no completa sobre los caminos de ejecución de un programa, utilizando ejecución simbólioca dinámica. Lo que permite cubrir una gran parte del programa bajo análisis, con una menor cantidad de casos de prueba y por lo tanto menos ejecuciones del programa.

Esta característica fundamental del fuzzer motivó su utilización en programas con entornos de ejecución complicados como firmwares o programas que corren sobre CPUs con pocas capacidades. Pero la limitación en el rango de programas compatibles con SAGE y las complejidades propias de estos entornos, imposibilitó su utilización con este fin.

1.2. Objetivos

En el presente trabajo se analizará el uso del algoritmo propuesto por *SAGE* [8] en entornos de ejecución distintos a aquel para el cual el binario a analizar fue compilado.

Para ello se deberá implementar un fuzzer utilizando dicho algoritmo, pero que nos permita realizar pruebas de fuzzing sobre programas compilados para diferentes arquitecturas (*ARM*, *X86*, etc.) y/o plataformas (*Windows*, *linux*).

Este fuzzer deberá lograr una separación e independencia del entorno natural de ejecución del programa, tal como el hardware para el cual el programa fue creado.

Por último, se pondrá a prueba el algoritmo en cuestión y las abstracciones logradas por el fuzzer, analizando los resultados de un caso de estudio arbitrariamente seleccionado para enfocar las pruebas en las características necesarias del fuzzer y el algoritmo. A su vez, se analizará como fueron afrontadas las complejidades propias de la naturaleza del problema.

En concreto, para cumplir los objetivos, se debe implementar un fuzzer con las siguientes características:

- Que sea multi arquitectura (*AARCH64*, *MIPS*, etc.).
- Que sea multi plataforma (*Linux*, *Windows*, etc.).
- Que utilice el algoritmo de *Búsqueda Generacional*, visto por primear vez en *SAGE*.
- Que permita realizar abstracciones del entorno natural de ejecución (Hardware, etc.)

Para luego, hacer un análisis del mismo con un caso de estudio práctico.

Capítulo 2

Marco Teórico

2.1. Fuzzing

Introducción

El término Fuzz nació en 1988 como un proyecto de la clase *"Advanced Operating Systems"* en la universidad de Wisconsin con el título de *"Operating System Utility Program Reliability - The Fuzz Generator"*, donde se propone desarrollar una herramienta que genere entradas aleatorias para luego probar la robustez de múltiples herramientas de sistemas UNIX.

Fuzzing o Fuzzing Testing es una técnica automática de inyección de fallos que consiste en darle entradas inválidas, inesperadas o aleatorias a un programa con el objetivo de hacerlo fallar. Hoy en día, esta técnica es un pilar muy importante dentro de la seguridad informática y el desarrollo de software seguro, utilizada principalmente para descubrir fallos de seguridad y lógicos dentro de programas, previa o posteriormente a la distribución del mismo.

Llamaremos *fuzzer* a aquel programa que, utilizando técnicas de fuzzing, tiene como objetivo encontrar errores en programas.

Categorización

Los fuzzers pueden ser categorizados de distintas maneras. Algunas de esas diferenciaciones son:

Según el método de generación de casos de prueba:

- Mutación: Si a partir de modificaciones en un caso de prueba inicial se generan nuevos casos.
- Generación: Si cada caso de prueba es generado independientemente.

Según el conocimiento sobre la entrada del programa:

- Estructurado: Si conoce la estructura de los datos que toma el programa, por ejemplo, si el programa toma como entrada un archivo en formato PDF y el fuzzer conoce esta estructura.

- No estructurado: Si los datos que toma el programa no tienen una estructura particular o el fuzzer no la conoce.

Según el conocimiento de la estructura del programa:

- White-box: Si la estructura del programa es conocida, por ejemplo, se tiene el código fuente del programa y se utiliza para generar casos de prueba.
- Black-box: Si la estructura del programa es totalmente desconocida, se ve al programa como una caja negra.
- Gray-box: Un intermedio entre Black-box y White-box, donde se conoce cierta información del programa, como la cobertura de un caso de prueba.

Un ejemplo de un fuzzer del estado del arte es AFL, un fuzzer de mutación, no estructurado y de gray-box.

En este trabajo nos centraremos en White-box Fuzzing, particularmente en Fuzzing guiado por cobertura combinado con ejecución simbólica. Ampliaremos sobre esto.

Formalización y notación

En este trabajo se utilizó una notación para poder formalizar el proceso de Fuzzing dada en *Exploring Abstraction Functions in Fuzzing*[3]. Para dar esta notación perimero se deben dar algunas definiciones.

Input: Los programas consumen datos como entradas para realizar sus operaciones, ya sea por la línea de comandos, por la red, archivos u otros. Un *Input* ι cubre todos los tipos de datos de entrada.

Espacio de inputs: El alfabeto de posibles ι es Σ y el conjunto de todos los posibles ι es Σ^*.

Estado concreto: Una imagen de todos los registros del sistema, memoria del programa, archivos y todo aquello que afecte a la ejecución del programa, representa un estado concreto y se denota como r. El conjunto de todos los posibles estados concretos se denota como C.

Espacio de Estado Concreto: Cada input ι alcanza una serie de estados concretos cuando es procesado. Esta traza de los estados concretos alcanzados por ι se llama *Traza de Estados Concretos* del input y se denota como cs_ι. Dado a la potencialmente infinita cantidad de inputs que pueden ser leídos por el programa, denotamos CS al conjunto de todos las *Trazas de Estados Concretos* posibles.

Espacio de Estados Abstractos: Como CS puede ser infinito (o computacionalmente imposible de enumerar), se debe abstraer el espacio de estados concretos para que diferentes estados puedan considerarse equivalentes. Esta desición fue inspirada por la técnica de *Interpretación Abstracta*, y en particular a los dominios abstractos de esta técnica.

Definimos el *Espacio de Estados Abstractos* o AS, como el dominio al cual el *Espacio de Estados Concretos* será mapeado. Los elementos de este dominio serán llamados *Estados Abstractos*.

Las técnicas de fuzzing razonan sobre el *Espacio de Estados Abstractos* de un input y no sobre el *Espacio de Estados Concretos*. Esto les permite agrupar inputs teniendo distintos cs y así generar inputs de interés de forma eficiente.

Para mapear entre AS y CS vamos a definir dos funciones:

Función de Abstracción (α): Esta función mapea un *Traza de Estados Concretos* a *Estados Abstractos*.
Y al conjunto de de las *Funciones de Abstracción* lo llamaremos A.

$$\alpha : CS \rightarrow AS$$

Función de Concretización (γ): Esta función mapea de un *Estados Abstractos* a *Trazas de Estados Concretos*.

$$\gamma : AS \rightarrow CS$$

Ahora, para un input ι y su cs_ι, podemos computar el *Estado Abstracto* que corresponde, aplicando α como $\alpha(cs_\iota)$. A esto lo vamos a llamar IAS.

$$IAS(\iota, \alpha) = \alpha(cs_\iota)$$

La tupla formada por un input (ι) y un IAS, compone un *Resultado de Fuzzing* (FR) de ι.

$$FR(\iota, \alpha) = (\iota, IAS(\iota, \alpha))$$

Para un conjunto de inputs I y una *Función de Abstracción* α, el conjunto de *Resultados de Fuzzing* es llamado *Conjunto de Resultados de Fuzzing* (FRS).

$$FRS(I, \alpha) = \{FR(\iota, \alpha), ...|\forall\iota \in I\}$$

Para el conjunto de inputs I y un conjunto de *Funciones de Abstracción*, se define el *Resultado de Fuzzing Completo* (CFR) como:

$$CFR(I, A) = \{FR(\iota, \alpha), ...\ |\forall\alpha \in A, \iota \in I\}$$

Dado un conjunto de *Funciones de Abstracción* A, dos inputs ι_a y ι_b; son considerados equivalentes si y solo si el *Resultado de Fuzzing Completo* es igual, es decir:

$$\iota_a \equiv \iota_b \iff CFR(\{\iota_a\}, A) = CFR(\{\iota_b\}, A)$$

Con estas definiciones podemos, formalmente definir una técnica de fuzzing $\hat{F}$ como una función que toma los siguientes argumentos:

- El set de *Funciones de Abstracción* para ser usados por la iteración actual (A_{curr}).
- El programa p_{curr} que será probado, con instrumentación adicional como sea necesario para la *Función de Abstracción* actual A_{curr}.

- El conjunto de *Resultado de Fuzzing Completo* de todos los inputs previamente probados y *Funciones de Abstraccion* (CFR_{prev}).
- El conjunto de inputs que serán utilizados en la iteración actual (I_{curr}).
- El tiempo y recursos consumidos hasta el momento (tr_{curr}).

Y produce los siguientes resultados:

- Un set de inputs para ser usados en la siguiente iteración (I_{next}).
- Un set de *Funciones de Abstracción* para ser usados en la proxima iteración (A_{next}).
- Una versión del programa (p_{next}) con la instrumentación que A_{next} necesita.
- Un nuevo CFR que incluye el CFR de I_{curr}. Ej: $\{CFR_{prev} \cup CFR(I_{curr}, A_{curr})\}$
- El nuevo tiempo y recursos consumidos hasta el momento (tr_{next}).

Así nos queda que:

$$\begin{aligned} \hat{F} :&(p_{all}, I_{all}, P(A), P(CFR_{all}), t_{all}, r_{all}) \\ &\to (p_{all}, I_{all}, P(A), P(CFR_{all}), t_{all}, r_{all}) \end{aligned}$$

Donde p_{all} es el conjunto de todas las posibles copias funcionalmente idénticas al programa p, $P(A)$ es el conjunto *partes* de todas las posibles *Funciones de Abstracción*, $P(CFR_{all})$ es el conjunto *partes* de CFR sobre todos los inputs I_{all} y todos los posibles conjuntos de *Funciones de Abstracción* A

$$CFR_{all} = CFR(I_{all}, A)$$

t_{all} es el conjunto de todos los tiempos consumidos posibles y r_{all} es el conjunto de todos recursos consumidos posibles.

En cada iteración, un procedimiento de fuzzing usualmente guarda los inputs *interesantes* usados. Un input es *interesante* si explora un *Estado Abstracto* que no fue alcanzado por ninguno de los inputs anteriores. Sea $\tilde{I}$ todos los inputs *interesantes*, e $\iota \in I_{curr}$, se considera *interesante* si:

$$\exists \alpha \in A_{curr} | \alpha(cs_\iota) \not\subseteq_\alpha \bigcup_{i \in \tilde{I}} \alpha(cs_i)$$

Donde A_{curr} es el conjunto de las *Funciones de Abstracción* usadas en esta iteración.

Lo que quiere decir que existe una *Función de Abstracción* tal que aplicada a la *Traza de estados Concretos* de ι, esta alcanza un *Estado Abstracto* que no fue alcanzado por ningún input anterior.

El proceso de Fuzzing $\bar{F}$ aplica la técnica $\hat{F}$ iterativamente hasta que los recursos son consumidos. Un proceso de Fuzzing $\bar{F}$ es, dado un programa p, un conjunto de inputs iniciales I_{init} (Conocido como semillas), un conjunto de *Funciones de Abstracción* iniciales A_{init}, una técnica de fuzzing $\hat{F}$, una cuota de tiempo t y una cuota de recursos r. Un proceso de fuzzing devuelve el conjunto de inputs que alcanzan estados inválidos del programa I_{inv}.

$$\bar{F} : (p_{all}, I_{all}, \hat{F}_{all}, t_{all}, r_{all}) \to I_{all}$$

El proceso de fuzzing $\bar{F}$ aplica $\hat{F}$ iterativamente hasta que la cuota de tiempo o de recursos se acaban.

$$\hat{F} \circ \hat{F} \circ \cdots \circ \hat{F} \circ \hat{F}(p, I_{init}, A_{init}, \emptyset, 0, 0)$$

Dada esta notación de fuzzing, es posible definir cualquier fuzzer describiendo 3 elementos:

- **Generado de Input** (I_{next}): Este describe como un nuevo input será generado.
- **Función de Abstracción** (A): El conjunto total de *Funciones de Abstracción* que pueden ser usadas en el proceso de Fuzzing.
- **Selector de Funciones de Abstracción** (A_{next}): Similar al generador de inputs, este describe el mecanismo usado para seleccionar la *Función de Abstracción* que será utilizada en la siguiente iteración.

La mayor parte de los fuzzers, como el resultante de este trabajo, poseen solo una *Función de Abstracción* para ser usada en todas las iteraciones ($\|A\| = 1$). A estas técnicas las llamaremos *Técnicas de Única Abstracción* (SAF).

2.2. Ejecución Simbólica

Ejecución Simbólica Clásica

La *Ejecución Simbólica* es una técnica de análisis de programas nacida en los años 70's utilizada para detectar si una propiedad es infringida en software.

En una ejecución concreta, un programa corre sobre un input específico, produciendo un solo camino de ejecución y un resultado. En contraste a esto, una ejecución simbólica, puede explorar en simultáneo múltiples caminos del programa que pueden ser tomados por diferentes input.

En este trabajo utilizaremos la notación presentada en el paper *A Survey of Symbolic Execution Techniques* [1].

La ejecución es realizada por un *intérprete Simbólico*, que por cada camino de ejecución guarda:

- Una fórmula Booleana de primer orden que describe las condiciones que el camino actual satisface para tomar las ramificaciones necesarias.
- Una *Memoria Simbólica* que mapea variables a *Estados Simbólicos.*

Y eventualmente utiliza un *SMT solver* para comprobar que la *fórmula* es satisfacible y por lo tanto que el camino actual es alcanzable.

Para poder realizar esto, se toma cada valor que no puede ser determinado por análisis estático (ej: el input del programa, el resultado de una llamada al sistema) como un símbolo α_i. En todo momento el *intérprete Simbólico* mantiene un estado de la ejecución $(stmt, \sigma, \pi)$ donde:

- *stmt* es la proxima instrucción a ser ejecutada.
- σ es el estado simbólico que que asocia variables a una expresión sobre valores concretos o valores símbolos.
- π es la *Condición del Camino*, una fórmula que expresa todas las condiciones que el símbolo α_i debe cumplir para que la ejecución alcance *stmt*.

Dependiendo de la instrucción *stmt*, el *intérprete Simbólico* cambia el estado de la siguiente manera:

- La evaluación de una asignación $x = e$ actualiza el *estado simbólico* σ, asociando x a una nueva expresión simbólica e_s. Denotaremos esta asociación con $x \mapsto e_s$, donde e_s es la expreción obtenida al evaluar e en el contexto del estado actual del intérprete.
- La evaluación de un salto condicional `if` e `then` st_{true} `else` st_{false} afecta la *condición del camino* π. El *intérprete Simbólico* es ramificado, creando dos estados nuevos con *condiciones de camino* π_{true} y π_{false} que corresponden a las dos ramificaciones: $\pi_{true} = \pi \wedge e_s$ y $\pi_{false} = \pi \wedge \neg e_s$, donde e_s es la expresión obtenida evaluando e.

Una *Ejecución Simbólica* exhaustiva provee, en teoría, una métodologia *sólida* y *completa* por cada análisis. La *solidez* impide falsos negativos y la *completitud* falsos positivos.

Una *Ejecución Simbólica* exhaustiva es posible en programas pequeños, pero en programas reales es complicado o imposible de escalar, debido a los desafíos que esta conlleva; tales como: manipular y guardar tipos de datos complejos, manejar la interacción con el entorno (llamadas a librerías, archivos, etc), crecimiento exponencial de caminos y resolución de fórmulas. En particular para este trabajo, las dos últimas son de interés:

- *Crecimiento exponencial de caminos*: Este desafío, también llamado *path explotion*, se debe a la *completitud* del método, cada vez que se alcanza una bifurcación en el programa, se toman ambos caminos y así sucesivamente, esto produce que el número de estados de ejecución aumente exponencialmente.
- *Resolución de fórmulas*: Los *STM solvers* pueden escalar para resolver ecuaciones complejas con múltiples fórmulas y miles de variables. Sin embargo construir esas aritméticas no lineales genera un mayor desafío para la eficiencia.

Ejecución Simbólica Dinámica

La *Ejecución Simbólica Dinámica* o *Ejecución Concólica* (llamada de esta forma por *Ejecución* ***conc****reta* y *Ejecución simb****ólica***), es una técnica que consiste en realizar una *ejecución simbólica* guiada por una *ejecución concreta*.

Esta técnica nace para solucionar las limitaciones de eficiencia en La *Ejecución Simbólica* clásica, tales como: reducir los recursos utilizados en resolver ecuaciones con un *SMT solver*, prevenir analizar toda la pila de software y no solo el programa en sí y el desafío previamente nombrado como *Crecimiento exponencial de caminos*.

En esta técnica, además del *estado simbólico* y las *condiciones de camino*, el intérprete, guarda un *estado concreto* σ_c. En vez de elegir un input arbitrario para comenzar el programa, ejecuta el programa de manera simbólica y concreta en simultáneo. Cuando se encuentra un divergencia en el camino, la ejecución toma el camino que seguiría la ejecución concreta y el intérprete guarda la *condición de camino* que se tomó. De esta forma no es necesario utilizar un *SMT solver* para encontrar una solución a las ecuaciones para tomar cada dirección.

Para lograr una mayor cobertura del programa, se puede negar una o más *condiciones de camino* y obtener una solución del *SMT solver* para generar un nuevo input y volver a ejecutar el programa, y de esta forma tomar un camino nuevo.

Ejemplo:

Dada la siguiente función, que no tiene otra utilidad más que inducir un error en la línea 5, realizaremos una ejecución concólica con el objetivo de alcanzar dicho error:

```
1          void vuln(int a, int b) {
2              int x = b * 2;
3              if (x == a)
4                  if (x > b + 10)
5                      abort();  // Error
6          }
7
```

Definimos, de forma arbitraria, las entradas del programa que nos darán el primer estado como:

$$\sigma_c = \{a \mapsto 22, b \mapsto 7\}$$
$$\sigma = \{a \mapsto a_s, b \mapsto b_s\}$$
$$\pi = \{\}$$

Donde σ_c es el estado concreto, σ es el estado simbólico de la ejecución, π son todas las *Condiciones de Camino* y a_s es el resultado de evaluar la expresión a.

Al interpretar la línea número 2, se crea una variable nueva y se le asigna `b * 2`, por lo que se obtiene el siguiente estado:

$$\sigma_c = \{a \mapsto 22, b \mapsto 7, x \mapsto 14\}$$
$$\sigma = \{a \mapsto a_s, b \mapsto b_s, x \mapsto b_s * 2\}$$
$$\pi = \{\}$$

Ahora, en la línea 3, tenemos una condición que es `x == a`, que su evaluación tiene valor `Falso` y de esta forma, la ejecución termina con un estado y una nueva condición de camino:

$$\sigma_c = \{a \mapsto 22, b \mapsto 7, x \mapsto 14\}$$
$$\sigma = \{a \mapsto a_s, b \mapsto b_s, x \mapsto b_s * 2\}$$
$$\pi = \{b_s * 2 \neq a_s\}$$

Como el objetivo es llegar al error, con la ayuda de un *SMT Solver*, se niega la condición de camino π y se encuentra un estado inicial que cumpla la condición de la primer sentencia `if`.
Suponiendo que el *solver* dice que para cumplir la condición $(b_s * 2 = a_s)$ el estado inicial debe ser:

$$\begin{aligned} \sigma_c &= \{a \mapsto 2, b \mapsto 1\} \\ \sigma &= \{a \mapsto a_s, b \mapsto b_s, \} \\ \pi &= \{\} \end{aligned}$$

Al evaluar hasta la tercer línea de código, se obtiene el siguiente estado:

$$\begin{aligned} \sigma_c &= \{a \mapsto 2, b \mapsto 1, x \mapsto 2\} \\ \sigma &= \{a \mapsto a_s, b \mapsto b_s, x \mapsto b_s * 2\} \\ \pi &= \{b_s * 2 = a_s\} \end{aligned}$$

Por lo que el programa toma el camino para alcanzar al error. De todas formas, en la siguiente línea del programa se evalúa otra condición que no cumple y se debe volver a realizar los mismos pasos para, por último, obtener el estado inicial con el cual se alcanza el error.

Capítulo 3

Not Yet Another Fuzzer

En este capítulo veremos el resultado de este trabajo, *Not Yet Another Fuzzer* (*NYAF*)[5]. Un fuzzer basado en emulación y guiado por Ejecución Simbólica Dinámica. Capaz de realizar pruebas de fuzzing sobre binarios compilados para distintas arquitecturas y plataformas, logrando una abstracción del entorno natural de ejecución de los programas a analizar.

Para mostrar dicho fuzzer, se dividió este capítulo en 3 secciones. En la primera se verá una pequeña introducción de alto nivel de NYAF, luego se formalizará el algoritmo principal de fuzzing y por último se verán detalles de implementación.

3.1. El fuzzer

Como fue presentado anteriormente, *NYAF* nace de la necesidad de ampliar las funcionalidades del fuzzer *SAGE* y poder realizar pruebas en binarios (sin tener el código fuente) de otras arquitecturas, en particular, ARM. Con esa premisa, por las características propias de *SAGE* podemos encasillar nuestro fuzzer como *White-box*, debido a que utiliza ejecución simbólica para obtener información sobre el comportamiento del programa para generar nuevos casos de prueba.

Algunas de los objetivos fundamentales del fuzzer son:

- Realizar pruebas de fuzzing en programas compilados, sin el código fuente.
- Funcionar con programas compilados para distintas arquitecturas, en particular ARM y x86 (32 y 64 bits), y distintos sistemas operativos o en ausencia de estos.
- Lograr una dependencia del entorno de ejecución del programa, no solo de los puntos previamente vistos, sino que también de los periféricos del hardware, como LEDs y periféricos de red.
- Facilitar la rápida configuración sin perder funcionalidades en la misma.

3.2. Algoritmo de fuzzing

Como fue visto en el capítulo anterior, es posible definir cualquier fuzzer SAF describiendo 2 parametros: el Generador de Input (I_{next}) y la *Función de Abstracción*.

En esta sección se describirá nuestro fuzzer SAF definiendo las siguientes dos componentes:

3.2.1. Función de Abstracción

Definiremos la *Función de Abstracción* α_{sage} como:

$$\alpha_{sage}(\iota) = \{(bb_i^1, bb_j^1, log_2(n_1)), (bb_i^2, bb_j^2, log_2(n_2)), ..., (bb_i^m, bb_j^m, log_2(n_m))\} \tag{3.1}$$

Donde $(bb_i^*, bb_j^*, log_2(n_*))$ son pares de *Bloques Básicos* (bb) tales que bb_j^* es visitado después de bb_i^* n veces cuando el programa procesó el input ι.

De esta forma podremos identificar a los input interesantes como aquellos que alcanzan por lo menos un nuevo par de *Bloques Básicos* que no fueron alcanzados previamente.

3.2.2. Generación de Inputs: *Búsqueda Generacional*

Al igual que $SAGE$[8], nuestro fuzzer implementa el algoritmo de *Búsqueda Generacional* sobre la Ejecución Simbólica Dinámica para generar nuevos casos de test.

Este algoritmo de búsqueda fue diseñado para explorar de forma *sistemática*, pero *parcial*, grandes programas con caminos de ejecución profundos, tratando de maximizar la cantidad de nuevos casos de test por corrida. Además usa heurísticas para maximizar la cobertura lo más rápido posible y es resistente a las divergencias.

En esta sección se explicará como funciona este algoritmo.

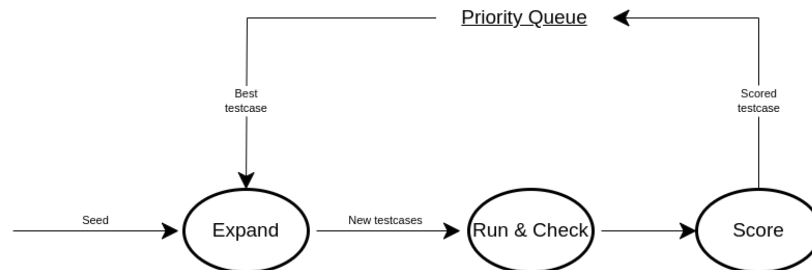

Figura 3.1: Loop del algoritmo de Búsqueda Generacional

Como podemos ver en la figura 3,1, el algoritmo es principalmente un bucle con 3 funciones principales: *Expand*, *Run & Check* y *Score*. Las mismas serán explicadas a continuación.

Al comienzo del proceso de fuzzing, se le da una semilla o *seed* a la función *Expand*, la cual se encargará de generar nuevos casos de prueba. Como es el primer ciclo del proceso, el caso de prueba inicial o semilla, será el único y diremos que pertenece a de la generación 0.

Luego, estos nuevos casos de prueba serán evaluados con el programa original para comprobar si incumplen alguna de las propiedades del programa con la función *Run & Check*, si es así, se notifica al usuario. Al terminar de comprobar, con la función *score* se le asigna un puntaje a cada caso de prueba y se guardan en una cola de prioridades (*Priority Queue*).

Al final y para cerrar el ciclo, se selecciona el caso de prueba con mayor puntaje de la cola de prioridades y se lo da como argumento a la función *expand*, tal como se hizo con la semilla, y se continúa con el ciclo.

El pseudo código del ciclo se vería de la siguiente forma:

```
1   def search(seed):
2       seed.generation = 0
3       work_list = [seed]
4       run_and_check(seed)
5
6       while work_list is not empty:
7           best_input = work_list.pop()
8           child_inputs = expand(best_input)
9
10          for child in child_input:
11              run_and_check(child)
12              score(child)
13              work_list.append(child)
14
```

Expand

La función *Expand* es la principal en el ciclo de fuzzing. Esta es la encargada de generar nuevos casos de prueba utilizando el algoritmo de búsqueda generacional.

```
1  def expand(input):
2      child_input = {}
3      path_const = compute_path_constraint(input)
4
5      for j in range(input.generation, |path_const|):
6          if (pat_copnst[0..(j-1)] and not pat_const[j])
       has a solution I:
7              new_input = input + I
8              new_input.generation = j
9              child_inputs = child_inputs + new_input
10
11     return child_inputs
12
```

Como se muestra en el pseudo-código, esta función recibe un caso de prueba y devuelve una lista de nuevos casos generados a partir de éste.

Para esto, lo primero que hace es ejecutar el programa bajo prueba con el input pero de forma *concólica*, lo que nos permitirá obtener cada *Condición de Camino* que se tomo.

Luego de esto, por cada *Condición de Camino* a partir de la que esté en la posición que coincida con el número de generación del caso de prueba, se niega la misma y con ayuda de un *SMT sovler* se comprueba si la ecuación obtenida tienen solución. Si es así, se crea un nuevo caso de prueba con esta solución y se agrega a la lista de nuevos casos. Esta cota inferior con el valor de la generación del caso de prueba impide que se trate de generar un caso que ya fue generado previamente.

Tomemos el siguiente programa como ejemplo:

```
1        void vuln(char input[4]) {
2            int cnt = 0;
3
4            if(input[0] == 'b') cnt ++;
5            if(input[1] == 'a') cnt ++;
6            if(input[2] == 'd') cnt ++;
7            if(input[3] == '!') cnt ++;
8
9            if(cnt >= 4) abort();  // Bug
10       }
11
```

vuln es una función escrita en *C* en la cual se simulará un error que se manifiesta dadas ciertas condiciones, utilizando la función *abort*. *vuln* toma como argumento un arreglo de 4 caracteres y la condición de fallo es que el mismo sea igual a `['b', 'a', 'd', '!']`.

Como se comentó anteriormente, en la primer corrida de la función *expand*, se utiliza la *semilla*, en este caso, se usa `['g', 'o', 'o', 'd']` como muestra la siguiente figura. En ella, cada punto es una *Condición de Camino* y la siguiente flecha para la izquierda significa que la condición tiene valor *falso* en esta ejecución, para la derecha lo contrario.

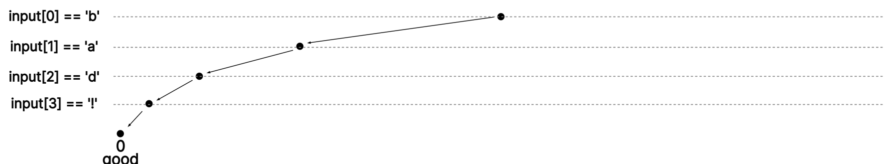

Figura 3.2: Ejecución del algoritmo con la semilla.

Como es la primer ejecución de la función, se niega cada una de las *Condiciónes de Camino* y se le da cada una al *STM Solver* para obtener nuevos casos de prueba que formarían la *primer generación*. En este caso se obtienen:

- `['b', 'o', 'o', 'd']`
- `['g', 'a', 'o', 'd']`
- `['g', 'o', 'd', 'd']`
- `['g', 'o', 'o', '!']`

Y tras volver a ejecutar el programa con estos nuevos casos de prueba, obtenemos:

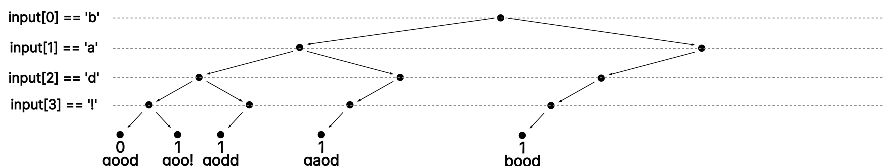

Figura 3.3: Primera generación de inputs.

Se vuelve a realizar el mismo procedimiento, pero esta vez se niegan solo las condiciones luego de la primera, ya que negar la primer condición no generará ningún caso de prueba nuevo, y obtenemos la *segunda generación*.

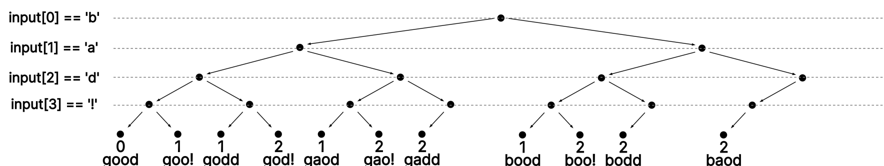

Figura 3.4: Segunda generación de inputs.

Se repite el procedimiento negando todas las condiciones luego de la segunda.

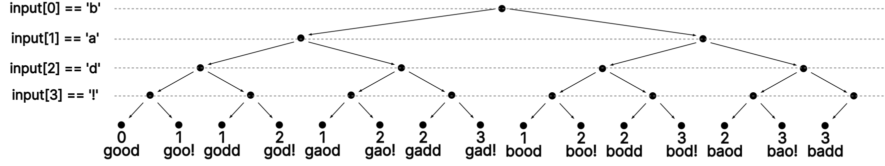

Figura 3.5: Tercera generación de inputs.

Y por último, negando solo la última condición obtenemos el resultado esperado, el caso de prueba que alcanza el error, `['b', 'a', 'd', '!']`.

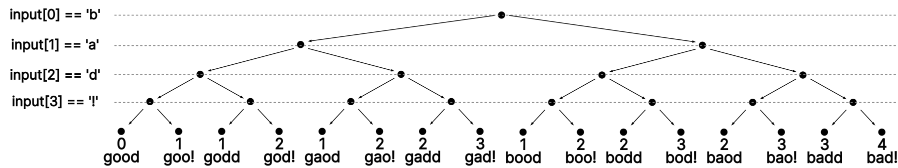

Figura 3.6: Cuarta y última generación de inputs.

Score

La función *score* es la encargada de ponerle un puntaje o ponderación a cada caso de prueba nuevo, de forma que en cada ciclo de fuzzing, se le adjudique mayor puntaje o score a los casos de prueba con más probabilidades de descubrir nuevos caminos.

Para puntuar un caso de prueba, se utilizó la cantidad de nuevos *Bloques Básicos* alcanzados por el mismo. Para esto se tiene que llevar un registro de todos los *Bloques Básicos* alcanzados por todos los casos de prueba anteriores.

Con este método de *scoring*, se favorecerá a la ejecución de la función *expand* sobre los casos de prueba que alcanzan nuevas partes del programa, y así, tener mayor probabilidades de aumentar la cobertura de la prueba.

Formalmente, podemos definir *score* como:

$$score(\iota_i) = \left| \, bb(\iota_i) - \bigcup_{n=0}^{i-1} bb(\iota_n) \, \right| \tag{3.2}$$

Donde $bb(\iota)$ son todos los *Bloques Básicos* alcanzados por ι.

Superficie y vector de ataque

Si bien un fuzzer puede tener múltiples superficies de ataque, tales como: línea de comandos, sistemas de archivos, protocolos de red y más, en este trabajo, con el fin de abarcar una mayor cantidad de objetivos de distintas naturalezas, tomaremos a todas las funciones como superficie de ataque.

Además, el vector de ataque será la inyección de datos en tiempo de ejecución, a través de un buffer de datos en memoria y punteros, sobrescribiendo los argumentos de la función objetivo.

3.3. Implementación

En esta sección se hará una descripción de alto nivel de la implementación de *NYAF*. Las herramientas utilizadas, las optimizaciones aplicadas y la razón de las mismas.

NYAF fue escrito completamente en *Python* por tres razones. La primera fue la compatibilidad con librerías y herramientas seleccionadas, la segunda, para hacer que la extensibilidad del fuzzer sea simple y fácil para la comunidad, ya que *Python* es el lenguaje más utilizado por la comunidad de *InfoSec*, y la tercera, para que el fuzzer sea multiplataforma.

3.3.1. Herramientas

Emulación

Para lograr el objetivo de crear un fuzzer que soporte múltiples arquitecturas y entornos distintos, la única alternativa es utilizar emulación. Podemos definir emulación como la imitación de un entorno en el cual los binarios se ejecutan.

En este trabajo nos interesan dos tipos de emulación: *emulación de CPU* y *emulación de sistema*.

- **Emulación de CPU**

  En este caso, como el nombre lo indica, se trata de imitar el comportamiento del CPU físico utilizando uno virtual, con un programa que interprete las instrucciones y guarde el estado de la ejecución (registros, banderas, memoria, periféricos) del programa paso a paso.
  Cabe destacar que no es suficiente para emular un programa como los que utiliza un usuario, ya que en general un programa corre sobre un sistema operativo.

- **Emulación de sistema**

  A un nivel superior, encontramos la emulación de sistema, que trata de imitar el comportamiento del entorno en el cual el programa se ejecuta, como el sistema operativo, imitando o redireccionando *llamadas al sistema*, interfaces de red y más.

En el caso del fuzzer, se necesitaba realizar una emulación completa del CPU y el sistema, imitando y encapsulando el entorno de ejecución. Esto se consiguió utilizando un framework de emulación llamado *Qiling*[6].

Este *framework* de código abierto que, a diferencia de otros emuladores, es una herramienta de análisis, nos permite realizar la emulación proporcionando una *API* muy expresiva a nivel de sistema operativo y de máquina virtual en *Python*, de tal forma que facilita la instrumentación de los binarios y el control de la ejecución.

Ejecución Simbólica Dinámica

Para realizar una ejecución concólica, analizar posibles caminos de ejecución y resolver condiciones de camino, se utilizó *Triton*[9], una libreria de análisis dinámico escrita en *Python*. Esta, no solo genera y guarda las condiciones de camino, sino que permite de forma fácil resolverlas con un *SMT Solver*.

3.3.2. Sanitizer

El objetivo del fuzzer es encontrar errores en el código, pero aunque varios tipos de ellos se exponen como *Core Dumps* (ej: lectura de memoria inválida), fallos en del CPU (ej: división por cero) y demás formas que terminan la ejecucin del programa, otros, lo hacen de una forma silenciosa (ej: escritura de memoria valida). Para descubrir los últimos, se utilizan los *Sanitizers*, ellos obligan a producir un error visible a partir de uno silencioso.

Existen distintos tipos de *Sanitizers*, pero en este trabajo solo se cubrirán los *Address Sanitizers* o de direcciones de memoria, que se encargan de los errores en el *Heap* del programa.

En particular se verá cómo afrontar tres tipos de errores: *Heap Overflow/Underflow*, *Use After Free* y *Double Free*.

Cuando un programa necesita un espacio de memoria de gran tamaño o necesita que éste no sea volátil, recure al *Heap*, un lugar de memoria manejado generalmente por librerías estándares del sistema operativo o llamadas al sistema. Por lo general son dos las funciones más importantes para manejarlo:

- **alloc**: Se encarga de separar un espacio de memoria para el programa. Cuando se llama a *alloc*, se debe especificar un tamaño de memoria para guardar y la función devuelve el puntero al espacio de memoria guardado.
- **free**: Libera un espacio de memoria previamente separado para que pueda volver a ser usado en otro momento.

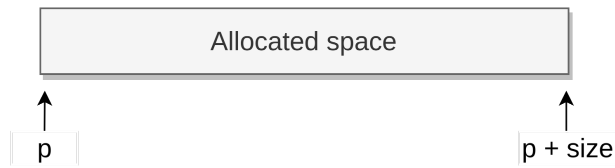

Figura 3.7: Representación de una llamada a *alloc*. Sea *p* el puntero al espacio de memoria separado por *alloc* y *size* el tamaño especificado.

Heap Overflow

Este error se da cuando se quiere escribir o leer por fuera de la memoria asignada. Por ejemplo, se esta escribiendo en un *buffer* de memoria obtenida con un *alloc* y no se definieron bien los limites, por lo cual, se sigue escribiendo luego del final de este espacio. Para solucionarlo, lo que el *sanitizer* hace es:
Cada vez que se llama a *alloc*, se cambia el tamaño de memoria requerido por el mismo más 8 bytes. Luego, se marcan los primeros 4 y los últimos 4 bytes como protegidos contra lectura y escritura y, por último, se devuelve como resultado de la llamada a la función el puntero incrementado en 4.

De esta forma el programa no nota diferencia alguna, pero si se trata de escribir o leer por fuera del espacio guardado se producirá un error.

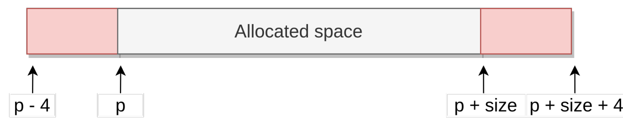

Figura 3.8: Representación de una llamada a *alloc* con el *sanitizer*. Sea *p* el puntero al espacio de memoria separado por *alloc* y *size* el tamaño especificado. El espacio de memoria en rojo es aquel que esta protegido contra lectura y escritura.

Use After Free

Este error se da cuando el programa intenta usar memoria que fue guardada con *alloc* y luego liberada con *free*, posibilitando errores como *leaks* de memoria. Para evitar esto, cada vez que se llama a la función *free*, el *sanitizer*, no libera esta memoria, sino que la marca como protegida. Esto produce un aumento en la memoria utilizada por el programa bajo análisis pero evita que se vuelva a usar ese espacio de memoria.

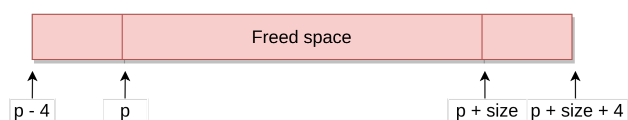

Figura 3.9: Representación de una llamada a *free* con el *sanitizer*. Sea p el puntero al espacio de memoria previamente separado por *alloc* y *size* el tamaño especificado. El espacio de memoria en rojo es aquel que esta protegido contra lectura y escritura.

Double Free

Este error se da cuando se quiere liberar con *free* más de una vez un mismo espacio de memoria, lo que puede producir comportamientos inesperados en el programa.

Como el *sanitizer* no libera realmente los espacios de memoria, esto es solucionado fácilmente al llevar la cuenta de todos los espacios de memoria actualmente separados por *alloc* y en cada llamada a *free* verificando que el espacio que se quiere liberar no fue liberado previamente.

3.3.3. Ciclo de Ejecución

Llamaremos *Ciclo de Ejecución* al ciclo en el cual el fuzzer utiliza los casos de prueba sobre el sujeto bajo analísis para encontrar errores.

Ciclo normal

El ciclo esperado de un fuzzer es el siguiente:

1. Se comienza como la emulación del programa.
2. En algún punto de la ejecución, definido previamente, se inyecta el caso de prueba generado.
3. El programa termina normalmente o con un error.
4. Se vuelve al paso 1.

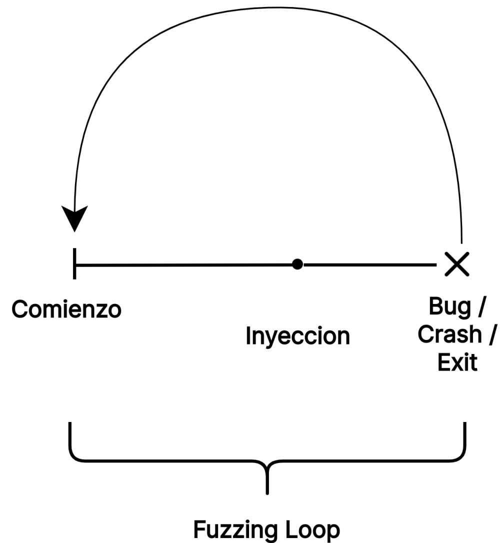

Figura 3.10: Ciclo de Ejecución normal.

Como podemos ver en la figura, hay un lapso de ejecución entre el comienzo y la inyección que se repite en cada ocasión, pero no es necesario. Para optimizar esto, se utilizó el conceptio de *Snapshot*.

Snapshot

Un *Snapshot* no es más que una imagen de todo el sistema (Registros, memoria, sistema de archivos, etc.) que se guarda para luego ser restaurada y comenzar la ejecución desde ese punto.
De esta forma, el *Ciclo de Fuzzing* nos queda de la siguiente forma:

1. Se comienza como la emulación del programa.
2. Se toma un *Snapshot* del sistema en un punto definido previamente.
3. En algún punto de la ejecución, definido previamente, se inyecta el caso de prueba generado.
4. El programa termina normalmente o con un error.
5. Se restaura el sistema y se vuelve al paso 3.

Ésto evita tener que ejecutar el programa desde el comienzo por cada caso y así poder optimizar el tiempo de ejecución.

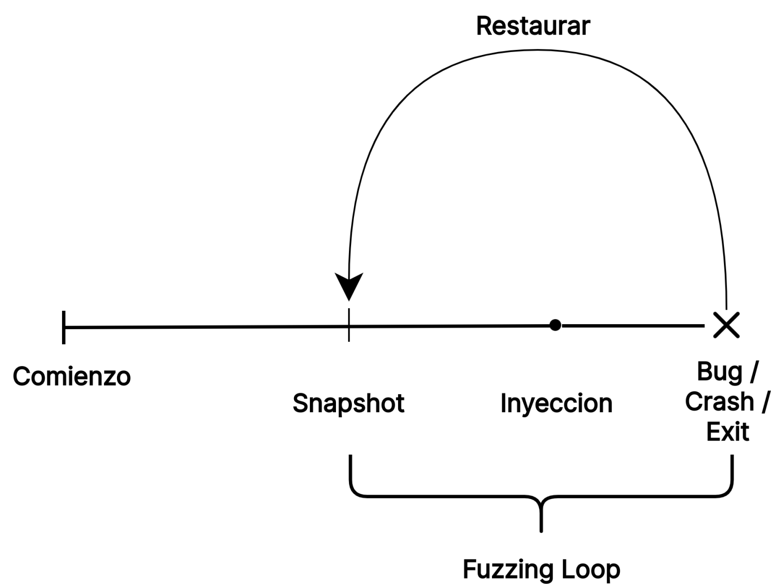

Figura 3.11: Ciclo de ejecución con restauración de *snapshot*.

Dependencia del hardware

En el comienzo de este trabajo se planteó hacer una disociación entre entorno de ejecución y el programa mismo, con el fin de realizar pruebas de Fuzzing en programas dependientes del entorno en el que normalmente se ejecuta. Por ejemplo, un programa que depende del hardware, ya sea inicializando o utilizando recursos del mismo.

Por esta razón se agrega una posible división en el ciclo de fuzzing:
Primero una etapa de inicialización donde el programa es ejecutado hasta el punto de *snapshot* en el entorno normal, y segundo, la etapa del ciclo de fuzzing propiamente dicho.

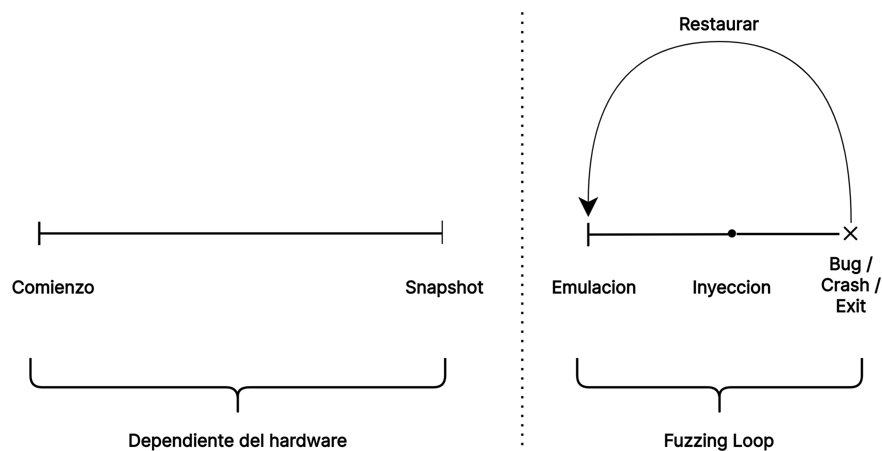

Figura 3.12: Disociación de la dependencia del hardware.

De esta forma, la sección dependiente del hardware o del entorno de ejecución propio del programa, puede ser ejecutada en el mismo entorno. Mientras que la parte independiente, puede ser emulada dentro de las pruebas de fuzzing.

3.3.4. Paralelismo

Para lograr una mayor eficacia en el uso de la cantidad de recursos computacionales disponibles, se implementaron algunas mejoras en el algoritmo.
En primer lugar se crearon distintos tipos de procesos:

- *Runners*: Los responsables de emular un programa, utilizados como base para la implementación de *Runners* más específicos.
  - *Coverage Runner*: Emula un programa y recolecta información sobre la cobertura de dicha emulación.
  - *Symbolic Runner*: Emula un programa y recolecta las condiciones de camino. Tiene la capacidad de producir nuevos casos de prueba a partir del primero.
- *Valuator*: Encargado de obtener el *score* de una ejecución.

De esta forma queda el siguiente diagrama:

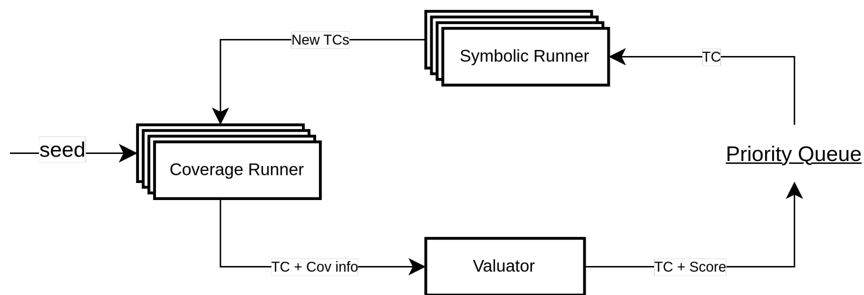

Figura 3.13: Ciclo optimizado para múltiples procesos. Donde *TC* es un caso de prueba, *Score* es la valuación del caso de prueba, *Cov info* es la información de cobertura del caso de prueba y *seed* es una semilla.

Aquií se puede observar que hay varios procesos del tipo *Coverage Runner* y *Symbolic Runner* pero un solo *Valuator*.

Toda la Comunicación Inter Procesos (IPC) está implementada con *Queues* y mensajes TCP en el *localhost*, pero como esto excede los objetivos de este trabajo, no se profundizada al respecto.

Capítulo 4

Caso de estudio

Para poder poner a prueba las capacidades de *NYAF*, se planteó un caso de estudio lo suficientemente simple como para facilitar su entendimiento, pero suficientemente complejo como para poder demostrar las ventajas del fuzzer.

En este caso de prueba, se plantea un escenario con dos actores:

- Atacante: Quien será el encargado de vulnerar el servidor utilizando *NYAF*.
- Servidor (o víctima): El objetivo del atacante, al que se tratará de vulnerar.

Específicamente, el servidor será un programa que recibe conexiones TCP/IP y procesa los mensajes que se le manden con el fin de proporcionar una interfaz capaz de interactuar con el hardware del dispositivo, en particular, con los LEDs. Dicho programa es compilado y ejecutado en un dispositivo *RaspberryPi 3B+*[4], que posee un procesador *ARM* de 64 bits. Al contrario, el dispositivo del atacante será una computadora personal con un procesador de arquitectura *X86* de 64 bits.

Para hacer esta prueba más acorde a un caso del mundo real, se utilizará solamente el binario compilado de dicho servidor, por lo cual, se deberá hacer un pequeño trabajo de ingeniería inversa. Por otro lado, para facilitar el análisis, que no es el objetivo principal del trabajo, se utilizará una conexión *SSH* con el dispositivo víctima para poder ejecutar el programa y leer la salida del mismo por la consola.

Dividiremos este capitulo en 3 partes:

- *Análisis*: Se realizará un análisis del objetivo en cuestión, comenzando desde una vista de alto nivel y terminando con el análisis del ejecutable mismo.
- *Preparaciones*: Se verá cómo se configura el fuzzer y todo lo que se debe tener en cuanta previamente a la prueba de fuzzing.
- *Fuzzing*: Por último, se realizarán las pruebas y se analizarán los resultados de la misma.

4.1. Análisis

En primer lugar se conecta al dispositivo que ejecutará el servidor por *SSH*, que en nuestro caso, se encuentra en la dirección IP *192.168.0.86* dentro de nuestra red, y se procede a ejecutar el servidor para hacer unas pruebas básicas.

Dichas pruebas consisten en mandar commandos al servidor por el puerto especificado, utilizando la herramienta *netcat*.

A continuación se pueden apreciar los resultados de las terminales, tanto dentro del dispositivo víctima como el atacante, corriendo en simultáneo.

Terminal del dispositivo víctima

```
$ ssh 192.168.0.86
    Linux raspberrypi 5.15.61-v8+ #1579 SMP PREEMPT
   Fri Aug 26 11:16:44 BST 2022 aarch64
    ...

$ sudo ./server/server 8181
    Hello...

    [+] Hardware init finish.
    [*] Starting the server.
    [+] Server binded and listenning on port: 8181

    [*] New connection.
    [+] Recv: LED ON
    [+] Led turned ON

    [*] New connection.
    [+] Recv: LED OFF
    [+] Led turned OFF

    [*] New connection.
    [+] Recv: BAD CMD
    [X] Bad cmd.
```

Terminal del dispositivo atacante

```
$ nc 192.168.13.2 8181
LED ON

$ nc 192.168.0.86 8181
Enter cmd: LED ON

$ nc 192.168.0.86 8181
Enter cmd: LED OFF

$ nc 192.168.0.86 8181
Enter cmd: BAD CMD
```

Con esto, se llega a la conclusión de que el servidor acepta una conexión a la vez y pretende que por cada una de estas conexiones se mande un comando. Dichos comandos son:

- *LED ON*: Para prender el LED.
- *LED OFF*: Para apagar el LED.
- Todo comando que no sea ninguno de los anteriores: Es tomado como error y descartado.

Se puede ver la anterior interacción con el servidor en le siguiente diagrama de secuencia:

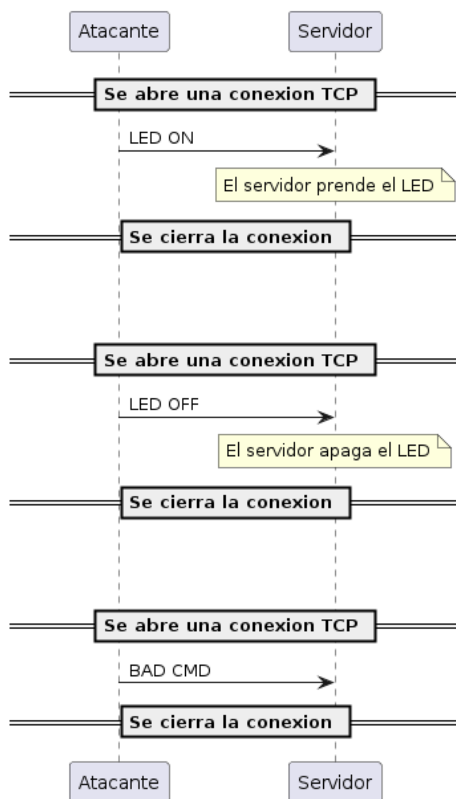

Figura 4.1: Comunicación del atacante y el servidor.

El siguiente paso en el análisis es obtener el binario y tratar de ver el formato. Para lo primero se utilizó la herramienta *scp* para copiar el binario (*'./server'*) desde el dispositivo víctima al atacante y luego, utilizando *Rizin*, una herramienta de análisis de binarios, se obtiene más información:

```
$ rz-bin -I ./server

[Info]
arch     arm
bintype  elf
bits     64
class    ELF64
compiler GCC: (Debian 10.2.1-6) 10.2.1 20210110
endian   LE
intrp    /lib/ld-linux-aarch64.so.1
lang     c
machine  ARM aarch64
os       linux
...
```

Con esto se puede ver que está compilado para la arquitectura ARM de 64 bits y en un sistema operativo Debian, lo que coincide con la descripción inicial del caso de estudio.

Paso siguiente, se procede a desensamblar el binario y estudiar la función principal del programa, *main*, utilizando *Cutter*, una interfaz gráfica para *Rizin*. Para facilitar, mostraremos una versión decompilada de la función:

```
1  void main(int argc, char **argv)
2  {
3      uint32_t uVar1;
4      char *arg1;
5      int64_t arg1_00;
6      int64_t arg2;
7      char *recv_buff;
8      server svr;
9      int size;
10
11     // int main(int argc,char ** argv);
12     puts("Hello...");
13     hardware_init();
14     puts("[*] Starting the server.");
15     arg1 = (char *)atoi(argv[1]);
16     arg1_00 = server_init(arg1);
17     arg2 = calloc(0x400, 1);
18     do {
19         uVar1 = serve(arg1_00, arg2);
20         operate(arg2, (uint64_t)uVar1);
21     } while( true );
22 }
```

Lo importante en este código desensamblado es que se puede ver cómo funciona la estructura principal del servidor. Primero, se inicializa el hardware en la línea 13, luego se inicializa el servidor propiamente dicho en la línea 16, se obtiene un espacio de memoria en el heap en la línea 17 como *buffer* de $0x400$ bytes y por último, en un ciclo infinito, se llama a *serve* y con el resultado se llama a *operate*.

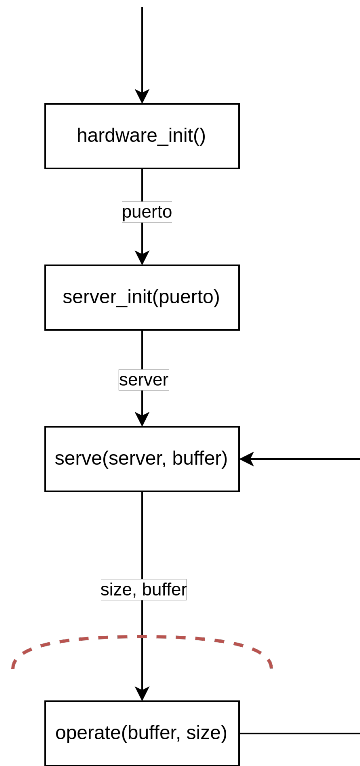

Figura 4.2: Función *main*.

Por su parte, la función *serve* nos da el siguiente código decompilado:

```
1 int32_t serve(int64_t arg1, int64_t arg2)
2 {
3     undefined4 uVar1;
4     int32_t iVar2;
5     int size;
6     int connfd;
7     char *recv_buff;
8     server svr;
9
10    // int serve(server svr,char * recv_buff);
11    uVar1 = accept(*(undefined4 *)arg1, 0, 0);
12    puts("[*] New connection.");
13    write(uVar1, "Enter cmd: ", 0xb);
14    iVar2 = read(uVar1, arg2, 0x800);
15    *(undefined *)(arg2 + iVar2) = 0;
16    printf("[+] Recv: %s\n", arg2);
17    close(uVar1);
18    return iVar2;
19 }
```

Esta funcón toma dos argumentos, un puntero a *server* y un puntero a un buffer. Primero, acepta una nueva conexión, lee hasta $0x800$ bytes en el buffer y cierra la conexión. En resumen, esta función sólo acepta conexiones y recibe datos en un buffer.

Se observa que el tamaño máximo de lectura es superior al tamaño del buffer, y esto resulta en un *Heap Buffer Overflow*.
Pero dejamos este error fuera del alcance de esta prueba.

Por último, la función *operate* nos da el siguiente código decompilado:

```
1 void operate(int64_t arg1, int64_t arg2)
2 {
3     int64_t iVar1;
4     int32_t iVar2;
5     uint64_t uVar3;
6     int size;
7     char *recv_buff;
8
9     // void operate(char * recv_buff,int size);
10    iVar1 = _CMD_LED_ON;
11    uVar3 = len(_CMD_LED_ON);
12    iVar2 = cmp(arg1, iVar1, uVar3 & 0xffffffff);
13    iVar1 = _CMD_LED_OFF;
14    if (iVar2 == 0) {
15        led_set(1);
16    } else {
17        uVar3 = len(_CMD_LED_OFF);
18        iVar2 = cmp(arg1, iVar1, uVar3 & 0xffffffff);
19        iVar1 = _CMD_CUSTOM;
20        if (iVar2 == 0) {
21            led_set(0);
22        } else {
23            uVar3 = len(_CMD_CUSTOM);
24            iVar2 = cmp(arg1, iVar1, uVar3 & 0
   xffffffff);
25            if (iVar2 == 0) {
26                vuln();
27            } else {
28                puts("[X] Bad cmd.");
29            }
30        }
31    }
32    return;
33 }
```

La herramienta *Cutter* nos muestra en el comentario de la línea 9 que la función toma como argumento dos elementos, el primero, un buffer de tipo `char *` o puntero a caracteres y un segundo elemento del tipo `int` que será usado como

indicador del tamaño del buffer. Luego, realiza una serie de comparaciones entre el buffer y variables globales (líneas: 12, 18, 24) para comprobar si son iguales (líneas: 14, 20, 25) y realizar, según el resultado de las comparaciones, algunas llamadas a funciones: `led_set(1)` (línea 15), `led_set(0)` (línea 21), `vuln()` y `puts("[X] Bad cmd.")` (línea 28).

Al ser la función que procesa las entradas del programa, se seleccionó como objetivo de fuzzing.

Podemos ver en la figura *4.2* una línea punteada roja, esta indica el vector de ataque, que será el *buffer* que se le da como entrada a la función *operate*.

4.2. Preparaciones

4.2.1. Snapshot

Para hacer una abstracción del hardware en el proceso de fuzzing, como fue propuesto en los objetivos, se dividió la obtención del snapshot en dos etapas. La primera etapa dentro del dispositivo víctima y la segunda en la del atacante.

Coredump

La primera etapa consiste en en utilizar un *scrip* escrito en python llamado *dumper* que, solo usando gdb, nos permite realizar un *coredump* (captura de un proceso en ejecución) de un programa en una dirección de memoria especifica. Esto nos permite obtener una captura del proceso en ejecución dentro del hardware específico.

```
 $ ./dumper.py -h
     usage: dummper [-h] pid address output

     NYAF process dumper tool.

     positional arguments:
     pid          pid of the target process (int).
     address      Address where to take the dump (
hex).
     output       Filename for the generated dump.

     optional arguments:
     -h, --help  show this help message and exit

 $ sudo ./dumper.py $(pgrep server) 0xf0c ./server.
core
     Preparing to generate the core dump.
     [*] Target pid: 1285
     [*] Target address: 0xf0c
     [*] Process base address: 0x558fd20000
```

```
[*] Waiting to reach the address 0x558fd20f0c
[+] Core dump saved in ./server.core
Bye...
```

core2snap

Luego de obtener el *coredump* se procede a, en la computadora atacante, convertir el *coredump* obtenido en un *snapshot* con un formato que el fuzzer pueda reconocer. Esto se logra utilizando otro *script*, también escrito en python, llamado *core2snap*.

```
$ ./core2snap.py -h
    usage: core2snap [-h] input output

    NYAF core dump to qiling snapshot conversi\'on
   tool.

    positional arguments:
    input       Core dump input file.
    output      Qiling snapshot output file.

    options:
    -h, --help  show this help message and exit

$ ./core2snap.py ./server.core ./server.snap
    Conversi\'on initializing
    [*] Convertig ./server.core and saving in ./server
   .snap.
    [+] Registers found (34):
    {<REGISTERS.AARCH64_X25: 88>: 0,
    ...
    [+] Registers converted (33):
    {'pc': 367485128460,
    ...
    [+] Memory mapping found:
    (367485124608, 367485132800, 5, '/home/tantrum/eko
   /server/server')
    ...
```

Ambos scripts están en el repositorio de *NYAF*.

4.2.2. Configuración

El siguiente paso para poder realizar la prueba de Fuzzing es crear la configuración para la prueba de Fuzzing propiamente dicha.

En nuestro caso, con NYAF, se debe crear un archivo en el formato *YAML* y ahí especificar la configuración de la siguiente forma:

```
cmd: ./server/server 8181
rootfs: ./fuzzing/rootfs/

fuzzing:
  strategy: generational_search
  runners: 1
  symbolics: 4

seeds:
  - ./fuzzing/seed

snapshot: ./fuzzing/server.snap
start_address: 0xf0c
stop_address:
  - 0xc84
  - 0xff0

injection:
  method: function
  address: 0xf0c
```

En primer lugar se especifican las configuraciones de ejecución del programa:

- *cmd*: especifica el comando que ejecutara el fuzzer para poder correr el programa por primera vez.
- *rootfs*: la dirección al directorio el cual el programa verá como raíz del sistema ('/').

Luego se realizan las configuraciones de la estrategia de fuzzing:

- *strategy*: estrategia de fuzzing a utilizar (Configuración pensada para una ampliación en las capacidades del fuzzer), en nuestro caso, búsqueda generacional.
- *runners*: cantidad de procesos utilizados como *Coverage Runner*.
- *symbolics*: cantidad de procesos utilizados como *Symbolic Runner*.

En la línea 18 se especifican las configuraciones de inyección. Dicha inyección será de método *function* en la dirección de memoria de la función *operate*. Este método, por defecto, inyecta el caso de prueba como un puntero a un buffer como primer argumento y el tamaño del buffer como segundo parametro, justo como los argumentos de la función a atacar.

Por último, tres configuraciones sobre la ejecución:

- *snapshot*: Dirección donde se encuentra el snapshot a utilizar.
- *start_address*: Dirección de memoria donde empezar la emulación, en este caso, el mismo que el snapshot y la función *operate*.
- *stop_address*: Lista de direcciones de memoria donde detener la ejecución. En nuestro caso lo utilizamos tanto como para separar la utilización del hardware como para optimizar la emulación. `0xc84` es la dirección de *led_set* y (0xff0) es la dirección del prologo de la función *operate*.

4.3. Fuzzing

Teniendo definido el objetivo de la prueba de Fuzzing y la configuración lista, se puede avanzar con la prueba.

La ejecución de *NYAF*, según lo descripto en su apartado de ayuda, es la siguiente.

```
$ nyaf -h
    USAGE: /usr/bin/nyaf [ config file ] [ output
 file ]
```

Por lo que solo se debe dar como argumento el archivo de configuración y el directorio donde guardar todos los casos de test que produzcan un error en el programa.

```
$ nyaf ./fuzzing/config.yml ./fuzzing/out/
```

Luego de aproximadamente 12 segundos, el fuzzer fue capaz de encontrar un error, tal como podemos ver en la siguiente secuencia de mensajes a consola.

```
    ...
[Runner 0 (coverage) ] Running: b'Aguante el Fernet!'
[   CoverageRunner   ] Restoring snapshot from: ./
   fuzzing/server.snap
[   CoverageRunner   ] Running with payload: b'Aguante
    el Fernet!'
[   CoverageRunner   ] Running from 0x558fd20f0c to 0
   x0
[PosixAddressSanitizer] Attached
[Runner 0 (coverage) ] BUG FOUND (EXCEPTION): 0xe4c
[       Fuzzer       ] Stopping the fuzzer
[       Fuzzer       ] Fuzzer stopped
[      Manager       ] Manager stopped
[      Manager       ] Manager stopped
[        NYAF        ] ===============  Bye
   ===============
```

Por último, comprobamos el archivo que creó el fuzzer con la entrada que hace fallar al programa y se lo enviamos al servidor.

```
$ nc 192.168.0.86 8181 < ./fuzzing/out
   /3477455037053629443
```

Finalmente, podemos ver que el servidor terminó de ejecutarse, retornando un código de error (255).

```
$ sudo ./server/server 8181
    Hello...
    [+] Hardware init finish.
    [*] Starting the server.
    [+] Server binded and listenning on port: 8181
    [*] New connection.
    [+] Recv: Aguante el Fernet!
    2023-02-02 19:33:34 sigHandler: Unhandled signal
   11, terminating

$ echo $?
    255
```

Capítulo 5

Conclusiones

En este capitulo haremos un análisis sobre el caso de estudio práctico previamente presentado, para poder obtener las conclusiones del mismo y presentar un conjunto de lineamientos para trabajos futuros.

5.1. Análisis de Resultados

Búsqueda Generacional

En cuanto al algoritmo seleccionado, para facilitar el análisis, se obtuvo el código fuente del caso de estudio:

```
1     char *CMD_LED_ON = "LED ON\0";
2     char *CMD_LED_OFF = "LED OFF\0";
3     char *CMD_CUSTOM = "Aguante el Fernet!\0";
4
5     int cmp(char *s1, char *s2, unsigned long size) {
6     for (unsigned long i = 0; i < size; i++)
7         if (s1[i] != s2[i])
8         return 1;
9
10    return 0;
11    }
12
13    void vuln(void) { *(int *)0 = 0; }
14
15    void operate(char *recv_buff, int size) {
16        if (cmp(recv_buff, CMD_LED_ON, len(CMD_LED_ON)
      ) == 0)
17            led_set(true);
18
19        else if (cmp(recv_buff, CMD_LED_OFF, len(
      CMD_LED_OFF)) == 0)
20            led_set(false);
```

```
21
22          else if (cmp(recv_buff, CMD_CUSTOM, len(
    CMD_CUSTOM)) == 0)
23              vuln();  // ERROR
24
25          else
26              printf("[X] Bad cmd.\n");
27      }
28
```

Aquí se puede observar que el error fue introducido en la función *vuln*, la cual hace una asignación del valor 0 en la dirección de memoria 0, que es una escritura de memoria inválida. Así mismo, esta función, es llamada en la línea número 23, si y solo si, el buffer que se le dio a *operate* tiene el mismo valor que la variable *CMD_CUSTOM*.

Lo que se puede deducir de esto es que, desde un caso base, el algoritmo de búsqueda fácilmente fue construyendo byte a byte el caso de prueba que llevaba a cubrir la función *vuln*, cuyo valor era el mismo de *CMD_CUSTOM*.

Para un fuzzer que genere casos de prueba aleatorios o simplemente sin tener conocimiento alguno de la semántica del programa, sería muy difícil o casi imposible lograr obtener los 18 bytes que logran alcanzar el error.

En particular, suponiendo que el fuzzer genera casos de prueba aleatorios con tamaño entre 1 y 18 bytes, tendría $\sum_{i<19}^{i} 255^i$ posibles casos. Esto, sumado a la sobrecarga de procesamiento que produce la emulación, no permitiría una aplicación práctica de dicho fuzzer.

Por otro lado, el algoritmo seleccionado utiliza ejecución concólica para generar casos de test y esto no es computacionalmente sencillo. Pero, lo que resulta más contraproducente en la frecuencia de casos de prueba, es la resolución de las ecuaciones con los *SMT Solver*.

Sin hacer anáisis en profundidad, podemos llegar a la conclusión de que el fuzzer ocupa más del 50 % del tiempo resolviendo ecuaciones.

Independencia del entorno

Otro de los objetivos principales planteados al inicio del trabajo fue la independencia del entorno, lo que hace referencia a separar el hardware, sistema operativo y demás componentes del entorno, de las pruebas de fuzzing.

En particular, se usaron dos mecanismos para lograrlo: *Emulación* y *Snapshot*.

Emulación

Como se hizo anteriormente, se puede dividir la emulación en dos partes:

- Emulación de CPU
- Emulación del sistema

Con la primera, se logró hacer una abstracción e independencia de la arquitectura del procesador, pudiendo emular un programa compilado para *ARM* de 64 bits en un CPU de arquitectura *X86_64*.
Y la segunda, nos permitió abstraernos del sistema operativo, simulando las llamadas al sistema y la carga del binario, esto no se vio reflejado en la prueba.

Snapshot y rangos de ejecución

Por otro lado, se requería una independencia de los componentes del hardware que no podían ser emulados, por ejemplo, los LEDs en la Raspberry Pi.

Para esto, en primer lugar, se utilizó la captura de *Snapshot* desde el dispositivo en el que se ejecutaba el programa y así poder evitar la emulación de la inicialización de los periféricos. Además, se puso una restricción en la emulación para que la misma termine si se llegaba a tratar de interactuar con los LEDs, ya que esta parte del programa quedaba fuera del rango objetivo.

5.2. Conclusión

Con el análisis previo se puede concluir que el algoritmo seleccionado tuvo éxito, cumpliendo el objetivo propuesto: producir la mayor cobertura posible utilizando la menor cantidad de casos de prueba. Esto logra una mejora en la eficiencia del fuzzer reduciendo el tiempo de emulación, pese al tiempo y recursos invertidos en la ejecución simbólica y en la resolución de ecuaciones por el *SMT solver*.

Así mismo, se pudo observar que el algoritmo fue capaz de encontrar errores que otros métodos aleatorios no permitirian encontrar fácilmente.

Por otra parte, se pudo comprobar en el caso de estudio que, a través de los los mecanismos propuestos (emulación, *snapshot* y limites de ejecución), se logró una gran independencia con respecto al entorno de ejecución. De este modo, es posible abstraer el análisis de la arquitectura, plataforma y hardware para el cual el binario fue compilado, logrando los objetivos propuestos en un comienzo.

5.3. Trabajo futuro

Luego del precedente que marca este trabajo, se pueden definir líneas de trabajos futuros con el fin de continuar con la investigación sobre el tema, tales como:

- Analizar el uso de otros algorímos con menor costo computacional para la generación de casos de test junto con *Búsqueda Generacional*, utilizando una heurística que decida cual usar en cada momento. De forma similar a como fue realizado en *Driller*[10].
- Analizar el uso de un modelo de aprendizaje automático para decidir cuales son los casos de prueba más importantes, en reemplazo de la función de *score*. Tal como fue realizado en *River*[2].
- Expandir el fuzzer para permitir la emulación de periféricos o elementos del hardware definidos por el usuario y comparar los resultados con el fuzzer actual.
- Estudiar un caso de uso en el cual, el binario objetivo, haya sido compilado para la plataforma *Windows*, y la prueba se realice sobre la plataforma *Linux*.

Apéndice A

NYAF - Repositorio

El repositorio de *NYAF* se puede encontrar en la dirección,
https://gitlab.com/nyaf/fuzzer/. En este podemos encontrar, entre otras cosas, lo siguiente:

- Un archivo con nombre *README.md*, con todas las explicaciones de cómo instalar, configurar y usar el fuzzer, además de un ejemplo básico.
- Un directorio con nombre *examples*, donde habrá binarios de distintas arquitecturas y plataformas con sus respectivas configuraciones a modo de ejemplo.
- Un directorio con nombre *src*, donde se ubica el código del fuzzer.
- Un directorio con nombre *utils*, donde se encuentran los scripts utilizados en el caso de estudio, que son:
  - *dumper.py*: utilizado para generar un *coredump* de un programa en ejecución.
  - *core2snap.py*: utilizado para generar el *snapshot* desde un *coredump*.

Bibliografia

[1] Roberto Baldoni et al. “A Survey of Symbolic Execution Techniques”. En: *ACM Comput. Surv.* 51.3 (2018).

[2] A. Stefanescu BC. Paduraru M. Paduraru. “RiverFuzzRL — An Open-Source Tool to Experiment with Reinforcement Learning for Fuzzing.” En: *In Proc. of 14th IEEE Conference on Software Testing, Verification and Validation (ICST'21), pp. 430-435, IEEE* (2021).

[3] A. Doupe C. Salls Aravind Machiry et al. “Exploring Abstraction Functions in Fuzzing”. En: *IEEE Conference on Communications and Network Security* (2020).

[4] Raspberry Pi Foundation. “Raspberry Pi 3 Model B+”. En: (). URL: `https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-Model-Bplus-Product-Brief.pdf`.

[5] Aznarez Rojo Gaston. “NYAF: Not Yet Another Fuzzer”. EkoParty. 2022. URL: `https://youtu.be/GuLXg1R-hNs`.

[6] LAU kaijern. *Qiling*. URL: `https://qiling.io/` (visitado 26-02-2023).

[7] Microsoft. “Microsoft Security Bulletin MS07-017 - Critical”. En: (). URL: `https://learn.microsoft.com/en-us/security-updates/securitybulletins/2007/ms07-017` (visitado 26-02-2023).

[8] David Molnar Patrice Godefroid Michael Y. Levin. “Automated Whitebox Fuzz Testing”. En: (1012).

[9] quarkslab. *Triton*. URL: `https://triton-library.github.io/` (visitado 26-02-2023).

[10] Nick Stephens et al. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution”. En: (2016).