






# HOUSE: Marco de trabajo modular de arquitectura escalable y desacoplada para el uso de técnicas de *fuzzing* en HPC

Francisco Borja Garnelo Del Río , Francisco J. Rodríguez Lera , Gonzalo Esteban Costales ,  
Camino Fernández Llamas , Vicente Matellán Olivera   
Universidad de León - Campus de Vegazana s/n, 24071 León (Spain)  
infbgd01@estudiantes.unileon.es, {fjrodl, gestc, cferll, vmato}@unileon.es

## Resumen

El *fuzzing* es una técnica de prueba automatizada que hace uso de mutaciones de entradas para ejecutar el software utilizando estas a fin de observar los valores de retorno y el estado externo del sistema; todo ello para identificar comportamientos no esperados. Por su simpleza y fácil automatización inicial, el *fuzzing* es una de las técnicas más populares para identificar vulnerabilidades en software en el mundo real. El uso de técnicas de *fuzzing* como parte del desarrollo de software en grandes compañías ha acelerado la evolución de las técnicas y mejorado las herramientas, como contrapartida muchos proyectos quedan huérfanos de sus desarrolladores originales, al ser captados por estas empresas y todas aquellas características con potencial uso comercial, son restringidas al ámbito privado o son dependientes de nubes propietarias. Esto supone una barrera inicial para nuevos desarrollos, una posible dependencia tecnológica de terceros y problemas de confidencialidad de los datos para adoptar modelos seguros de desarrollo o realizar investigaciones. Las investigaciones sobre *fuzzing* y las nuevas herramientas tienen en común partir desde cero o una base teórica y focalizarse en una característica o funcionalidad. Por todo lo anterior resulta valioso desarrollar un marco de trabajo común que de coherencia y continuidad de forma global abordando el *fuzzing* como un flujo con distintas fases, organizando todos los procesos implicados, incluyendo también aquellos con relación directa, de forma modular, abierta y agnóstica a las herramientas y técnicas. Con ello se potencia la reutilización y se elimina la dependencia a herramientas o casos de uso, a la vez que permiten sinergias con otros enfoques de seguridad en el software. El uso de una arquitectura desacoplada y escalable permite la distribución y paralelización de trabajos. Esta arquitectura es ideal para entornos de computación de alto rendimiento en adelante HPC (high-performance computing por su traducción al inglés) o basados en cargas de trabajo. El código fuente y la documentación empleada para la prueba de concepto se encuentra disponible públicamente en GitHub.

**Index Terms**—Desarrollo seguro, marco de trabajo, *fuzzing*, análisis de datos, adaptación de código, HPC, seguridad del software, automatización pruebas software, SDL, AFL++, Slurm

## I. INTRODUCCIÓN

El *fuzzing* [1] es la automatización de generación y prueba de entradas malformadas en el software con el fin de encontrar comportamientos no esperados en el mismo, habitualmente

*crashes*<sup>1</sup>. Este término se usó por primera vez en un estudio de la seguridad de UNIX en la década de 1990 [3], [4].

Hace poco más de una década, las herramientas de *fuzzing* eran poco más que generadores de ruido aleatorio. En la actualidad hay nuevas técnicas capaces de entender en mayor medida la semántica y los flujos de ejecución del software de forma automatizada. Algunos ejemplos son la generación de firmas [5], la ejecución simbólica dinámica [6], las pruebas de complejidad [7], la generación de casos de prueba basados en gramáticas [8] y las pruebas de comportamiento [9].

Las últimas investigaciones han avanzado mucho en las capacidades de generación de entradas y en el análisis de las estructuras internas del software; ambas con el fin de evolucionar características o funcionalidades de las herramientas de *fuzzing*. Gran parte del éxito de estas investigaciones ha venido de la mano de aplicar los últimos avances en el análisis de datos [10] y algoritmos de aprendizaje automatizado [11] o de su uso específico para la búsqueda de vulnerabilidades<sup>2</sup> de seguridad [13]. Como respuesta a su uso como herramienta de búsqueda de vulnerabilidades, también están surgiendo técnicas *antifuzzing* [14].

Respecto a su uso a gran escala fuera del ámbito privado, se han realizado experimentos de su uso en HPC [15] y es posible replicar experimentos parecidos en entornos cloud compatibles con las herramientas y arquitecturas HPC como AWS [16], Azure [17] o GCS [18]. No obstante, el *fuzzing* no está exento de contratiempos a la hora de su uso o aplicación.

La complejidad y la vasta cantidad de información existente sobre el *fuzzing* es el primer problema que se presenta. Esto suele derivar en que muchos de estos avances son proyectos separados que sólo son implementados como pruebas de concepto de forma aislada, sin ser integrados con otras mejoras, u optimizaciones de forma general, que acaban siendo descontinuados o parcialmente integrados en herramientas *open source* [19].

<sup>1</sup>*crash* [2] Fallo repentino y completo de un sistema o componente informático.

<sup>2</sup>*Vulnerabilidad* [12] La existencia de una debilidad, diseño o error de implementación que puede llevar a un evento inesperado e indeseable que comprometa la seguridad del sistema informático, red, aplicación o protocolo involucrados.

El segundo problema se debe a la monopolización de las herramientas e investigaciones hacia objetivos comerciales o de uso interno por parte de la industria u organizaciones. Esto es especialmente relevante en aquellos proyectos orientados a utilizar técnicas de *fuzzing* a gran escala o como parte de un modelo de desarrollo seguro. Tales proyectos presentan serias limitaciones para ser adaptados a distintos entornos diferentes de aquellos para los que fueron creados; ya sea debido a su origen privativo [20], [21] o a su enfoque comercial [22], [23].

Esto, en caso de productos comerciales, presenta serias limitaciones de cara a la privacidad dado que implica el acceso al código fuente <sup>3</sup> y el uso de *Software as a Service* (SaaS) en entornos de nube en alguna de sus modalidades. Respecto a aquellos proyectos de origen privativo, las limitaciones suelen venir por la dificultad de aprendizaje de la solución y por su adaptación a un entorno diferente, con esfuerzos reiterados en cada nueva evolución de la solución original.

La tendencia generalizada a utilizar infraestructuras en la nube, promovidas en gran parte por las mismas organizaciones que derivan el desarrollo de las herramientas al ámbito privado, impactan en una pérdida de autonomía y confidencialidad de los datos. Sirva de ejemplo un proveedor de nube específico [24] o la toma de decisiones de forma unilateral por el prestador de servicio [25].

Un tercer problema es debido a la dificultad y especialización necesarias para poner en marcha el *fuzzing* de forma que este aporte valor sin un esfuerzo recurrente y pueda aplicar los resultados de forma efectiva o reutilizar los esfuerzos previos realizados para casos similares.

Realizar pruebas de *fuzzing* en software es una tarea compleja puesto que requiere preparar [26] el código fuente para facilitar la automatización de entradas. Los protocolos de red o los componentes del núcleo de los sistemas operativos son dos ejemplos de software que necesitan una preparación previa para lograr obtener resultados efectivos.

En los casos donde la entrada de datos presente una complejidad elevada (como por ejemplo reproductores multimedia, intérpretes de lenguajes o protocolos de comunicaciones) implica un proceso previo de generación de entradas [26]. Para ello es necesario preparar las entradas iniciales o semillas utilizando ficheros con estructuras complejas o programables, como pueden ser los documentos XML, JavaScript, etc.

De forma generalizada a todo proceso de *fuzzing* es necesaria una especialización para el aprovechamiento de los resultados para por ejemplo identificar puntos de mejora de seguridad<sup>4</sup> o en la calidad del software.

Estos problemas pueden ser abordados sobre una estructura ordenada y orientada a las fases comunes del *fuzzing*, con un diseño modular de componentes que permita la reutilización

e integración de distintas soluciones. El marco de trabajo HOUSE que se presenta en este trabajo propone una aproximación que permite mantener la confidencialidad del código fuente, garantizando la escalabilidad y funcionamiento independiente de cada uno de sus módulos para evitar esfuerzos recurrentes. Además, facilita la investigación de cada uno de los componentes que forman parte del proceso de *fuzzing* y la adopción de estas técnicas en proyectos de desarrollo. Esto reduce la barrera inicial y además proporciona una estructura modular consistente que permite simplificar las tareas de integración, preparación-ejecución de pruebas y análisis de resultados. Debido a que brinda una separación funcional, con automatizaciones para su puesta en funcionamiento e integrado en una organización superior preparada para la paralelización y reutilización.

Los principales objetivos de esta investigación son:

1. Conocer el estado actual del *fuzzing* enfocándolo a su uso en entornos computación de alto rendimiento.
2. Dar una visión global de las actividades previas, los procesos y componentes del *fuzzing* respecto a las últimas mejoras y las adaptaciones a entornos de computación escalable.

El resto del artículo se estructura de la siguiente manera. La Sección II relaciona las distintas disciplinas que convergen o tienen un punto común con el *fuzzing* (como son el desarrollo seguro de software y la seguridad), así como conceptos clave relacionados o que forman parte de su evolución. La Sección III desarrolla el flujo de trabajo en sus distintas fases atendiendo a los distintos procesos implicados. La Sección IV expone la experiencia de uso del marco de trabajo HOUSE en un entorno de supercomputación real. Por último, la Sección V detalla las conclusiones del trabajo. La documentación de uso del Marco de trabajo HOUSE esta disponible el repositorio de GitHub [27].

## II. CONTEXTO Y TRABAJOS RELACIONADOS

Con la evolución en complejidad y extensión de los sistemas informáticos y de comunicaciones, el software que los gobierna cuenta cada vez con más superficie susceptible de fallos, lo que a su vez supone una mayor dificultad para su búsqueda y prueba.

### II-A. Tipos de pruebas software

De una forma sencilla podemos reducir los tipos de pruebas realizadas al software en activas o dinámicas y pasivas o estáticas. Activas son aquellas pruebas que implican la ejecución total o parcial del software y las pasivas se centran en analizar el código fuente.

Dentro de las activas o dinámicas, atendiendo a un enfoque puramente de seguridad en el software [28], su identificación puede ser proactiva o reactiva.

Según nuestras evidencias, aparentemente no hay alternativas en la literatura al *fuzzing* desde un enfoque proactivo. El objetivo del *fuzzing* es identificar por medio de automatizaciones comportamientos no esperados en el software habitualmente en forma de *crashes*. Algunos de estos fallos

<sup>3</sup>**Código fuente** [2]: Instrucciones y definiciones de datos del ordenador expresadas en una forma adecuada para su introducción en un ensamblador, compilador u otro traductor. También considerada la versión legible para el ser humano de la lista de instrucciones [programa] que hace que un ordenador realice una tarea.

<sup>4</sup>**Seguridad** [12] Todos los aspectos relacionados con la definición, adquisición y mantenimiento de la confidencialidad de los datos, la integridad, la disponibilidad y la responsabilidad.

en el software pueden ser utilizados para comprometer la seguridad del sistema donde este es ejecutado o la información a la que tiene acceso. Debido a la automatización, es necesario acotar las pruebas a un método de entrada de datos específico en el software y limitarlas a un conjunto de funcionalidades que interactúen con las entradas de la manera más determinista posible. Ejemplos de *fuzzers* son herramientas como BFF [29], radamsa [30] o AFL [31].

Observando el comportamiento del software en un entorno de producción, es posible identificar de manera reactiva comportamientos no esperados. Estos por ejemplo pueden ser controles basados en contexto, como Security-Enhanced Linux (*SELinux*) [32] o Application Armor (*AppArmor*) [33], para distribuciones Linux o soluciones comerciales de seguridad basadas en la monitorización de eventos y protecciones [34], [35] siendo las más comunes combinaciones de Endpoint Detection and Response (*EDR*) y Endpoint Protection Platform (*EPP*).

Existen numerosas herramientas públicas para realizar pruebas automatizadas sobre el software de manera pasiva utilizando su código fuente. Como por ejemplo CodeQL [36] centrada en facilitar el análisis semántico del código fuente como si se tratase de consultas a una base de datos, u otras herramientas más tradicionales como los analizadores de código Sonarqube, Yasca o Flawfinder [37].

Para realizar análisis más completos es necesario combinar distintos enfoques como el proactivo por medio de *fuzzing* y el pasivo con herramientas que permitan minimizar o priorizar qué funcionalidades son susceptibles a tener más bugs<sup>56</sup>. Este es el caso cuando por complejidad no sea posible realizar pruebas con cobertura completa del alcance o cuando se requieren adaptaciones a medida para permitir la integración de las herramientas de pruebas con el software objetivo. La misma problemática se repite en el caso de querer utilizar infraestructuras de cómputo intensivo como supercomputadores o clusters en la nube [17].

Respecto a los datos utilizados en el proceso, hay algunos componentes de generación de datos de entrada, limpieza de falsos positivos y análisis de resultados, la mayoría teóricos o con una implementación muy dirigida.

Este escenario tan heterogéneo repleto de piezas aisladas dispares o rígidos componentes, hace difícil introducir de forma mantenible en el tiempo y de manera eficiente pruebas de *fuzzing* en el desarrollo y ralentiza las investigaciones de mejoras de los componentes del proceso.

## II-B. Tipos de fuzzing

Los tipos de *fuzzing* se clasifican en función de los detalles de la implementación disponible, estando claramente diferenciados dos tipos por la utilización del código fuente para la realización de las pruebas o el uso de software ya compilado

<sup>5</sup>**Bug** [2] Anomalía, defecto, error, excepción o fallo no controlado en un software.

<sup>6</sup>**Error** [38] La diferencia o discrepancia entre un valor o condición calculado, observado o medido y el valor o condición verdadero, establecido o teóricamente correcto.

en formato binario [28]. Estos dos tipos son el *white-box* fuzzing [39] y el *black-box* fuzzing [40] respectivamente.

Entre estos dos tipos se situaría un tercero, el *grey-box* fuzzing [26], que tiene características de ambos, como es el caso de las pruebas donde se utiliza instrumentación en el código fuente [41] y el análisis BVA (*Boundary Value Analysis*) [42] sobre los comportamientos observados en las ejecuciones del binario.

En la actualidad hay múltiples proyectos alrededor del *fuzzing*, desde el análisis de resultados [43], creación de conjuntos de datos de entrada de forma automatizada [44], aquellos específicos de tareas internas como optimizaciones [45] y creación de algoritmos de prueba [46]. Todos ellos necesitan, para poder ser completados de forma práctica, unas condiciones que en la mayoría de los casos exigen un esfuerzo de creación de un entorno completo de *fuzzing*. Similar caso sucede a desarrolladores que demanden poder realizar pruebas de calidad o seguridad en su software y quieran mantener la confidencialidad de su código fuente y evitar los esfuerzos de puesta en marcha de un entorno de *fuzzing* y realizar desde cero su integración o adaptación.

Las pruebas proactivas y activas con fines de calidad centradas en la seguridad [28] como el *fuzzing* están cada vez más presentes en los ciclos de vida de desarrollo de software SDLC (*Software Development Life Cycle*), siendo parte fundamental de los procesos para la implantación de un modelo de desarrollo seguro SSDLC (*Secure Software Life Cycle*) [47].

## III. MARCO DE TRABAJO HOUSE

HOUSE es un marco de trabajo con un enfoque funcional que permite organizar todas las actividades necesarias de cara a aplicar técnicas *fuzzing* sobre un proyecto software. Su nombre hace referencia a la clásica metáfora [48] de describir un proceso como el diseño y construcción de una casa y, en este caso, describe el proceso de desarrollo seguro del software pero desde el punto de vista del *fuzzing*.

En cuanto a sus características, HOUSE posee una arquitectura modular e interoperable cuyo funcionamiento se basa en aplicar el *fuzzing* a un proyecto software a través de un proceso compuesto por 4 fases, (ver Fig. 1):

1. **Común o de preparación del entorno de trabajo.** Es la única fase transversal a todo el proceso ya que se suele realizar una única vez, bien en la puesta en marcha de las herramientas necesarias para el *fuzzing* o bien durante el despliegue inicial del marco de trabajo.
2. **Prefuzzing o de preparación de las pruebas software.** En esta fase se elige el tipo de *fuzzing* a utilizar en función del grado de acceso y las necesidades de adaptación del código fuente. También admite los requisitos o preferencias del tipo de pruebas e incluso los resultados que se persigan con él. En función del tipo elegido, se preparan los binarios correspondientes: empleando binarios ya compilados o realizando una compilación específica si es necesario. En función del binario selec-

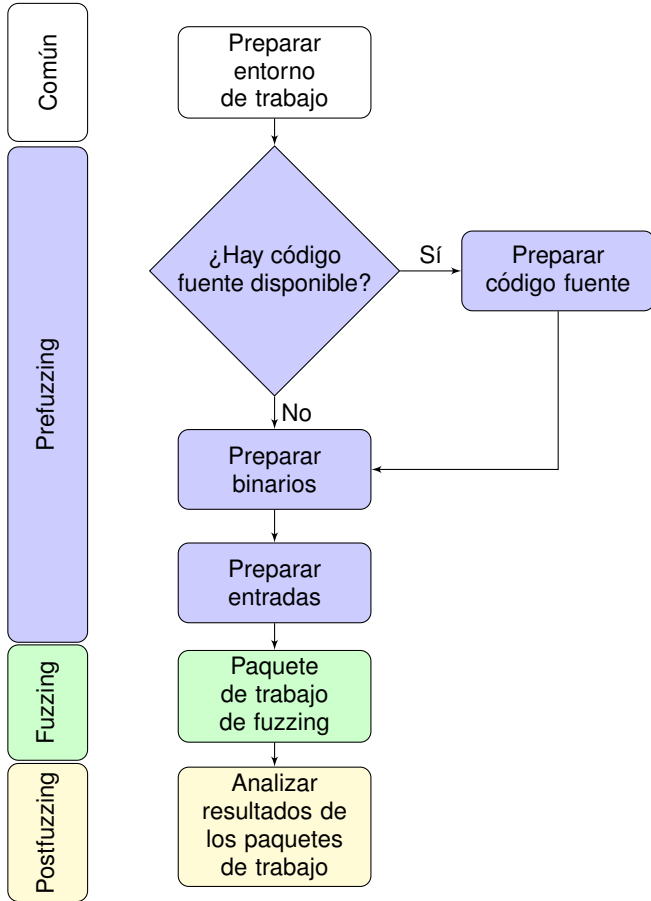


Figura 1. Diagrama de flujo para describir el funcionamiento de HOUSE según las diferentes fases de fuzzing.

cionado, también se confecciona o valida un conjunto de entradas para esa campaña de *fuzzing* [26].

3. **Fuzzing o de ejecución de las propias pruebas.** Fase en la que se somete el software bajo prueba (PUT - *Program Under Test*) al *fuzzing* utilizando la configuración realizada previamente.
4. **Postfuzzing o de análisis de resultados y tareas posteriores.** En esta última fase, bien si se desea o bien si por volumen de trabajo es necesario automatizar la tarea de análisis de resultados, es posible ejecutar un paquete de trabajo creado expresamente para poner en valor los resultados del *fuzzing*.

A su vez, dicho proceso trabaja sobre un conjunto de datos organizados en seis módulos, que pueden ser orquestados de manera escalable en un entorno de HPC por medio de cargas de trabajo agrupadas por paquetes:

1. **TooL Box (0.TLB).** Colección de herramientas y utilidades auxiliares a cualquiera de las fases del *fuzzing*.
2. **Source Code Integration (1.SCI).** Colección de códigos fuente, incluyendo parches, integraciones y optimizaciones previas al *fuzzing*.
3. **Binaries Ready to Fuzz (2.BRF).** Colección de binarios preparada para el *fuzzing*.

4. **WorkLoad Package (3.WLP).** Colección de paquetes con cargas de trabajo<sup>7</sup>, que pueden ser de cualquier fase del *fuzzing*, como herramientas que analicen el código fuente para identificar áreas más propensas a fallos, tareas propias de *fuzzing* o de analítica de resultados.
5. **Input Repository (A.IR).** Repositorio de entradas como semillas para generadores de entradas, diccionarios, archivos de muestra, etc.
6. **Output Repository (B.OR).** Repositorio de salidas con los contextos de ejecuciones de los *fuzzers*, resultados y cualquier información útil generada por el flujo de trabajo como logs.

Cabe señalar que cada módulo puede estar formado por componentes, unidades funcionales o una combinación de ambos. La principal diferencia entre dichos elementos radica en que las unidades funcionales realizan tareas de automatización o ejecución y los componentes son solo información. Por otro lado, los módulos identificados por letras (como **A.IR** y **B.OR**) están formados únicamente por datos, mientras que los identificados por números (**0.TLB**, **1.SCI**, **2.BRF** y **3.WLP**) tienen al menos una actividad que forma parte del flujo de *fuzzing*. En relación a estos últimos, el propio número indica su grado de dependencia dentro del proceso del *fuzzing*; siendo 0 la más baja y 3 la más alta.

Teniendo en cuenta todo lo anterior, esta estructura de fases junto a su modularización hacen que HOUSE promueva la reutilización de las diferentes partes del proceso de *fuzzing*; permitiendo así su escalabilidad o paralelización. Adicionalmente, dicha estructura también permite que el marco de trabajo sea agnóstico a las herramientas *fuzzing* empleadas. Por ejemplo, es posible cambiar el *fuzzer* utilizado sin tener que rehacer las correspondientes adaptaciones del código fuente, sin tener que preparar nuevos binarios o, incluso, sin tener que parar los trabajos en curso. De igual modo, también permite la reutilización de los conjuntos de datos de entrada creados para un proyecto concreto o el hecho de combinarlos con herramientas que los generen desde archivos de muestra.

### III-A. Flujo de trabajo y procesos asociados

La Fig. 2 representa el flujo de trabajo de HOUSE utilizando el lenguaje de modelado *Business Process Model and Notation* (BPMN) en su versión 2.0.2 [49]. La utilización de BPMN responde a la necesidad de modelar de extremo a extremo todo el flujo de *fuzzing* al detalle suficiente. De haber utilizado el lenguaje UML (*Unified Modeling Language*), el resultado hubiese sido menos preciso y más complejo al tener menos objetos representables y estar centrado en los flujos de datos.

Dicho lo anterior, en las columnas del diagrama BPMN de la Fig. 2 se sitúan los módulos de HOUSE y en las filas sus fases. Los siguientes apartados detallan individualmente esas fases dentro de cada módulo del marco de trabajo.

<sup>7</sup>Carga de trabajo [2]: Combinación de tareas que se ejecutan en un sistema informático determinado. Sus principales características es incluir los requisitos de entrada y salida, cantidad, tipo de calculo y computo requerido.

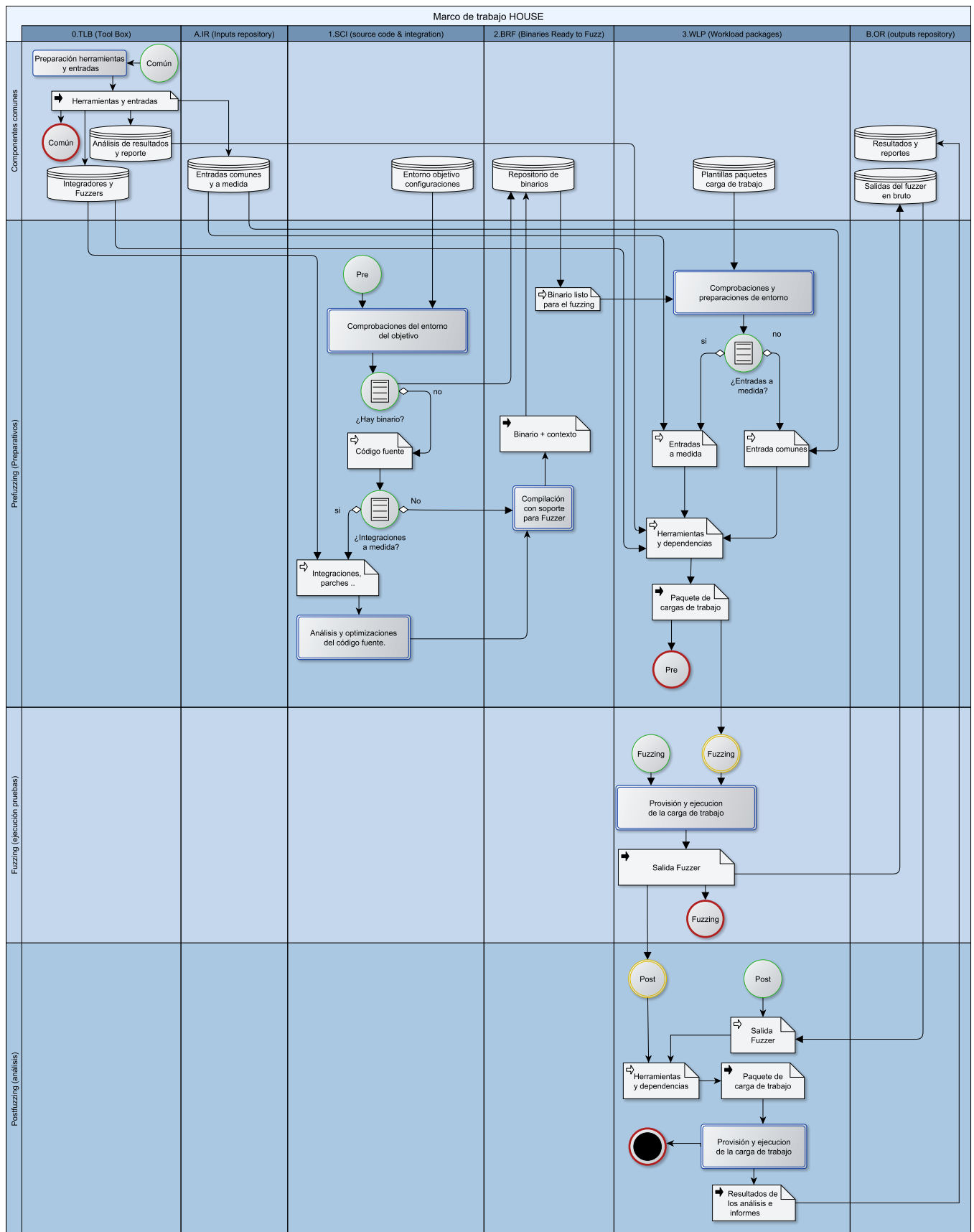


Figura 2. Diagrama BPMN del marco de trabajo HOUSE.

**III-A1. Común:** Esta fase aporta el esqueleto necesario para modelar el proceso de *fuzzing* dentro del marco de trabajo HOUSE. También define los elementos esenciales para las propias tareas de *fuzzing* o las adyacentes (como por ejemplo los analizadores de código, los generadores de entradas o el reporte de resultados). Dicho de otra manera, esta fase es responsable de habilitar—entre otros elementos—cualquier herramienta, librería o conjunto de datos de entradas que se vayan a utilizar en el entorno de ejecución de las pruebas. Todo esto se realiza tanto en el módulo **0.TLB** (para el caso de herramientas tales como *fuzzers*, analizadores u optimizadores de código fuente, generadores de conjuntos de datos de entrada, etc) como en el módulo **A.IR** (para los diccionarios, muestras de ficheros o conjuntos de datos utilizados).

**III-A2. Prefuzzing:** Todas aquellas tareas previas al *fuzzing* tales como preparativos, configuraciones, provisiones de muestras o conjuntos de datos forman parte de esta fase. Según el tipo de *fuzzing* a realizar, las tareas pueden comenzar en el módulo **1.SCI** o en el **2.BRF**.

Por un lado, el primer caso se da cuando el tipo de *fuzzing* seleccionado es *white-box* o *grey-box*; en función de la dependencia de las pruebas del código fuente. En este caso, el módulo es el encargado de realizar las diferentes actividades necesarias para preparar el código fuente de cara a su compilación y atendiendo a las necesidades de alcance, estrategia, requisitos y herramientas elegidas para ese flujo. Por otro lado, el segundo caso se da cuando el tipo de *fuzzing* seleccionado es *black-box* y, en cuyo caso, el módulo **1.SCI** no está implicado. Usándose directamente el módulo **2.BRF**.

Así mismo, el módulo **2.BRF** es común a cualquier flujo de *fuzzing* independientemente del tipo empleado. La finalidad de este módulo es realizar el almacenamiento del binario objetivo del *fuzzing* y el código utilizado para su generación si está disponible; llevando a cabo la compilación del código fuente previamente adecuado dentro del módulo **1.SCI**. Más aún, en el módulo **2.BRF** se realizan, si procede, las comprobaciones necesarias para verificar que los binarios son funcionales y también la alineación con la estrategia y requisitos planteados.

Cabe señalar que un punto común entre los módulos **1.SCI** y **2.BRF** es el código fuente utilizado en alguno de los tipos de *fuzzing*. No obstante, existe una diferencia fundamental entre ambos y es el contexto y la finalidad de este último. En el caso de **1.SCI**, el objetivo es trabajar con el código fuente original del software y realizar las transformaciones necesarias para el *fuzzing*. Por su parte, en el caso de **2.BRF** se utiliza un código modificado para un contexto específico de *fuzzing*. Al mismo tiempo, a nivel de almacenamiento también hay diferencias. Mientras que en **1.SCI** se asume que el código fuente es una única dependencia asociada a una versión de software, **2.BRF** mantiene un repositorio con el código modificado y los binarios asociados a este.

También se debe agregar que en esta fase se confecciona el paquete de trabajo para el *fuzzing* o para cualquier tarea de análisis o generación de dependencias para este, en el módulo **3.WLP**; en función del tipo de herramienta, técnica empleada y entorno de ejecución de las pruebas. Si es necesario, también

se alinean los conjuntos de datos del repositorio **A.IR**.

Otros posibles usos de los paquetes de trabajo con el fin de ejecutar las tareas de forma escalable o paralelizable son: Los de análisis de código estático para preparar la estrategia de *fuzzing*, el filtrado de resultados y el uso de generadores de entradas.

**III-A3. fuzzing:** Esta fase, común a todos los tipos de *fuzzing*, comprende únicamente el módulo **3.WLP**. En ella se realizan las tareas de provisión del paquete de trabajo, el seguimiento de su ejecución y el volcado de los resultados en el repositorio de datos de salida en crudo **B.OR**. Un ejemplo de tarea realizada en esta fase dentro del componente **3.WLP** sería la gestión de colas de trabajo asociadas a un paquete.

**III-A4. Postfuzzing:** Esta última fase se realiza en el módulo **3.WLP**. Se centra en el procesamiento y análisis de los resultados de los paquetes de trabajo previos de *fuzzing*; ultimando para ello el repositorio **B.OR**. Ejemplos de tareas pertenecientes a esta fase podrían ser el uso de herramientas de filtrado de falsos positivos, la generación de reportes, etc.

#### IV. PRUEBA DE CONCEPTO

El marco de trabajo HOUSE es *open source* y está disponible públicamente en GitHub [27]. De cara a validar su propuesta, se está llevando a cabo una prueba de concepto en el entorno de ejecución del HPC Caléndula [50]. Este entorno utiliza Slurm [51] como sistema de gestión de cargas y las arquitecturas Intel, Cascade Lake y Haswell, sobre el sistema operativo CentOS 7.7. Caléndula cumple con las políticas y requisitos del Esquema nacional de seguridad [52].

Cabe destacar que para la prueba de concepto no se ha utilizado ningún sistema de contenedores en ninguno de los módulos, ni se aplican excepción de políticas de seguridad o uso de permisos especiales.

La prueba de concepto consiste en desarrollar diversos scripts para Slurm e implementar casos de uso básicos con el *fuzzer* AFL++ [53], un proyecto *open source* que integra las últimas técnicas de *fuzzing* y continúa el proyecto discontinuado de AFL. El tipo de *fuzzing* elegido para esta prueba fue el *white-box*, utilizando la cobertura de código sobre los binarios *date* y *expr* de la colección de utilidades coreutils, perteneciente a los sistemas GNU [54]. También se incluyó un caso básico de Buffer Overflow. En relación a la entrada de datos, en ambos casos se utilizó la entrada estándar.

Desde el inicio de las actividades se han resuelto distintos problemas derivados de la naturaleza del HPC Caléndula, un entorno multipropósito y multiusuario con requisitos de seguridad y privacidad elevados que condicionan el desarrollo.

Los principales problemas identificados son el uso de nodos compartidos, lo que imposibilita relajar las protecciones y mecanismo de monitorización existentes. Destacan las limitaciones de uso de herramientas basadas en ASAN [55] o la monitorización de *crashes* usando ABRT [56], ya sea para evitar bloquear recursos comunes de los nodos o por su uso como parte de las protecciones de los sistemas. Esto se soluciona utilizando otras estrategias de detección basadas totalmente en instrumentación.



Inicialmente se trató de evitar complejidad usando la instrumentación mas básica del AFL++, afl-gcc similar a la del experimento, para reducir el número de dependencias y la complejidad para la prueba de concepto. Por los problemas anteriormente mencionados, se está trabajando en la adecuación de instrumentación apoyada en LLVM.

```
CALENDULA [root@cn3064 ~]# cd /frontend2/data/coreutils.scayle && ./status.sh
Individual fuzzers
=====
>>> name: MF987802 state: RUNNING (0 days, 0 hrs) node: cn3064 PID: 10441 JOBID: 987802 <<<
Fuzzer activity metrics:
last_path      : none seen yet
last_crash     : none seen yet
last_hang      : none seen yet
cycles_wo_finds : 225
cycle 227, lifetime speed 496 execs/sec, path 0/1 (0%)
pending 0/0, coverage 0.14%, no crashes yet

Performance metrics:
AveCPU AveDiskRead AveDiskWrite AveRSS AveVMSize MaxDiskRead MaxDiskWrite MaxRSS MaxVMSize
00:00.000 453399 71987 4554K 418232K 453399 71987 4554K 418232K

>>> name: SF987803 state: RUNNING (0 days, 0 hrs) node: cn3064 PID: 17885 JOBID: 987803 <<<
Fuzzer activity metrics:
last_path      : none seen yet
last_crash     : none seen yet
last_hang      : none seen yet
cycles_wo_finds : 220
cycle 221, lifetime speed 505 execs/sec, path 0/1 (0%)
pending 0/0, coverage 0.14%, no crashes yet

Performance metrics:
AveCPU AveDiskRead AveDiskWrite AveRSS AveVMSize MaxDiskRead MaxDiskWrite MaxRSS MaxVMSize
00:00.000 448135 67339 4543K 418260K 448135 67339 4543K 418260K

Summary stats
=====
Fuzzers alive : 2
Total run time : 3 minutes, 51 seconds
Total execs : 115 thousands
Cumulative speed : 1001 execs/sec
Average speed : 500 execs/sec
Pending paths : 0 faves, 0 total
Pending per fuzzer : 0 faves, 0 total (on average)
Crashes found : 0 locally unique
Cycles without finds : 225/220
Time without finds : 0
```

Figura 3. Ilustración status de los trabajos de fuzzing.

En la Fig. 3 se puede observar la ejecución de un paquete de trabajo de *fuzzing* utilizando dos trabajos y las métricas de rendimiento y actividad. Así como el sumatorio total de la actividad de todos los *fuzzers*.

## V. CONCLUSIONES

El *fuzzing* está cobrando cada vez más protagonismo como actividad para garantizar la calidad y la seguridad del software. Esta tendencia es utilizada por empresas y organizaciones para poner en valor sus desarrollos o servicios de forma comercial, a la vez que anima al desarrollo de mejoras y herramientas. Esta situación provoca la dependencia tecnológica y un riesgo a la confidencialidad para su adopción en un ciclo de desarrollo seguro y dificulta la puesta en marcha. Esta dependencia también frena la adopción de mejoras, el uso de alternativas totalmente libres y agnósticas de la tecnología.

HOUSE propone un modelado del flujo de trabajo del *fuzzing* caracterizado por tener un enfoque abierto y agnóstico a las herramientas. Con esta finalidad, el marco de trabajo propone una visión ordenada de todas las fases previas y posteriores a las pruebas. De esta manera, HOUSE permite resolver o simplificar las tareas más tediosas y complejas relacionadas con la aplicación de técnicas de *fuzzing*.

Este flujo se basa en los pasos y elementos establecidos en la Fig. 2.

Así pues, esta investigación sienta las bases teóricas comunes y las prácticas mínimas de un marco de trabajo que permite reducir el esfuerzo inicial a la hora de poner en marcha un entorno de *fuzzing* escalable que garantice la confidencialidad de la información. De igual modo, HOUSE facilita, tanto a investigadores como a desarrolladores, el hecho de poder trabajar en un componente o funcionalidad específica o el hecho de incluir aquellas características de seguridad proactiva en el software dentro de su ciclo de desarrollo. Estos aspectos se traducen en una reducción de la carga de trabajo en tareas auxiliares o distintas a las de los objetivos planificados.

Esta primera versión del marco de trabajo brinda la estructura de directorios, repositorios de datos y recursos para utilizar el AFL++ [53] con instrumentación básica, incluyendo las automatizaciones para la creación de cargas de trabajo, usando un sistema de colas Slurm [51].

## AGRADECIMIENTOS

Este trabajo ha sido parcialmente financiado por INCIBE mediante la Adenda 4 al convenio marco con la Universidad de León. Los autores agradecen al centro de supercomputación SCAYLE la posibilidad de utilizar sus infraestructuras para validar la propuesta presentada en este trabajo de investigación.

## REFERENCIAS

- [1] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Upper Saddle River, NJ: Addison-Wesley, 2007.
- [2] IEEE Communications Society, "IEEE Standard Glossary of Software Engineering Terminology," *Office*, vol. 121990, no. 1, p. 1, 1990. [Online]. Available: [http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=159342](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=159342)
- [3] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990. [Online]. Available: <https://dl.acm.org/doi/10.1145/96267.96279>
- [4] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services," p. 23, 1995.
- [5] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proceedings of the 16th ACM Conference on Computer and Communications Security - CCS '09*. Chicago, Illinois, USA: ACM Press, 2009, p. 566. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1653662.1653730>
- [6] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. Lisbon, Portugal: IEEE, Jun. 2009, pp. 359–368. [Online]. Available: <http://ieeexplore.ieee.org/document/5270315/>
- [7] J. Wei, J. Chen, Y. Feng, K. Ferles, and I. Dillig, "Singularity: Pattern fuzzing for worst case complexity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Lake Buena Vista FL USA: ACM, Oct. 2018, pp. 213–223. [Online]. Available: <https://dl.acm.org/doi/10.1145/3236024.3236039>
- [8] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming," in *Computer Security – ESORICS 2016*, I. Askoxylakis, S. Ioannidis, S. Katsikas, and C. Meadows, Eds. Cham: Springer International Publishing, 2016, vol. 9878, pp. 581–601. [Online]. Available: [http://link.springer.com/10.1007/978-3-319-45744-4\\_29](http://link.springer.com/10.1007/978-3-319-45744-4_29)
- [9] M. Y. Wong and D. Lie, "IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware," in *Proceedings 2016 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2016. [Online]. Available: <https://www.ndss-symposium.org/wp-content/uploads/2017/09/intelldroid-targeted-input-generator-dynamic-analysis-android-malware.pdf>

- [10] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "LEOPARD: Identifying Vulnerable Code for Vulnerability Assessment through Program Metrics," 2019. [Online]. Available: <http://arxiv.org/abs/1901.11479>
- [11] M. Z. Nasrabadi, S. Parsa, and A. Kalaei, "Neural Fuzzing: A Neural Approach to Generate Test Data for File Format Fuzzing," 2018. [Online]. Available: <http://arxiv.org/abs/1812.09961>
- [12] ENISA, "Risk Management: Implementation principles and Inventories for Risk Management/Risk Assessment methods and tools "Survey of existing Risk Management and Risk Assessment Methods") Conducted by the Technical Department of ENISA Section Risk Management," 2006.
- [13] Y. Li, S. Ji, C. Lv, Y. Chen, J. Chen, Q. Gu, and C. Wu, "V-Fuzz: Vulnerability-Oriented Evolutionary Fuzzing," pp. 1–16, 2019. [Online]. Available: <http://arxiv.org/abs/1901.01142>
- [14] E. Güler, C. Aschermann, A. Abbasi, and T. Holz, "ANTIFUZZ: Impeding Fuzzing Audits of Binary Executables," in *28th USENIX Security Symposium*. Usenix Association, Aug. 2019, p. 18.
- [15] C. R. Cioce, D. G. Loffredo, and N. J. Salim, "Program Fuzzing on High Performance Computing Resources." Tech. Rep. SAND2019-0674, 1492735, Jan. 2019. [Online]. Available: <http://www.osti.gov/servlets/purl/1492735/>
- [16] Amazon, "AWS Plugin for Slurm - repository." [Online]. Available: <https://github.com/aws-samples/aws-plugin-for-slurm>
- [17] Microsoft, "Azure CycleCloud," Feb. 2021. [Online]. Available: <https://docs.microsoft.com/en-us/azure/cyclecloud/slurm?view=cyclecloud-8>
- [18] SchedMD, "Cloud Scheduling Guide." [Online]. Available: [https://slurm.schedmd.com/elastic\\_computing.html](https://slurm.schedmd.com/elastic_computing.html)
- [19] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research," *USENIX*, p. 12, 2020.
- [20] C. Cioce, N. Salim, J. Rigdon, and D. Loffredo, "Multi-Node Program Fuzzing on High Performance Computing Resources." Tech. Rep. SAND2020-8215, 1650237, 690013, Aug. 2020. [Online]. Available: <https://www.osti.gov/servlets/purl/1650237/>
- [21] Abhishek Arya, Oliver Chang, Max Moroz, Martin Barbella, and Jonathan Metzman, "Google Online Security Blog: Open sourcing ClusterFuzz," Feb. 2019. [Online]. Available: <https://security.googleblog.com/2019/02/open-sourcing-clusterfuzz.html>
- [22] Microsoft Research, "Microsoft Security Risk Detection ("Project Springfield") - Microsoft Research," Jan. 2015. [Online]. Available: <https://www.microsoft.com/en-us/research/project/project-springfield/>
- [23] Microsoft, "GitHub - microsoft/onefuzz: A self-hosted Fuzzing-As-A-Service platform." [Online]. Available: <https://github.com/microsoft/onefuzz>
- [24] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker, "Google Testing Blog: Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software," Dec. 2016. [Online]. Available: <https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html>
- [25] Microsoft, "Microsoft Security Risk Detection." [Online]. Available: <https://www.microsoft.com/en-us/security-risk-detection/>
- [26] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," *arXiv:1812.00140 [cs]*, Apr. 2019. [Online]. Available: <http://arxiv.org/abs/1812.00140>
- [27] Francisco Borja Gamelo Del Rio, "HOUSE framework repository," 2021. [Online]. Available: <https://github.com/b0rh/HOUSE>
- [28] A. Takanen, J. DeMott, C. Miller, and A. Kettunen, Eds., *Fuzzing for Software Security Testing and Quality Assurance*, 2nd ed., ser. Artech House Information Security and Privacy Series. Norwood, MA: Artech House, 2018.
- [29] "CERT BFF," Oct. 2016. [Online]. Available: <https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=507974>
- [30] Aki Helin, "Radamsa source code repository." [Online]. Available: <https://gitlab.com/akihe/radamsa>
- [31] Michał Zalewski, "AFL - american fuzzy lop." [Online]. Available: <https://lcamtuf.coredump.cx/afl/>
- [32] NSA, "Security-Enhanced Linux," 2008. [Online]. Available: <https://web.archive.org/web/20200915000700/https://www.nsa.gov/What-We-Do/Research/SELinux/>
- [33] Chris Brown, "Protect your applications with AppArmor - Linux.com," Aug. 2006. [Online]. Available: <https://www.linux.com/news/protect-your-applications-apparmor/>
- [34] N. N. A. Sjarif, S. Chuprat, M. N. Mahrin, N. A. Ahmad, A. Ariffin, F. M. Senan, N. A. Zamani, and A. Saupi, "Endpoint Detection and Response: Why Use Machine Learning?" in *2019 International Conference on Information and Communication Technology Convergence (ICTC)*. Jeju Island, Korea (South): IEEE, Oct. 2019, pp. 283–288. [Online]. Available: <https://ieeexplore.ieee.org/document/8939836/>
- [35] Gilad Maayan, "Comparing endpoint security: EPP vs. EDR vs. XDR - Infosec Resources," Dec. 2020. [Online]. Available: <https://resources.infosecinstitute.com/topic/comparing-endpoint-security-epp-vs-edr-vs-xdr/>
- [36] GitHub Security Lab, "CodeQL." [Online]. Available: <https://securitylab.github.com/tools/codeql/>
- [37] NIST, "Source Code Security Analyzers." [Online]. Available: [https://samate.nist.gov/index.php/Source\\_Code\\_Security\\_Analyzers.html](https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html)
- [38] "ISO/IEC 2382:2015(en) Information technology — Vocabulary," 2015. [Online]. Available: <https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:ed-1:v1:en>
- [39] P. Godfrey, M. Y. Levin, and D. Molnar, "Automated Whitebox Fuzz Testing," *ResearchGate*, p. 17, Jan. 2008.
- [40] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Hoboken, NJ: John Wiley & Sons, 2012.
- [41] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *2009 IEEE 31st International Conference on Software Engineering*. Vancouver, BC, Canada: IEEE, 2009, pp. 474–484. [Online]. Available: <http://ieeexplore.ieee.org/document/5070546/>
- [42] Muthu Ramachandran, "Testing software components using boundary value analysis," in *Proceedings of the 20th IEEE Instrumentation Technology Conference (Cat No 03CH37412) EURMIC-03*. Belek-Antalya, Turkey: IEEE, 2003, pp. 94–98. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/1231572>
- [43] M. Roper, "Using Machine Learning to Classify Test Outcomes," in *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*. Newark, CA, USA: IEEE, Apr. 2019, pp. 99–100. [Online]. Available: <https://ieeexplore.ieee.org/document/8718223/>
- [44] M. Z. Nasrabadi, S. Parsa, and A. Kalaei, "Format-aware Learn&Fuzz: Deep Test Data Generation for Efficient Fuzzing," *Neural Computing and Applications*, vol. 33, no. 5, pp. 1497–1513, Mar. 2021. [Online]. Available: <http://arxiv.org/abs/1812.09961>
- [45] L. Cheng, Y. Zhang, Y. Zhang, C. Wu, Z. Li, Y. Fu, and H. Li, "Optimizing seed inputs in fuzzing with machine learning," pp. 1–2, 2019. [Online]. Available: <http://arxiv.org/abs/1902.02538>
- [46] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-Aware Greybox Fuzzing," 2018. [Online]. Available: <http://arxiv.org/abs/1812.01197>
- [47] Nooper Davis, "Secure Software Development Life Cycle Processes," Jul. 2013.
- [48] Gerbrand van Dieijen, "Metaphors in software development," Jul. 2010. [Online]. Available: <https://xebia.com/blog/metaphors-in-software-development/>
- [49] OMG, "Business Process Model and Notation (BPMN) Version 2.0.2," Dec. 2013. [Online]. Available: <https://www.omg.org/spec/BPMN/2.0.2/>
- [50] "SCAYLE, Supercomputación Castilla y León." [Online]. Available: <https://www.scayle.es/>
- [51] "Slurm Workload Manager." [Online]. Available: <https://slurm.schedmd.com/>
- [52] "BOE-A-2010-1330 Real Decreto 3/2010, de 8 de enero, por el que se regula el Esquema Nacional de Seguridad en el ámbito de la Administración Electrónica." [Online]. Available: <https://www.boe.es/buscar/act.php?id=BOE-A-2010-1330&b=23&m=1&p=20151104>
- [53] "AFLplusplus is the daughter of the American Fuzzy Lop fuzzer by Michal "lcamtuf" Zalewski." [Online]. Available: <https://aflplusplus.com/>
- [54] GNU, "Coreutils." [Online]. Available: <https://www.gnu.org/software/coreutils/coreutils.html>
- [55] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," *USENIX*, p. 10, 2012.
- [56] "Chapter 25. Automatic Bug Reporting Tool (ABRT) Red Hat Enterprise Linux 7 — Red Hat Customer Portal." [Online]. Available: [https://access.redhat.com/documentation/en-us/red\\_hat\\_enterprise\\_linux/7/html/system\\_administrators\\_guide/ch-abrt](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/system_administrators_guide/ch-abrt)