Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Fuzzing In-Vivo Mediante Amplificación de Ejecuciones

Tesis de Licenciatura en Ciencias de la Computación

Octavio Adolfo Galland

Director: Juan Pablo Galeotti
Codirector: Marcel Böhme
Buenos Aires, 2023

FUZZING IN-VIVO MEDIANTE AMPLIFICACIÓN DE EJECUCIONES

Uno de los mayores cuellos de botella al aplicar fuzz testing sobre librerías es la necesidad de contar con *fuzz drivers*. Estos son programas que hacen las veces de intermediarios entre el fuzzer y la librería siendo testeada. El hecho de que vulnerabilidades críticas sigan siendo encontradas en librerías que son continuamente sometidas a fuzzing delata la insuficiencia de estos drivers en la práctica.

En este trabajo, proponemos un enfoque alternativo al fuzzing de librerías, que haga uso de una ejecución válida de un programa que utilice la librería (el *host*), y *amplifique* su ejecución. Más concretamente, ejecutamos el host hasta que una determinada función de una lista de funciones *objetivo* sea ejecutada, y luego procedemos a aplicar fuzzing guiado por cobertura sobre la misma. Una vez agotado el presupuesto de tiempo asignado a este objetivo, avanzamos al siguiente objetivo de la lista. De esta manera no solo reducimos la cantidad de esfuerzo manual requerido para incorporar fuzzing al ciclo de desarrollo de una librería, sino que también permitimos llevar a cabo el testing de la misma en un contexto que refleje el uso que se le daría en un contexto productivo.

Palabras claves: fuzzing, testing automatizado, seguridad de software, detección de vulnerabilidades, exploración de caminos.

IN-VIVO FUZZING BY AMPLIFYING ACTUAL EXECUTIONS

A major bottleneck that remains when fuzzing software libraries is the need for *fuzz drivers*, i.e., the glue code between the fuzzer and the library. Despite years of fuzzing, critical security flaws are still found, for instance by manual auditing, because the fuzz drivers do not cover the complex interactions between the library and the real-world programs using it.

In this work we propose an alternative approach to library fuzzing, which leverages a valid execution context set up by a given program using the library (the *host*), and *amplify* its execution. More specifically, we execute the host until a designated function from a list of *target* functions has been reached, and then perform coverage-guided function-level fuzzing on it. Once the fuzzing quota is exhausted, we move on to fuzzing the next target from the list. In this way we not only reduce the amount of manual work needed by a developer to incorporate fuzz testing into their workflow, but we also allow the fuzzer to explore parts of the library as they are used in real-world programs that may otherwise not have been tested due to the simplicity of most fuzz drivers.

Keywords: fuzzing, automated testing, software security, vulnerability detection, path exploration.

Índice general

1. INTRODUCCIÓN

1.1. Software testing

Denominamos *software testing* (o testing de software) a la práctica de ejecutar un *sistema bajo test* (SUT, por sus siglas en inglés) y contrastar el comportamiento observado contra el esperado. La existencia de discrepancias entre el comportamiento observado y el esperado (*fallas*) es síntoma de *defectos* en el código, que deberán ser arreglados por un desarrollador.

A la combinación de un dato de entrada para el SUT y un mecanismo para comprobar si el comportamiento observado se corresponde con el esperado, la denominaremos *caso de test*. Cuando el caso de test se ejecuta manualmente, se debe determinar de antemano y de manera analítica cuál es el comportamiento deseado por parte del SUT en respuesta a un estímulo, y luego comprobar manualmente que el SUT exhiba dicho comportamiento.

1.2. Testing automatizado

Testear manualmente un SUT es costoso y propenso a errores, por lo que resulta deseable automatizar este proceso tanto como sea posible. A modo de ejemplo de una técnica de testing automatizado que goza de una gran adopción en la práctica podemos mencionar al testing unitario. El mismo consiste en escribir fragmentos de código que lleven a cabo los casos de test de manera automática. Es decir, que ejerciten el SUT utilizando como entrada datos de prueba determinados al momento de escribir el test, y luego comparen la respuesta provista por el SUT con el resultado esperado.

Si bien automatizar la ejecución de casos de test reduce el trabajo manual requerido durante el testeo, no elimina la necesidad de desarrollar dichos casos en primera instancia. Reducir la cantidad de esfuerzo manual necesario para crear estos casos es el problema que busca abordar la generación automática de casos de test.

1.3. Oráculos

Una dificultad que se presenta al intentar automatizar la generación de casos de test es la necesidad de contar con un *oráculo*. Esto es, un componente que permita determinar, a partir de un estímulo provisto al SUT y la respectiva respuesta, si el comportamiento observado se corresponde con el esperado. La necesidad de contar con dichos oráculos suele ser un problema, dado que en muchos casos solo se cuenta con una descripción informal del comportamiento esperado del SUT. En estos casos resulta difícil determinar automáticamente qué comportamientos son correctos para un estímulo dado. Este problema puede mitigarse si se cuenta con una especificación del SUT, o si la misma puede derivarse de documentación o de trazas de ejecución [4].

1.3.1. Oráculos implícitos

En los casos en los que no se cuente con una especificación formal del SUT, se puede hacer uso de lo que se denominan *oráculos implícitos*. Estos son oráculos que permiten

discernir comportamientos que en casi cualquier circunstancia pueden ser considerados erróneos, independientemente del dominio del problema. Por ejemplo, si un programa intenta dereferenciar una dirección de memoria inválida, es seguro asumir que se trata de una falla, producto de un defecto en el código que debe ser remediado.

Los *crashes* que ocurran durante la ejecución de un programa podrían utilizarse a modo de oráculo implícito, ya que ningún programa debería terminar de manera anormal. Inclusive en los casos en los que un programa detecte una condición de error, resulta esperable que el programa reporte el error y finalice su ejecución de manera normal, sin que el sistema operativo tenga que abortarlo forzosamente.

En la práctica existen situaciones en las que no se ocasiona un crash, pero que igualmente se exhibe un comportamiento que podría considerarse incorrecto en cualquier circunstancia. Tal es el caso, por ejemplo, de un *use-after-free*, situación en la cual un proceso libera memoria a la que posteriormente vuelve a acceder. Otro ejemplo que podría considerarse son las condiciones de carrera, en las cuales dos hilos de ejecución intentan acceder al mismo recurso compartido, sin utilizar mecanismos de control de concurrencia para arbitrar el acceso a dicho recurso. Existen numerosas herramientas que buscan detectar este tipo de fallas de manera automática en tiempo de ejecución, las cuales también pueden ser utilizadas a modo de oráculos implícitos [2, 8, 15, 25, 30]

1.4. Fuzzing

Fuzzing (o fuzz testing) se presenta como una técnica de testing automatizado que nos permite valernos de oráculos implícitos para buscar errores en el SUT, ejecutándolo con distintas entradas generadas automáticamente. Esta técnica se concibió para testear utilidades de `UNIX`, y originalmente consistía en generar cadenas de caracteres alteatorias, y proveerselas al SUT, monitoreando el proceso para identificar posibles crashes [20]. De manera más general, podemos considerar a la entrada de un SUT como una secuencia de bytes, y así testear programas que no necesariamente se limiten al procesamiento de texto. Mediante este proceso se pueden exponer fallas simples en programas poco robustos, pero la técnica rápidamente se torna insuficiente al intentar testear programas más complejos. Esto ocurre dado que, al ejecutar el SUT moderadamente complejo con una entrada generada de manera completamente aleatoria, es altamente probable que la ejecución solo ejercite regiones del código relacionadas con la validación de la entrada, impidiéndonos testear regiones de código más "profundas" (y, por lo tanto, mas interesantes).

Para subsanar este problema, se puede intentar generar entradas de manera más eficiente, tomando en cuenta información del dominio del programa. A modo de ejemplo, consideremos el caso de un intérprete de un lenguaje de programación. Está claro que si generamos cadenas de texto de manera uniformemente aleatoria la mayoría de las entradas generadas van a ser rechazadas por el parser, sin lograr ejecutar regiones de código relacionadas al runtime del lenguaje. En este caso podríamos utilizar la gramática del lenguaje para generar programas sintácticamente válidos, y así ejercitar regiones de código que no se limiten necesariamente al parser [1, 16, 27, 28].

Otro enfoque es el denominado *mutation-based fuzzing* (o fuzzing basado en mutaciones), el cual consiste en, a partir de un conjunto de entradas dado, aplicar mutaciones sobre algunas de ellas para generar nuevas entradas con las cuales estimular al SUT. Esta técnica nos permite explorar la "vecindad" de un subconjunto del espacio de entradas del programa. Si el conjunto de entradas con el que se cuenta al iniciar el proceso incluye en-

tradas que ejerciten regiones de código "profundas" dentro del SUT, resulta esperable que las mutaciones generadas también lo hagan. En particular, si se parte de entradas patológicas, o que evidenciaron defectos en versiones anteriores del SUT (es decir, *regresiones*), podremos arribar con mayor rapidez a entradas que expongan defectos nuevos.

Este enfoque mutacional no requiere de información de dominio provista explícitamente por el usuario. Al no contar con dicha información, resulta esperable también que se generen entradas inválidas, es decir, que no cumplan con el formato requerido por el SUT. Pero, al tratarse de variaciones de entradas válidas, existe la posibilidad de que estas entradas sean lo suficientemente similares a una entrada válida como ser consideradas erróneamente como válidas por el SUT y así exponer fallas.

La técnica a la que nos vamos a referir de aquí en adelante es la denominada *coverage-guided greybox fuzzing* (CGF) [6, 23]. La misma consiste en aplicar fuzzing basado en mutaciones, pero guiando el proceso utilizando información recabada durante las ejecuciones del SUT. Para guiar el proceso se utiliza, típicamente, información de cobertura de aristas en el grafo de control de flujo (CFG, por sus siglas en inglés) del programa siendo testeado. En particular, se mantiene un *corpus* de entradas (inicialmente conformado por un conjunto de entradas dadas por el usuario), en el cual cada elemento ejercita un *camino* distinto en el CFG del SUT, y conforme se obtengan entradas que ejercitan caminos previamente desconocidos, las mismas se agregan a dicho corpus.

En Algorithm 1 se presenta el pseudo-código correspondiente a una campaña de CGF. Dado el SUT, un conjunto entradas y un presupuesto de tiempo, iteramos primero sobre el conjunto entradas dado, quedándonos con un subconjunto minimal del mismo que cubra los mismos caminos (ciclo que abarca las líneas 4-10). Posteriormente, mientras el presupuesto de tiempo no haya expirado, seleccionamos una entrada i del corpus y le asignamos una *energía* e (líneas 11-13).

El valor de la energía indica la cantidad de mutantes que vamos a generar a partir de la entrada seleccionada. Cada CGF provee su propio mecanismo para determinar este valor. A modo de ejemplo podemos mencionar que `AFL++` [10], si bien provee varias políticas para asignar energía, por defecto le asigna a cada entrada una energía [6]:

$$e(i) = \text{mín}\{C\frac{2^{s(i)}}{f(i)}, M\}$$

Con M y C constantes, $f(i)$ la frecuencia con la que se haya ejercido el camino que ejerce la entrada i en el CFG del SUT, y $s(i)$ la cantidad de veces que se haya seleccionado i del corpus. La intuición detrás de esta fórmula es priorizar entradas que, a pesar de ser seleccionadas repetidas veces, hayan ejercido caminos poco frecuentes (asegurando que la energía sea mayor a una cota inferior M). En el caso de `LibFuzzer` [19], se utiliza una política de asignación de energía más sofisticada basada en teoría de la información [5]. En todos los casos, el fuzzer debe mantener actualizadas las estadísticas de las frecuencias con la que se observó el camino ejercido por cada input del corpus a lo largo de toda la campaña. De esta manera la decisión de cuánto tiempo invertir en mutar y ejecutar cada entrada está informada por información de cobertura histórica sobre el SUT.

Una vez determinado un valor para e, procedemos a generar e mutantes a partir de i. Los mutantes se obtienen como resultado de aplicar operadores de mutación sobre la entrada seleccionada. Normalmente estos operadores trabajan a nivel de bits/bytes, aunque es posible hacer uso de operadores de mutación que tengan en cuenta información sobre la estructura esperada de los inputs. Ejecutamos el SUT con cada entrada generada

Algorithm 1 *Coverage-Guided Greybox Fuzzing*

Input: s SUT, Q_0 initial corpus set, B budget
Output: $Q_{✗}$ crashing inputs

```
1:  Q✗ ← ∅
2:  Q ← Q0
3:  P ← ∅
4:  for i ∈ Q0 do
5:      p ← execute_and_trace(s, i)
6:      if p ∉ P then
7:          Q ← Q ∪ {i}
8:          P ← P ∪ {p}
9:      end if
10: end for
11: while B not expired do
12:     i ← chooseNext(Q)
13:     e ← assignEnergy(i)
14:     for n : 1 ≤ n ≤ e do
15:         in ← mutate(i)
16:         p ← execute_and_trace(s, in)
17:         if s crashed then
18:             Q✗ ← Q✗ ∪ {in}
19:         end if
20:         if p ∉ P then
21:             P ← P ∪ {p}
22:             Q ← Q ∪ {in}
23:         end if
24:     end for
25: end while
26: return Q✗
```

y, por cada una, guardamos un registro de si se generó una falla (recordar que solo nos valemos de oráculos implícitos para detectar fallas), y si se ejercitó un nuevo camino en el CFG. Si se detectó una falla agregamos la entrada al conjunto resultado (línea 17-19). Si la entrada ejercita un camino previamente desconocido, la agregamos al corpus y agregamos el camino a la lista de caminos explorados (líneas 20-23).

Esta técnica posibilita descubrir, de manera incremental, entradas que ejerciten caminos cada vez más complejos en el SUT. Esto se puede aplicar de manera genérica sobre cualquier SUT, requiriendo únicamente un proceso de instrumentación que posibilite recolectar información concerniente al control de flujo en tiempo de ejecución.

1.5. Fuzzing de librerías

Para poder aplicar fuzz testing sobre librerías de código, resulta necesario sortear algunos problemas. En lo anterior, asumíamos que era posible ejecutar el SUT, y que el mismo recibía secuencias de bytes a modo de entrada *a nivel de sistema*. Por un lado, las librerías no tienen un único punto de entrada, sino que tienen un conjunto de puntos

desde los cuales los usuarios pueden invocar las diferentes funcionalidades (conocido como *application programming interface*, o API). Sumado a la dificultad de identificar dichos puntos de entrada, se presenta la dificultad de que en muchos casos existen dependencias entre llamadas a la API (por ejemplo, en muchos casos es necesario invocar una rutina de inicialización antes de invocar a cualquier otra rutina). Otra dificultad que se presenta es que ya no contamos con un SUT que tome una única entrada a nivel de sistema, sino que tenemos que generar distintas entradas *a nivel de función* por cada porción de la librería que deseemos testear.

Una forma de sortear estos problemas adoptada en la industria consiste en utilizar *fuzz drivers* (también denominados *harnesses*). Un harness es un fragmento de código que actúa como punto de entrada durante la campaña de fuzzing. El mismo recibe la entrada (en forma de secuencia de bytes) provista por el fuzzer y la convierte al formato correspondiente para brindarlo como input a un subconjunto de la API de la librería a testear. De esto se sigue que el harness determina cuáles funcionalidades de la librería se van a testear. En la práctica suelen implementarse varios harnesses para una misma librería, cada uno atacando un subconjunto diferente de la misma.

Existen varias desventajas asociadas a la utilización de fuzz drivers. Por un lado, el desarrollo de fuzz drivers aún se realiza mayormente de manera manual, lo cual representa un gran cuello de botella en la práctica. Por otra parte, los harnesses construyen un contexto sintético dentro del cual invocan a las funcionalidades de la librería que se desea testear. Este contexto suele ser demasiado simplista y, por lo tanto, difícilmente refleje las condiciones en las que la librería sería utilizada en una aplicación real. A modo de ejemplo, podemos mencionar que en la documentación de `LibFuzzer` [19], se recomienda desarrollar harnesses rápidos y que testeen porciones lo más acotadas posible de la librería. También se recomienda, dado que el harness es invocado múltiples veces dentro del mismo proceso, que no se modifique el estado global del proceso. Seguir indicaciones como estas permiten al fuzzer maximizar la cantidad de ejecuciones por segundo, pero también fomentan un testeo superficial de la librería. Cabe destacar que `LibFuzzer` es uno de los fuzzers que goza de mayor adopción en la práctica, siendo utilizado por Google en el proyecto `OSS-Fuzz` [26], en el cual se someten a fuzz testing de manera continua a muchos proyectos open-source relevantes y críticos en términos de seguridad.

Como caso paradigmático de la insuficiencia de este enfoque se puede mencionar a la vulnerabilidad identificada como `CVE-2022-3602`, una vulnerabilidad de severidad alta en el proyecto `OpenSSL`. La misma se trata de un simple *buffer overflow* al decodificar `Punycode` durante la verificación de certificados `X.509`. La librería `OpenSSL` estaba integrada al proyecto `OSS-Fuzz` y había sido sometida a testing de manera continua durante 6 (seis) años al momento de reportarse la vulnerabilidad (inclusive contaba con un fuzz driver que atacaba específicamente la verificación de certificados `X.509`). Poco tiempo después de la publicación de esta vulnerabilidad, se comprobó que la misma se podía encontrar en cuestión de minutos mediante fuzzing, si tan solo se hubiera testeado la funcionalidad correspondiente.

1.6. Fuzzing *in-vivo*

Para superar las limitaciones planteadas por la utilización de harnesses para aplicar fuzzing sobre librerías proponemos una técnica denominada *in-vivo fuzzing*. La misma consiste en testear la librería (a cual denominaremos *target*) haciendo uso de un programa

Signature	Constraints
ASN1_parse(BIO* _, const char *pp, long len, int _)	sizeof(pp) = len ∧ len < C
OPENSSL_hexstr2buf(const char *str, long *_)	sizeof(str) < C
ossl_punycode_decode(char *pEnc, size_t encLen, int *pDec, int *pOutLen)	sizeof(pEnc) = encLen ∧ sizeof(pDec) = encLen ∧ encLen < C ∧ sizeof(pOutLen) = 1 ∧ *pOutLen = encLen
cms_kek_cipher(char **_, size_t *_, char *in, size_t inlen, CMS_KeyAgreeRecipientInfo *_, int _)	sizeof(in) = inlen ∧ inlen < C

Tab. 1.1: Extracto de las más de 100 precondiciones generadas semi-automáticamente para la librería OpenSSL. La constante C es arbitraria y fija para prevenir errores de memoria espurios. Por brevedad, se omiten los nombres de los parámetros que debe ser ignorados por el fuzzer.

pre-existente que la utilice (al cual denominaremos *host*). Ejecutamos el host, y permitimos que la ejecución continue normalmente hasta llegar a un *punto de amplificación*, el cual no es otra cosa que una función en el target especificada por el usuario. Una vez alcanzado dicho punto *amplificamos la ejecución*. Para esto bifurcamos la ejecución, y en un *fork* de la ejecución original procedemos a remplazar los argumentos provistos al punto de amplificación con datos aleatorios provistos por un fuzzer. Finalmente, permitimos que la ejecución del fork continue normalmente hasta su terminación, monitoreando al proceso para detectar eventuales crashes.

Esto nos permite llevar a cabo fuzzing sobre funciones del target en un contexto realista, ya que el mismo es erigido por un host productivo. Al modificar los argumentos provistos a la función no solo se está testeando la función en sí, sino también el efecto que esta puede tener sobre llamadas subsecuentes al target por parte del host (por ejemplo, modificando el estado global de la librería). Nos permite, también, prescindir de los fuzz drivers, aminorando la cantidad de esfuerzo manual necesaria para llevar a cabo el proceso de testing. Adicionalmente, nos permite testear *cualquier* función dentro de la librería, sin necesidad de limitarnos a funciones que formen parte de la API.

Este enfoque requiere de una especificación provista por el usuario en la cual se incluya una lista de funciones dentro del target a tratar como puntos de amplificación. También se debe detallar en esta especificación el conjunto de condiciones que los argumentos provistos a cada punto de amplificación deben satisfacer (ya que resulta inmediato ver que asignar valores totalmente aleatorios a los argumentos de funciones dentro del target puede resultar en una alta tasa de falsos positivos). En esta especificación se detallan también cuáles de los parámetros no deben ser modificados durante la campaña de fuzzing. Una muestra de esta especificación para una librería particular puede apreciarse en Tabla 1.1

Con el fin de contrastar la efectividad de esta técnica implementamos una prueba de concepto de la misma, basándonos en el fuzzer `AFL++ 4.02c` [10]. Dicha implementación consta de una versión modificada del ya mencionado fuzzer, de un pase de compilador implementado sobre `LLVM 14` [18] para llevar a cabo la instrumentación necesaria, y de una librería que provee utilidades auxiliares en tiempo de ejecución para permitir la am-

plificación de ejecuciones.

2. IMPLEMENTACIÓN

Dada una librería que se desea testear (a la cual denominaremos *target*) y un conjunto de funciones consideradas interesantes F (a las cuales denominaremos *puntos de amplificación*), buscamos aprovecharnos de un programa *host* que utilice al target para crear un contexto de ejecución realista dentro del cual testear a las funciones en F. Al ejecutarse un punto de amplificación dentro del proceso host, bifurcamos la ejecución del proceso repetidamente (a fines de evitar interferencia con el proceso original) y procedemos a mutar los argumentos provistos a la función objetivo dentro de estos forks, con el fin de encontrar un conjunto de parámetros que ocasionen un crash. Esta mutación se realiza respetando un conjunto de restricciones C especificado por el usuario sobre los parámetros de cada punto de amplificación.

In-vivo fuzzing está basado en la misma idea que el fuzzing mutacional para programas de procesamiento de archivos [1, 6, 23, 27, 28], o implementaciones de protocolos [9, 12, 22, 24]. Dada una entrada válida para un programa, por ejemplo un archivo `PDF` para un lector de este tipo de archivos, un fuzzer mutacional aplica pequeñas modificaciones sobre la misma para generar entradas "casi válidas" que puedan inducir crashes en el programa. A la entrada que utilizamos para iniciar el proceso la denominamos *semilla*, ya que es utilizada para iniciar la campaña y es a partir de ella que se obtiene el resto de las entradas generadas. Esta semilla existe en el espacio de entradas del programa, y a lo largo de la campaña la "vecindad" de esta semilla en dicho espacio es explorada por el fuzzer. Si el corpus inicial de semillas es lo suficientemente diverso, el fuzzer puede cubrir una gran variedad de comportamientos dentro del programa.

Análogamente, proponemos tomar al contexto de ejecución y los argumentos provistos por el host al invocar al target como semillas sobre las cuales aplicar fuzzing. De esta manera, nos circunscribimos a una vecindad de un estado válido del programa al generar nuevas entradas para cada función objetivo. Si el host interactúa con varios de los puntos de amplificación, el fuzzer in-vivo puede penetrar en regiones profundas del programa y cubrir y amplificar un conjunto diverso de estados del programa.

2.1. Procedimiento general

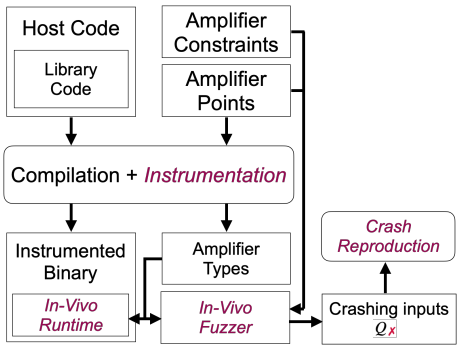


Fig. 2.1: Arquitectura de la implementación.

En Figura 2.1 se bosqueja la estructura general de nuestra implementación. Dados

el código del host y del target, sometemos a ambos a un proceso de instrumentación. En este paso insertamos llamadas a nuestra librería de runtime en el preámbulo de cada punto de amplificación de la librería y en el punto de entrada del programa. Cuando el control de flujo sea transferido a esta librería, podremos crear forks de la ejecución original para luego mutar los argumentos provistos a los puntos de amplificación utilizando las entradas provistas por el fuzzer. Este paso de instrumentación también recopila y exporta la información correspondiente a los tipos de los parámetros de cada punto de amplificación en formato `JSON`. Dicha información será utilizada en tiempo de ejecución para convertir las secuencias de bytes provistas por el fuzzer a valores cuyos tipos de datos se correspondan con los parámetros de los puntos de amplificación. También nos permitirá realizar el proceso inverso, para utilizar los argumentos provistos al target durante la primera ejecución como semillas iniciales.

Para determinar cuáles van a ser los puntos de amplificación, nos valemos de una lista de funciones provista por el usuario. Estas funciones no están necesariamente limitadas a la API de la librería. Si bien la identificación de tales funciones depende de conocimiento experto por parte del usuario, se pueden emplear heurísticas para facilitar este proceso en bases de código con las cuales el usuario no tenga suficiente familiaridad. Se pueden emplear una variedad de análisis estáticos o dinámicos para identificar funciones cuya firma o comportamiento sugiera la existencia de vulnerabilidades o procesamiento de datos potencialmente controlados por un adversario. A modo de ejemplo, nuestro prototipo de implementación busca identificar funciones relacionadas con *parsing*, ya que las mismas suelen procesar datos no confiables y ser propensas a errores. La identificación de las mismas se logra mediante la búsqueda de funciones con palabras clave en el nombre (por ejemplo, *parse*, *decode*), y cuyos parámetros incluyan un arreglo de bytes y un entero (con la expectativa de que se trate de un buffer de datos y su longitud, respectivamente).

Para minimizar la tasa de falsos positivos, el fuzzer recibe junto al conjunto de puntos de amplificación un conjunto de restricciones que los argumentos provistos a cada función deben cumplir. Estas restricciones son análogas a las precondiciones utilizadas en testing basado en propiedades [7, 21], y toman la forma de relaciones binarias entre parámetros y/o constantes (ver Tabla 1.1 a modo de ejemplo).

Las restricciones especificadas sobre los parámetros denotan un digrafo en el cuál cada nodo es un parámetro, y cada arista (u, v) surge de una restricción (relación) cuyo lado izquierdo es u y cuyo lado derecho es v. Esto representa una dependencia sobre v por parte de u. En tiempo de ejecución este grafo deberá ser recorrido en un orden topológico, aplicando la restricción que cada relación denota. Por esta razón no permitimos al usuario especificar relaciones que denoten ciclos en el grafo.

2.2. Algoritmo de fuzzing *in-vivo*

En Algorithm 2 se detalla el proceso de *in-vivo fuzzing*. Toma como entrada los puntos de amplificación y restricciones provistas por el usuario, una instancia del host instrumentado, y dos presupuestos de tiempo t_0 y t_1 (los cuales determinarán el largo de las dos fases en las que está divida la campaña de fuzzing).

A lo largo de la campaña se mantienen dos corpus globales, Q y $Q_{\text{✗}}$, y un corpus local Q_f por cada función $f \in F$. Cada entrada en los corpuses globales registra a qué punto de amplificación está asociada la entrada, junto con la versión serializada de los argumentos provistos a la misma. En los corpuses locales, en cambio, el punto de amplificación

Algorithm 2 In-Vivo Fuzzing – Function `fuzz`

Input: Amplifier points F, Types T, Constraints C, Instrumented process p, Time budgets t_0 and t_1
Output: Crashes $Q_{\text{✗}}$

1: Global corpus $Q = \emptyset$
2: Local corpus $Q_f = \emptyset$ for all $f \in F$
3: Crashes $Q_{\text{✗}} = \emptyset$
4: Shadow process $p' = \texttt{fork}(p)$
5: **for each** function $f \in F$ executed in p **do**
6: Objects $objs = \texttt{collect\_initial\_args}(p', f)$
7: Types $t \in T$ corresponding to f
8: Args $args = \texttt{serialize}(objs, f, t)$
9: Add $args$ to local corpus Q_f
10: Add $\langle f, args \rangle$ to global Q
11: **end for**
12:
13: **for each** function $f \in F$ executed in p **do**
14: **while** t_0 not expired **do**
15: Args $q = \texttt{select}(Q_f)$
16: For f, find types $t \in T$ and constraints $c \in C$
17: $\texttt{fuzz\_function\_args}(p', f, t, c, q, Q, Q_f, Q_{\text{✗}})$
18: **end while**
19: **end for**
20: **while** campaign not aborted and t_1 not expired **do**
21: Tuple $\langle f, q \rangle = \texttt{select}(Q)$
22: For f, find types $t \in T$ and constraints $c \in C$
23: $\texttt{fuzz\_function\_args}(p', f, t, c, q, Q, Q_f, Q_{\text{✗}})$
24: **end while**
25: **return** $Q_{\text{✗}}$

correspondiente ya es conocido de antemano, por lo que solo se almacenan los bytes que codifican a los argumentos. Q y Q_f incluyen a todas las entradas que incrementaron la cobertura en la librería, mientras que $Q_{\text{✗}}$ incluye a todos los inputs que indujeron carshes durante la ejecución. Es importante destacar que cada vez que se mutan los argumentos de un punto de amplificación f y se registra un incremento en cobertura de código, se agrega la entrada correspondiente a Q y Q_f por igual, por lo que toda entrada está duplicada en el corpus global y en uno local.

En la linea 4 se crea un fork del proceso p, el cual se va a utilizar para amplificar ejecuciones sin interferir con el proceso original. Además de asumir la no-interferencia con el proceso original, asumimos que la ejecución del mismo puede (conceptualmente) rebobinarse hasta la invocación de cualquier punto de amplificación. Vamos a hacer uso de esta funcionalidad para alternar entre las diferentes funciones objetivo durante la campaña.

En el loop comprendido entre las lineas 5 y 11 el fuzzer recolecta las semillas iniciales para la campaña a partir del fork de la ejecución. Estas semillas van a ser las utilizadas para llevar a cabo fuzzing mutacional guiado por cobertura. Para lograr esto, nuestro prototipo ejecuta el host en su totalidad, interceptando las llamadas que este realice a

puntos de amplificación en el target (lo cual es posible gracias a la instrumentación insertada en tiempo de compilación) y guardando una copia de los argumentos provistos por el host. Estos argumentos son posteriormente serializados y agregados a los corpuses correspondientes.

En las lineas 13 a 19 se lleva a cabo la fase de exploración. En esta se dedica un tiempo mínimo de fuzzing a cada punto de amplificación en F para recolectar información de cobertura necesaria para el loop de fuzzing principal. El presupuesto de tiempo t_0 asignado a cada punto de amplificación es determinado por el usuario. Si no se llevara a cabo esta fase exploratoria, todos los puntos de amplificación serían considerados igual de efectivos a fines de aumentar cobertura, y el fuzzer no podría valerse de la información de cobertura para elegir las mejores entradas [6]. Esto es porque durante la ejecución inicial, la cual no es amplificada, las entradas iniciales correspondientes a cada punto de amplificación reportan exactamente la misma cobertura de código. Esta fase, entonces, fuerza al fuzzer a explorar qué regiones de código se puede llegar a cubrir a partir de cada punto de amplificación. Dentro del cuerpo de este loop, se realiza una pequeña campaña de fuzzing local a cada función durante una cantidad de tiempo t_0. En la línea 15 se elige una entrada del corpus local al punto de amplificación correspondiente, y en la linea 17 se la utiliza para mutarla y ejecutarla.

En las líneas 20 a 24, el fuzzer continúa con la campaña, esta vez tomando entradas tomadas del corpus global (junto con su función correspondiente). Para seleccionar la semilla a utilizar, se utilizan las heurísticas implementadas sobre el fuzzer subyacente (en este caso, `AFL++`).

2.3. Fuzzing mutacional de argumentos de funciones

Algorithm 3 Function `fuzz_function_args`

Input: Process p', function f, types t, constraints c, args q, Global corpus Q, Local corpus Q_f, Crashes $Q_{\text{✗}}$
Output: Global corpus Q, Local corpus Q_f, Crashes $Q_{\text{✗}}$

1: Energy e = `compute_energy`(f, q, Q)
2: **while** e not expired **do**
3: Mutated args q' = `mutate`(q)
4: Mutated objs o = `deserialize`(q', t, c)
5: Process p'' = `fork_rewind_wait`(p', f)
6: Result r = `substitute_continue`(p'', f, o)
7: **if** r = new crash detected **then**
8: Add $\langle f, q' \rangle$ to $Q_{\text{✗}}$
9: **else if** r = coverage increased **then**
10: Add $\langle f, q' \rangle$ to Q
11: Add q' to Q_f
12: **end if**
13: **end while**
14: **return** Q, Q_f, $Q_{\text{✗}}$

En Algorithm 3 se presenta el procedimiento `fuzz_function_args`, utilizado en las líneas 17 y 23 de Algorithm 2. El mismo toma como entradas un fork del proceso original,

el punto de amplificación, los tipos de los parámetros con sus restricciones, y la semilla. Se encarga de mutar la semilla para generar argumentos alternativos, y utiliza estas variantes para remplazar los argumentos provistos por el host al punto de amplificación. Cuando los argumentos generados ejerciten nuevas regiones de código, son agregados a los corpuses Q y Q_f, mientras que los que induzcan crashes son agregados al conjunto resultado $Q_{\text{✗}}$.

Primeramente, se determina la cantidad e de mutaciones óptima a aplicar a partir de la semilla dada. Luego se procede a mutar la versión serializada de los argumentos e veces, obteniendo en cada iteración q' una entrada distinta. Tanto para determinar la energía de una semilla como para mutarla, reutilizamos los operadores presentes en el fuzzer subyacente.

En la línea 4, el runtime recibe la secuencia de bytes provista por el fuzzer y la convierte en valores del tipo correspondiente utilizando la información sobre los tipos recabada en tiempo de instrumentación. Este proceso es determinístico: Deserializar la misma secuencia de bytes múltiples veces resulta en los mismos argumentos siendo generados, lo cual asegura que los crashes encontrados sean reproducibles. El proceso de deserialización también garantiza la satisfacción de las restricciones c especificadas por el usuario. El funcionamiento de la función `deserialize` se discute en profundidad en Sección 2.4.

En las líneas 5 y 6 se crea un segundo fork del proceso, el cual se rebobina hasta la *primera invocación* del punto de amplificación. Esta funcionalidad puede implementarse en una máquina virtual utilizando un mecanismo de *snapshots*, pero esto implicaría un alto costo en términos de tiempo de ejecución y de consumo de memoria. En nuestro prototipo esta funcionalidad se logra re-ejecutando el programa hasta que se alcance el punto deseado (lo cual, asumiendo determinismo por parte del host, equivale conceptualmente a rebobinarlo), e iniciando un *fork-server* en el punto de amplificación. Este interactúa con el fuzzer para recibir los argumentos a proveerle a la función objetivo. Es importante destacar que al implementar la funcionalidad de *rewinding* de esta manera estamos limitándonos a hosts determinísticos.

En las líneas 7 a 12, el fuzzer agrega a los respectivos corpus las entradas que permitieron incrementar la cobertura del target, o las que generaron crashes en el proceso.

2.4. Serialización y deserialización

Para poder reutilizar greybox fuzzers para implementar la selección de semillas, priorización y mutación (`select` en Algorithm 2, y `compute_energy` y `mutate` en Algorithm 3, respectivamente), necesitamos contar con un mecanismo que nos permita convertir conjuntos de argumentos a secuencias de bytes (con las que el fuzzer pueda operar), y viceversa. Las funciones que realizan dicha conversión (`serialize` y `deserialize`) están comprendidas en la librería de runtime contra la cual se enlaza al binario instrumentado (ver Figura 2.1). Notar que este enfoque es similar al propuesto por Padhye et al. [21] para aplicar fuzzing sobre lenguajes orientados a objetos, como `Java`.

En Algorithm 4 se presenta el proceso de deserialización de argumentos. El inverso es análogo. Dada una secuencia de bytes, los tipos de los argumentos, y sus restricciones, el algoritmo de deserialización construye valores de los tipos correspondientes para proveer a la función objetivo. En las líneas 1 a 5, se genera un objeto por cada argumento utilizando la información de los tipos. En las líneas 7 a 25 se expone la función recursiva `deserialize_arg`, función que se encarga también de garantizar la validez de las restricciones especificadas por el usuario.

Algorithm 4 Function `deserialize`

Input: Function argument byte sequence q, Function argument types t, Function argument constraints c
Output: Function argument objects o

```
1:  Objects o = ⟨⟩
2:  for type in t do
3:      Object obj = deserialize_arg(q, type, c)
4:      append(o, obj)
5:  end for
6:
7:  function deserialize_arg(q, type)
8:      Object obj
9:      if type is primitive then
10:         obj = q.consume_bytes(type.bitWidth / 8)
11:         enforce_constraints(obj, c)
12:     else if type = Struct then
13:         for field, fieldType in type.fields do
14:             obj.field = deserialize_arg(q, fieldType, c)
15:         end for
16:     else if type = Pointer then
17:         type' = type.pointeeType
18:         length = q.consume_bytes(4)
19:         enforce_constraints(length, c)
20:         for i ∈ {0, ..., length − 1} do
21:             obj[i] = deserialize_arg(q, type', c)
22:         end for
23:     end if
24:     return obj
25: end function
```

La secuencia de bytes q se "consume" de manera tal que cada byte se utiliza una única vez a lo largo del proceso de construcción de argumentos. La función `consume_bytes` mantiene un desplazamiento dentro del buffer provisto por el fuzzer, que inicialmente toma el valor 0. Al ser llamada, esta función lee del buffer la cantidad de bytes indicada a partir del desplazamiento actual, y actualiza el valor del desplazamiento. Si durante el proceso de deserialización se intentan leer más bytes de los que existen en el buffer provisto por el fuzzer, el mismo se rellena con 0.

Para deserializar valores de tipos de datos primitivos (`char`, `float`, etc), simplemente se consume la cantidad de bytes correspondiente al tamaño del tipo de dato y se realiza un *cast* al tipo apropiado (líneas 9 a 11).

Para deserializar estructuras de datos primero se reserva el espacio suficiente para albergar un valor de esta estructura, y luego se aplica recursivamente la misma función sobre cada campo de la misma.

El caso de los punteros resulta patológico, dado que parámetros de este tipo suelen tener semánticas complejas definidas por los desarrolladores. En algunos contextos los punteros son tratados como direcciones de memoria de otros elementos y dereferenciados

de manera incondicional, mientras que en otros contexto los punteros pueden ser nulos sin que esto implique un defecto en el código. Por otro lado, dado que nuestra implementación opera sobre la representación intermedia de `LLVM`, los punteros resultan indistinguibles de los arreglos. Para abordar esta complejidad recurrimos a tratar a los punteros, en todos los contextos, como punteros a arreglos (siendo los punteros a un solo elementos arreglos de largo 1, y punteros nulos arreglos de largo 0). Al intentar deserializar un puntero, primero se lee un entero de 4 bytes del buffer, y este se trata como la longitud del arreglo a deserializar (línea 18). Posteriormente, se reserva la cantidad de memoria necesaria y se aplica recursivamente la misma función sobre cada uno de los elementos (líneas 20 a 22).

La serialización (utilizada en la línea 8 de Algorithm 2), funciona de manera análoga, convirtiendo a los valores dados en bytes y concatenándolos a un buffer resultado q. Si se aplica Algorithm 4 sobre el q obtenido como resultado de la serialización se recuperarán los mismos valores iniciales (módulo direcciones de memoria). La mayor complejidad del proceso de serialización se encuentra, nuevamente, en el caso de los punteros. Sin información de contexto, no podemos predecir si un puntero hace referencia a un elemento, a un arreglo de elementos, o si no apunta a nada. Para salvar la ambigüedad, dependemos de las restricciones especificadas por el usuario para determinar el largo del "arreglo" al cual hace referencia cada puntero (nuevamente, bajo la interpretación de que todos los punteros hacen referencia a arreglos, aunque sean de largo cero, uno, o más). Si las restricciones planteadas por el usuario no son lo suficientemente informativas como para determinar unívocamente el tamaño de dicho arreglo recurrimos a las siguientes heurísticas: Tratamos los punteros a bytes como cadenas de caracteres terminadas cen bytes nulos e intentamos determinar su largo utilizando la rutina `strlen`. Para todos los demás casos, asumimos que el puntero se refiere a un arreglo de largo 1. Si las restricciones provistas por el usuario son insuficientes y el uso de estas heurísticas resulta errado, esto se evidenciará en tiempo de ejecución a modo de accesos inválidos a memoria.

Cabe destacar que al operar sobre tipos de datos en la representación intermedia de `LLVM` estos 3 tipos de datos cubren la mayoría de los tipos que pueden expresarse en lenguajes de alto nivel como `C`. Algunos tipos del lenguaje `C` no son tratados por nuestra implementación, tales como punteros a funciones. Otro tipos compilan a alguno de estos tipos básicos (los *union* compilan al tipo más grande dentro la unión, los punteros a *void* compilan a putneros a byte, etc), pero conservan una semántica distinta a la de los tipos a los cuales son reducidos. Estos casos también quedan fuera de nuestro análisis.

Adicionalmente, le permitimos al usuario indicar, mediante una opción en un archivo de configuración, si durante una ejecución normal del host (es decir, fuera de la campaña del fuzzer) debe aplicarse el procedimiento de serialización de argumentos. Esto lo hacemos con fines de *debugging*, ya que durante el desarrollo de este trabajo encontramos muy dificultoso depurar errores en la implementación que solo surgen cuando el proceso es ejecutado por el fuzzer.

2.4.1. Aplicación de restricciones

Las restricciones especificadas por el usuario toman la forma de relaciones (nuestro prototipo permite únicamente las relaciones $<$, $>$, $=$) entre pares de argumentos o argumentos y constantes (Tabla 1.1). En Algorithm 5 se presenta el algoritmo utilizado para aplicar este tipo de restricciones sobre los valores deserializados en Algorithm 4 (líneas 11 y 19).

Es importante destacar que las restricciones predican siempre sobre valores escalares.

Algorithm 5 Function `enforce_constraints`

Input: Scalar s, Argument constraints c
Output: Scalar s satisfies all constraints in c

```
for lhs, rel, rhs in c do
    if lhs matches current argument then
        if rel = eq then
            s ← rhs
        else if rel = le then
            s ← min(s, rhs)
        else if rel = ge then
            s ← max(s, rhs)
        end if
    end if
end for
```

Esto es porque las mismas predican sobre tipos de datos primitivos (todos los cuales pueden ser interpretados como escalares) o sobre la cantidad de elementos en un arreglo. Dados los escalares involucrados en la restricción y el tipo de relación, basta con aplicar el operador correspondiente sobre los valores del lado izquierdo y derecho de la restricción para obtener un nuevo valor para el lado izquierdo que cumpla la restricción indicada.

Una precondición que asumen Algorithm 4 y Algorithm 5 es que al momento de deserializar un argumento que es referenciado en el lado izquierdo de una relación, ya conocemos el valor del lado derecho. Esto implica que el conjunto de restricciones denota un digrafo de dependencias entre los valores de los argumentos y, durante la deserialización, debemos recorrer los argumentos en este grafo en un orden topológico. En caso de existir más de un único orden topológico posible, podemos tomar uno arbitrario, con la única condición de que el ordenamiento utilizado sea el mismo a lo largo de toda la campaña. A su vez, esto implica que el conjunto de restricciones debe denotar un digrafo libre de ciclos, para que dicho orden exista.

Además de permitir especificar restricciones binarias entre argumentos, permitimos al usuario etiquetar parámetros de tipo `char *` (más generalmente, cualquier puntero a bytes) como *nombres de archivo*. Cuando tal restricción es encontrada, el runtime vuelca el buffer entero provisto por el fuzzer a un archivo temporal, y remplaza el valor del argumento por el nombre del archivo.

2.5. Punto de fork y terminación temprana

Si bien la ejecución del host nos provee con un contexto de ejecución realista, ejecutar el host entero de principio a fin por cada entrada que se desee testear puede implicar una penalidad de performance que anule por completo la utilidad de esta técnica. A fin de evitar esta situación, iniciamos el forkserver dentro de la instrumentación inmediatamente antes de ejecutar el punto de amplificación a testear. Sin embargo, esto en algunos casos no resulta deseable. En particular, si el punto de amplificación es llamado luego de adquirir un recurso que sea compartido entre los procesos padre e hijo (como por ejemplo, abrir un archivo, un socket, etc), esto puede generar grandes problemas de inestabilidad en la campaña de fuzzing (entendiendo a esta como la posibilidad de que el mismo input,

ejecutado dos veces, ejerza dos caminos distintos y esto entorpezca la campaña de fuzzing). Es por esto que mediante un archivo de configuración le permitimos al usuario especificar si el forkserver debe ser iniciado junto con el programa, o si debe diferirse hasta la ejecución del punto de amplificación de turno.

Análogamente, una vez amplificada la ejecución, esperar a que la misma termine normalmente puede llevar un tiempo lo suficientemente largo como para hacer mermar la eficiencia de la campaña de fuzzing. Intuitivamente, esperamos que los crashes que se generen producto de una amplificación se den poco tiempo después de que el punto de amplificación relevante haya sido invocado. Por esta razón permitimos al usuario indicar también una cantidad de milisegundos que deben esperarse luego de ejecutar el punto de amplificación antes de terminar forzosamente la ejecución del fork.

Con estas dos funcionalidades, le permitimos al usuario controlar cuándo debe iniciarse el forkserver y cuándo detener la ejecución de los forks. Esto le permite experimentar con el host hasta lograr un equilibrio entre performance y estabilidad durante el proceso fuzzing.

2.6. Reproducción de crashes

Una vez finalizada la campaña, se obtiene como resultado el conjunto de entradas que ocasionaron crashes en el host, junto con el nombre de las funciones asociadas con cada entrada. El usuario podrá entonces reproducir y examinar la causa de cada crash haciendo uso de la funcionalidad de reproducción de crashes (ver Figura 2.1).

Para permitir la reproducción de crashes no basta con persistir la entrada causante del mismo y el nombre de la función a la cual estaba dirigida. Necesitamos también permitir la amplificación de una ejecución autocontenida del host (es decir, sin estar comprendida esta en el contexto del fuzzer). Con este fin permitimos al usuario especificar una función sobre la cual se amplificará la ejecución independientemente de ocurrir esta fuera de una campaña de fuzzing. Cuando el punto de amplificación relevante sea invocado, este tomará el buffer de entrada (q en Algorithm 4) de `stdin`, de manera tal que el usuario pueda proveer la entrada deseada de manera manual.

2.7. Identificación de puntos de amplificación

Para identificar automáticamente potenciales puntos de amplificación sobre un target inicialmente desconocido utilizamos una *query* sobre la herramienta de análisis estático `CodeQL` [13]. La misma intenta identificar funciones asociadas con parsing de texto o formatos binarios. Para esto identificamos todas las funciones cuyo nombre contenga a alguna palabra de una lista de palabras claves (por ejemplo, “parse”, “decode”, “process”). Adicionalmente, requerimos que esta función reciba dos parámetros contiguos, donde uno sea un puntero a bytes y otro un entero, con la expectativa de que este par de parámetros denote un buffer a parsear y el largo del mismo. Tales funciones reciben el nombre de “interesantes”. La definición de funciones “interesantes” es recursiva, ya que también lo son todas las funciones que llamen a otra función interesante, pasándole dos de sus argumentos sin aplicarles ninguna modificación.

Esta heurística nos provee una lista de potenciales puntos de amplificación que luego deberá ser recorrida manualmente por el usuario removiendo las funciones inapropiadas. Para las funciones seleccionadas el usuario debe proveer una lista de restricciones como

las expuestas en Tabla 1.1, la cual será procesada por un parser descendiente recursivo y transformadas a un formato `JSON` para ser consumido por nuestro runtime.

3. EXPERIMENTACIÓN

Para contrastar la efectividad de esta técnica vamos a dividir la evaluación en tres partes, de acuerdo a tres casos de uso de nuestro prototipo. A saber:

- Fuzzing *in-vivo*: Dada una librería, un host que la utilice, y una ejecución del mismo, amplificamos esta ejecución (que presumimos análoga a la que se produciría en un entorno productivo de la librería) con el objetivo de entender las implicancias que esta puede tener del punto de vista de seguridad.

- *Autoharnessing*: Dada una librería, buscamos amplificar la ejecución de *algún* host que la utilice, como método para incorporar fuzz testing a una librería sin necesidad de escribir un harness. En este caso utilizamos herramientas del estado del arte de generación de harnesses automatizada como punto de comparación.

- Amplificación de casos de test: Si tratamos a la testsuite de una librería como host esperamos ejecutar una mayor cantidad de puntos de amplificación, y esperamos también que las ejecuciones amplificadas cubran casos borde dentro de la librería.

3.1. Fuzzing *in-vivo*

3.1.1. Setup experimental

Subject	Type	#LOC	#Stars	Version
`boringssl`	Encryption	483.2k	1.6K	dd52194
`bzip2`	Compression	8.2k	N/A	1.0.8
`libass`	Rendering	35.4k	810	0.17.1
`libexif`	Parsing	30.7k	234	0.6.24

Tab. 3.1: Información sobre los sujetos seleccionados.

Para este caso de uso amplificamos ejecuciones de cuatro librerías diferentes a lo largo de campañas de 24 horas de duración. En Tabla 3.1 se pueden ver los detalles de los sujetos seleccionados. Los mismos fueron tomados de manera arbitraria, intentando cubrir una amplio espectro de aplicaciones, incluyendo criptografía y renderizado. Adicionalmente estas librerías suelen utilizarse para procesar datos no confiables o potencialmente controlados por atacantes, lo cual implica que errores presentes en ellas pueden implicar vulnerabilidades de seguridad. Se utilizó el script decripto en Sección 2.7 para identificar puntos de amplificación en cada uno de estos proyectos.

En Tabla 3.2 se presentan los hosts utilizados para cada sujeto, junto con cuantos puntos de amplificación se lograron ejecutar por cada host. Para cada librería, se seleccionó un host que estuviera desarrollado por el mismo grupo que desarrolla la librería, o que haya sido promovido por el mismo. De esta manera minimizamos el riesgo de que los crashes encontrados se deban a una secuencia de invocaciones la API por parte del host que no respete el protocolo de la librería, y no a un error en la misma. Para `boringssl` recurrimos

Subject	Host	#Executed ampl. points
boringssl	crypto_test	37
bzip2	bzip2	1
libass	ffmpeg	4
libexif	photographer	2

Tab. 3.2: Hosts used for each subject's fuzzing campaign

a usar un binario de prueba a modo de host, dado que la documentación oficial de este proyecto desalienta su uso en proyectos externos, y por lo tanto no existen hosts externos que la utilicen (dicha librería es utilizada internamente por Google en proyectos como Chrome o Android, los cuales no pueden ser testeados por fuzzers como AFL++). bzip2 provee su propia aplicación junto con la librería, la cual utilizamos en nuestros experimentos para descomprimir un archivo de prueba (cuyo contenido original es un archivo de texto que contiene el pangrama inglés: *the quick brown fox jumped over the lazy dog*). Para libass utilizamos ffmpeg como host, cuya ejecución fue generada agregando subtítulos a un archivo de video minimal. Para libexif utilizamos una aplicación de ejemplo adjunta en el código fuente de la librería, la cual se utiliza para decodificar metadata de una imagen de prueba para generar la ejecución original.

Para cada sujeto, se realizaron 20 campañas de fuzzing concurrentes de 24 horas cada una sobre un procesador AMD EPYC 7713P 64-Core con 256GB de RAM. Cada campaña se corrió dentro de un contenedor Docker con 2 núcleos asignados a cada uno (sin permitir solapamiento). Durante la ejecución de estas campañas se minimizó la utilización del sistema para evitar interferencias. De estas campañas nos interesa evaluar la cobertura de código, crashes encontrados y tasa de falsos positivos.

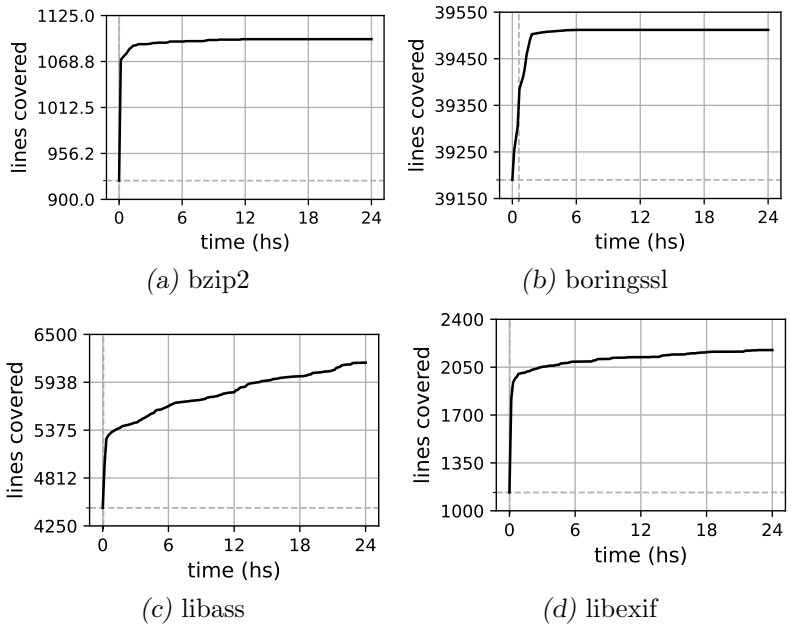
Al inicio de cada campaña, el fuzzer reserva 25MB de memoria por cada punto de amplificación, región en la cual se serializarán los argumentos iniciales al inicio de la campaña. Adicionalmente, el fuzzer reserva otros 25MB de memoria, donde almacenará los argumentos serializados para ser utilizados por el punto de amplificación siendo testeado. Esta implementación resulta en un uso de RAM ligeramente superior al dado por AFL++, por lo que asumimos que la cantidad de memoria instalada en la computadora sobre la cual se llevaron a cabo los experimentos es suficiente.

3.1.2. Resultados

Los resultados se muestran en Figura 3.1. Allí se muestra la cobertura promedio de las 20 ejecuciones, tomada a intervalos de 10 minutos, para cada sujeto. Las líneas punteadas verticales y horizontales indican cuándo se terminó la fase exploratoria y la cobertura lograda por la ejecución sin amplificar, respectivamente.

Los mayores incrementos en términos de cobertura se lograron en libass y libexif, donde logramos un incremento del 38,24 % (1706 líneas) y del 91,79 % (1040 líneas) sobre las ejecuciones originales, respectivamente. Para bzip2 y boringssl logramos un incremento del 18,75 % (173 líneas) y del 0,82 % (321 líneas), respectivamente.

La falta de incremento en cobertura de código en boringssl se refleja en Figura 3.1b, donde vemos que se alcanza una meseta en la curva una hora después del inicio de la campaña. Si bien la cobertura de código no se incrementó pasadas las primeras horas, sí se



(a) bzip2 *(b)* boringssl

(c) libass *(d)* libexif

Fig. 3.1: Cobertura de código promedio vs tiempo para los sujetos de prueba a lo largo de la campaña. La línea puntada horizontal indica la cobertura lograda por la ejecución original no-amplificada. La línea vertical punteada indica *cuándo* se terminó la fase de exploración y se pasó al loop principal.

agregaron nuevas entradas al corpus a lo largo de toda la campaña, lo cual denota que se lograron ejecutar caminos nuevos dentro del CFG de la librería, pero no se cubrieron nuevas regiones de código. Este comportamiento se lo atribuimos al hecho de que de las 2044 funciones ejecutadas por el host, 1381 ya estaban saturadas en términos de cobertura por la ejecución sin amplificar. Esto implica que había poco espacio para que la amplificación de la ejecución cubriera nuevas regiones de código.

En `libass` y `libexif` observamos un incremento sostenido de la cobertura, el cual parecería poder extenderse más allá del presupuesto asignado por nuestro experimento. Esto sugiere que la amplificación de ejecuciones puede ser ampliamente beneficiosa a los efectos de la búsqueda de vulnerabilidades si las ejecuciones originales ejercitan regiones de código "profundas".

Durante ninguna de las campañas se registraron crashes. Naturalmente, esto implica que no se reportaron falsos positivos, aunque es importante destacar que la tasa de falsos positivos es altamente dependiente de la calidad de las restricciones especificadas. Tampoco se reportaron errores por falta de memoria (*out of memory*), de lo cual inferimos que el consumo de memoria RAM se ajustó a las prestaciones de la computadora en la que se llevaron a cabo las pruebas.

3.2. *Autoharnessing*

Una ventaja de fuzzing in-vivo es que nos permite prescindir de la necesidad de un harness para aplicar fuzz testing sobre una librería. El harness sería el encargado de crear un contexto de ejecución, aceptar datos del fuzzer, darles el formato apropiado y utilizarlos para estimular a la librería. Mediante amplificación de ejecuciones, el primero de estos

Generator	Library	Synth. fuzz driver	(LoC)	Host	#Amp. points
FuzzGen	libaom	av1_dec_fuzzer	1131	aomdec	4
	libvpx	simple_decoder	482	vpxdec	5
	libgsm	cod2lin	371	STL/rpedemo	3
FUDGE	htslib	hts_open	152	samtools	2
	leptonica	pix_rotate_shear	68	tesseract	18

Tab. 3.3: Resumen del setup para cada sujeto en la comparación contra FUDGE/FuzzGen.

pasos lo lleva a cabo el host, mientras que los demás se dividen entre el host y el runtime de nuestra herramienta.

Existen en la práctica diversos enfoques que buscan generar harnesses de manera automática. Por ejemplo, `FuzzGen` [17] busca inferir la interfaz de una librería y sintetiza drivers para ejercitarla. `FUDGE` [3] busca usos de la API de una librería y utiliza *program slicing* [29] para extraer los fragmentos de código correspondientes, a partir de los cuales sintetiza los harnesses. `IntelliGen` [31] también infiere la interfaz de la librería y genera drivers para cada punto de entrada en la misma. `Daisy` [32] analiza dinámicamente cómo el host invoca a la librería, y luego sintetiza drivers que sigan el mismo patrón de llamadas.

Un enfoque alternativo que busca remplazar a los harnesses (en lugar de generarlos) es `GraphFuzz` [14]. El mismo se basa en inferir dependencias entre las funciones de una librería, y en tiempo de ejecución genera llamadas que satisfagan estas dependencias en un orden aleatorio.

Sin embargo, estos enfoques interactúan con la librería de manera muy superficial. Las interacciones que se dan entre las librerías y los harnesses generados no necesariamente reflejan las que se darían en aplicaciones reales. Nuestra propuesta es, entonces, amplificar usos reales de la librería, en lugar de *imitar* un uso real de la misma.

Para contrastar la efectividad de nuestra herramienta en esta aplicación, la comparamos contra el estado del arte en generación automática de harnesses.

3.2.1. Setup experimental

Consideramos en esta comparación enfoques que se centren en librerías escritas en `C`, y que estén públicamente disponibles o que hayan publicado drivers generados para poder reproducir las campañas. Esto nos da como resultado a `FuzzGen` y `FUDGE`. No pudimos incluir `GraphFuzz` ya que el mismo está diseñado para librerías orientadas a objetos, las cuales no son fácilmente tratadas por nuestro prototipo. En el caso de `Daisy`, si bien se cuenta con los harnesses generados por esta herramienta, no tuvimos éxito al intentar compilarlos debido a dependencias faltantes. `IntelliGen` no disponibiliza ni el código de la herramienta ni los harnesses generados, por lo que tampoco pudo ser incluido en esta comparación. `FUDGE` no es de acceso público, pero sí se publicaron algunos harnesses generados por la herramienta. Si bien FuzzGen es de acceso público, no pudimos utilizarlo para generar nuevos harnesses, pero pudimos valernos de los harnesses publicados por los autores.

Para la comparación contra `FuzzGen` seleccionamos tres de las siete librerías para los cuales se publicaron harnesses, las cuales se pueden ver en Tabla 3.3. De las cuatro librerías excluidas, tres tenían una API que consistía de una única función que aceptaba una estructura de datos cuyos campos codificaban el propósito de la llamada y el estado de la librería. Estas interfaces no podían ser capturadas por el sistema de restricciones

Subject	Type	#LOC	#Stars	Version
`libaom`	Video Codec	693k	N/A	3613e5d
`libvpx`	Video Codec	516	831	1.12.0
`libgsm`	Speech compressor	8.65k	12	1.0.22
`htslib`	File Parser	99k	723	1.16
`leptonica`	Image Processing	320K	1.5K	1.83.0

Tab. 3.4: Información de los sujetos considerados.

implementado en este trabajo. La librería restante contaba con un harness defectuoso, que no pudimos arreglar fácilmente. En cuanto a los harnesses utilizados para cada sujeto, los autores de `FuzzGen` publican un único driver para `libaom` y `libvpx`, y para `libgsm` tomamos aleatoriamente al driver `cod2lin`. Todos estos harnesses intentan decodificar un arreglo de bytes provisto por el fuzzer.

En el caso de FUDGE, tomamos todos los harnesses resaltados en el paper, con la excepción de `OpenCV`, ya que esta librería es orientada a objetos y, por lo tanto, no puede ser tratada por nuestro prototipo. `leptonica` y `htslib` (ver Tabla 3.3) son librerías populares para el procesamiento de imágenes y formatos de datos secuenciales, respectivamente. Los drivers sintetizados por FUDGE para ambas librerías intentan decodificar los datos provistos por el fuzzer con alguno de los formatos soportados por las respectivas librerías. En Tabla 3.1 se presenta la información detallada de cada sujeto utilizado.

En Tabla 3.3 se detallan los hosts utilizados para cada target. Para que la comparación sea lo más justa posible, incluimos en el corpus de cada harness sintético una entrada equivalente a la provista al host durante la amplificación de ejecuciones. Con equivalente nos referimos a una entrada tal que la haga que la librería reciba las mismas entradas durante una primera ejecución del harness, en los casos en donde esto sea posible (por ejemplo, para los decoders de video, intentamos que tanto el host como el harness decodifiquen el misma *stream*). Al igual que en Subsección 3.1.1, intentamos seleccionar hosts que hayan sido desarrollados o promovidos por el mismo grupo que desarrolla el target. Con esto buscamos minimizar la probabilidad de que los eventuales crashes se deban a un uso erróneo de la librería y no a un defecto en la misma.

Para `libaom` y `libvpx` usamos como hosts los decodificadores incluidos en la distribución de las librerías, y generamos la ejecución de los mismos decodificando el *stream* más chico posible de cada formato. Para `libgsm` utilizamos una utilidad de línea de comandos para procesar streams gsm[1], y generamos la ejecución de manera análoga a las anteriores. Para `leptonica` utilizamos `tesseract`, una herramienta de reconocimiento de caracteres, y generamos la ejecución utilizando esta herramienta para reconocer texto escrito en inglés en una imagen, tal como se presenta en la documentación del proyecto[2]. Para `htslib` utilizamos la aplicación `samtools`, desarrollada como parte de una misma iniciativa y que hace las veces de host canónico. Generamos la ejecución original procesando unos archivos de prueba en formato `sam` y `fa`, incluidos como datos de prueba en la librería.

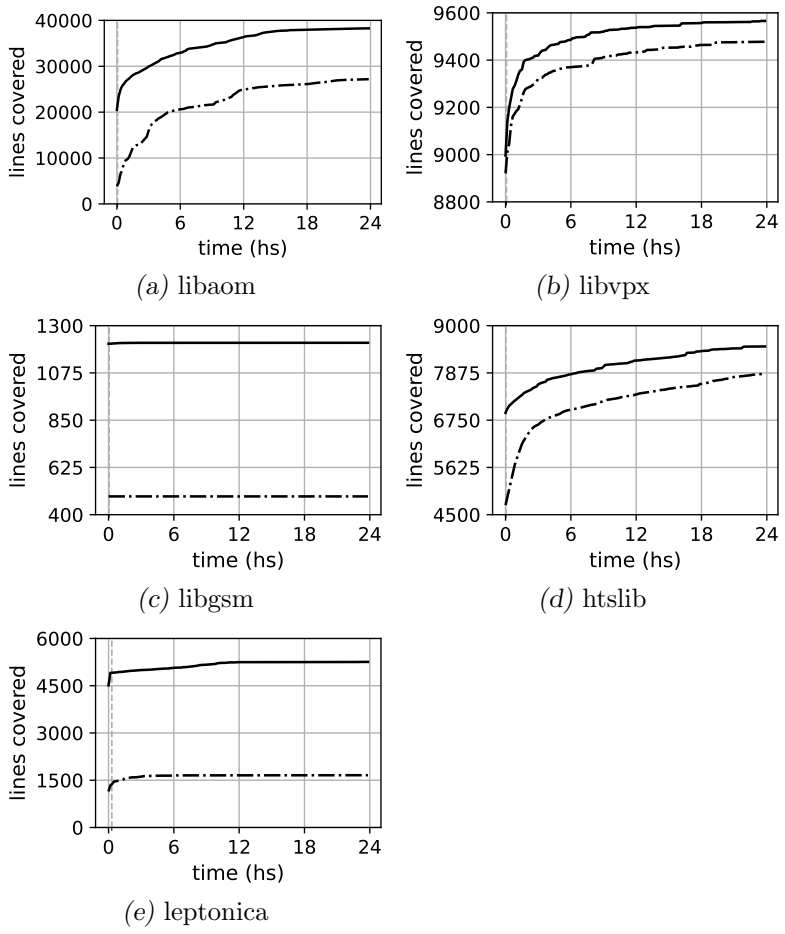
En Tabla 3.3 se presenta cuántos puntos de amplificación de los identificados por nuestro análisis (Sección 2.7) se ejecutaron por cada host.

Para cada uno de los cinco sujetos de prueba, ejecutamos 20 campañas de in-vivo fuzzing y de fuzzing sobre el harness sintético. Nuevamente, la experimentación se llevó a

[1] `https://github.com/openitu/STL/blob/dev/src/rpeltp/rpedemo.c`
[2] `https://tesseract-ocr.github.io/tessdoc/Command-Line-Usage.html`

cabo una sobre un procesador `AMD EPYC 7713P 64-Core` con `256GB` de `RAM` en containers de `Docker` para cada campaña.

3.2.2. Resultados



(a) libaom *(b)* libvpx

(c) libgsm *(d)* htslib

(e) leptonica

Fig. 3.2: Cobertura de código vs tiempo para in-vivo fuzzing (sólida) y estado del arte (punteada).

En Figura 3.2 se presenta la cobertura de código promedio en función del tiempo para todas las campañas. Las lineas punteadas verticales indican cuándo finalizó la fase exploratoria y empezó el ciclo principal de fuzzing (notar que en los casos donde solo se amplificó una cantidad reducida de funciones esta línea resulta apenas visible).

En todas las campañas nuestro prototipo logra una mayor cobertura que los drivers sintéticos. Adicionalmente, ambos enfoques se estabilizan casi simultáneamente en cada caso. Sin embargo, in-vivo logra cubrir una cantidad significativamente mayor de código antes de alcanzar su meseta. Resulta interesante que el fuzzer in-vivo cuenta consisténtemente con una mayor cobertura inicial en comparación a los harnesses sintéticos. Esto se lo atribuimos a que los harnesses suelen invocar a un conjunto reducido de funciones en la API de la librería, mientras que los hosts suelen interactuar con los targets de maneras más complejas (incluyendo secuencias de inicialización y de-inicialización más complejas). Los harnesses sintéticos aparentemente no fueron capaces de imitar estas interacciones.

El caso de `libgsm` resulta, a primera vista, patológico, ya que se logra un incremento de la cobertura muy bajo para ambas herramientas. Esta observación motivó una inspección manual sobre la librería, mediante la cual pudimos comprobar que las rutinas siendo testeadas por ambos fuzzers (`gsm_implode`, `gsm_decode`, `gsm_encode`) constan de secuencias de código sin instrucciones de control de flujo. Resulta esperable, entonces, que ejecutar estas funciones variando sus argumentos no permita obtener distintos resultados en términos de cobertura de código.

Library	Percentage
`libaom`	94,43 %
`libvpx`	95,15 %
`libgsm`	97,53 %
`htslib`	45,38 %
`leptonica`	62,51 %

Tab. 3.5: Porcentaje de solapamiento entre estado del arte e in-vivo fuzzing.

En Tabla 3.5 se presenta el porcentaje de cobertura logrado por el estado del arte que se solapa con la cobertura lograda por nuestro prototipo. Allí podemos observar que el solapamiento es menor para los sujetos `htslib` y `leptonica`. Esto se lo atribuimos al hecho de que estas últimas librerías proveen una mayor cantidad de prestaciones y, por lo tanto, hay una mayor probabilidad de que dos aplicaciones (o fuzz drivers) distintas ejerciten distintas regiones de código.

Library	Bug Type	File
`htslib`	NULL ptr. deref.	`cram/cram_encode.c`
`htslib`	UAF	`md5.c`
`htslib`	Buffer overflow	`header.c`
`htslib`	Out-Of-Memory	`cram/cram_encode.c`
`htslib`	Out-Of-Memory	`cram/cram_encode.c`
`htslib`	Assertion violation	`cram/cram_codecs.c`
`htslib`	Assertion violation	`cram/cram_codecs.c`

Tab. 3.6: Errores encontrados durante la campaña de fuzzing.

En cuanto a la efectividad en términos de cantidad de crashes descubiertos, nuestra herramienta encontró siete crashes en `htslib`, los cuales se presentan en Tabla 3.6. Se trata de dos violaciones de aserciones, y cinco errores de manejo de memoria (dos *out-of-memory errors*, un *use-after-free*, un *heap buffer overflow* y una dereferenciación de un puntero nulo). Estos errores fueron encontrados únicamente por nuestra herramienta, a pesar de haber sido esta librería sometida a fuzzing utilizando los harnesses generados por FUDGE durante cuatro años[3]. Estos fueron los únicos crashes reportados en todas las campañas.

Con respecto a la tasa de falsos positivos, podemos destacar que todos los errores reportados por nuestra herramienta durante la experimentación pudieron ser reproducidos luego de terminada la campaña de fuzzing. Más aun, se los pudo reproducir utilizando una versión no-instrumentada del host y la librería, utilizando los archivos reportados por el fuzzer como entradas al host en cuestión.

[3] `https://github.com/google/oss-fuzz/commit/af319543`

3.3. Amplificación de casos de test

Una ventaja presentada por in-vivo fuzzing es que una vez especificado el target podemos utilizar más de una ejecución para amplificar. Es decir, en lugar de restringirnos a un único host, o una única ejecución del mismo, podemos variar el host y así focalizar distintos puntos de amplificación en distintos contextos. En particular, partiendo de la *test suite* del target (la cual no es otra cosa que un conjunto de binarios ejecutados en algún orden determinado), estamos efectivamente utilizando como semilla para la campaña de fuzzing a un conjunto de llamados a funciones que potencialmente incluye casos borde o patológicos. Adicionalmente, las test suites suelen estar diseñadas para ejercitar gran parte de la API de la librería, por lo que resulta esperable que las mismas ejerciten más puntos de amplificación que los alcanzados por ejecuciones puntuales de hosts arbitrarios.

Una test suite suele estar compuesta por varios binarios, cada uno de los cuales testea un subconjunto distinto de la funcionalidad provista por la librería. Al amplificar tal suite buscamos dedicarle la misma cantidad de tiempo de testing a todos los puntos de amplificación, dado que a priori no tenemos información sobre cuáles resultarán más fructíferos. El problema es que en una campaña de amplificación de casos de test ya no estamos tratando con un host, sino con múltiples hosts, los cuales pueden solaparse en cuanto a los puntos de amplificación que ejercitan. Esto resulta problemático a la hora de distribuir un presupuesto de tiempo entre los diferentes tests de la suite.

Algorithm 6 Test amplification

Input: Test suite S, Amplifier points F, Types T, Constraints C, Time t_0

```
1:  Map test2func = ∅
2:  Set funcs = ∅
3:  for Test s ∈ S do
4:      test2func[s] = get_exec_amplifiers(s, F)
5:      funcs = funcs ∪ test2func[s]
6:  end for
7:  executed = |funcs|
8:  fuzzed_funcs= ∅
9:  while not aborted do
10:     for s in S do
11:         unfuzzed = |test2func[s] − fuzzed_funcs|
12:         if unfuzzed > 0 then
13:             Time budget t1 = unfuzzed / executed
14:             fuzz(F, T, C, exec(s), t0, t1)
15:             fuzzed_funcs = fuzzed_funcs ∪ test2func[s]
16:         end if
17:     end for
18: end while
```

En Algorithm 6 se presenta el algoritmo que utilizamos para asignar porciones del presupuesto total de la campaña de fuzzing a los distintos tests que conforman la suite. El mismo toma como entrada un conjunto de casos de test S, los puntos de amplificación a testear F, la información de tipos T recabada durante la instrumentación, el conjunto de restricciones C y el presupuesto de tiempo t_0 a utilizar para la fase de exploración de cada campaña. En una primera iteración sobre la suite, identificamos cuáles puntos de ampli-

ficación son ejercitados por cada test, y mantenemos también un registro de la cantidad total de puntos de amplificación ejecutados a lo largo de la ejecución de la suite (líneas 3 a 7). Luego volvemos a iterar sobre todos los tests, identificando para cada uno cuáles puntos de amplificación que no hayan sido todavía testeados son ejecutados por el mismo. Si este ejecuta por lo menos un punto de amplificación nuevo, le asignamos un presupuesto proporcional a la cantidad de puntos de amplificación nuevos ejecutados por s y se ejecuta una campaña utilizando Algorithm 2. Luego de terminada la campaña, actualizamos el registro de cuáles puntos de amplificación ya fueron testeados. Es importante notar que los puntos de amplificación que ya fueron testeados con anterioridad (los comprendidos en *fuzzed_funcs*), son ignorados por la instrumentación para evitar testear repetidamente las mismas funciones.

A continuación analizamos la efectividad de la amplificación de suites de casos de test y su tasa de falsos positivos.

3.3.1. Setup experimental

Library	Type	#LOC	#Stars	Version
`openssl`	Cryptography	1M	22.8K	3.0.6
`libxml2`	Parsing	308K	483	2.10.3
`opus`	Speech compressor	80K	1.9K	1.3.1

(a) Información de los sujetos seleccionados.

Library	Test Suite Coverage	#Executed ampl. points
`openssl`	60 %	93
`libxml2`	61 %	14
`opus`	93 %	9

(b) Test suite information.

Tab. 3.7: Información relativa a las test suites de los sujetos seleccionados.

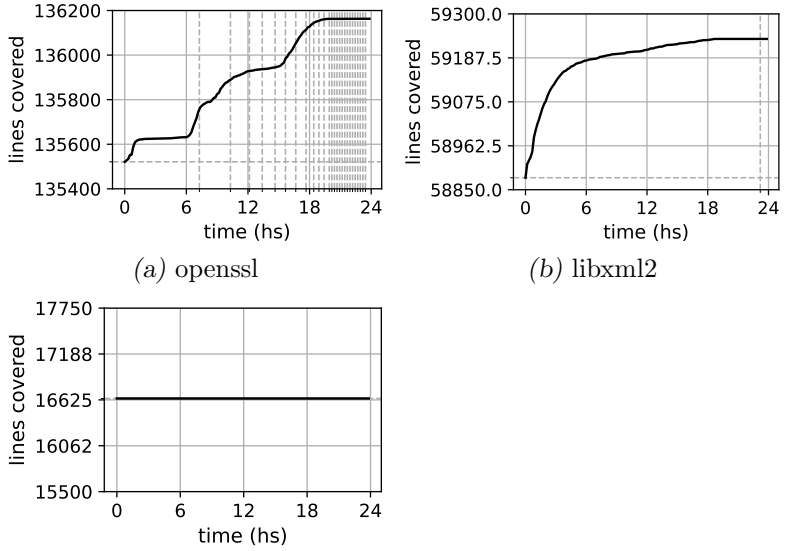
En Tabla 3.7a se presenta la información de las librerías seleccionadas. Tomamos librerías de diversos dominios, escritas en `C`, críticas en términos de seguridad y que hayan sido sometidas a fuzzing de manera continua por periodos extendidos de tiempo[4].

Dado que vamos a valernos de las test suites de las librerías para llevar a cabo las campañas de fuzzing, no resulta necesario seleccionar hosts ni tomar decisiones extra con respecto a las ejecuciones a amplificar. En Tabla 3.7b se presenta la información relevante a la test suite de cada sujeto. Nuevamente, en todos los casos identificamos puntos de amplificación utilizando nuestro script basado en `CodeQL` (Sección 2.7), y agregamos las restricciones de manera manual sobre la lista resultante.

Para cada uno de los tres sujetos de prueba, ejecutamos 20 campañas de amplificación de casos de test de 24 horas de duración, distribuyendo el tiempo de fuzzing entre los tests acorde a Algorithm 6. Nuevamente, la experimentación se llevo a cabo una sobre un procesador `AMD EPYC 7713P 64-Core` con `256GB` de `RAM` en containers de `Docker` para cada campaña.

[4] Los tres proyectos seleccionados fueron integrados a OSS-Fuzz en 2016

3.3.2. Resultados



(a) openssl *(b)* libxml2

(c) opus

Fig. 3.3: Cobertura de código vs tiempo para las campañas de amplificación de casos de test.

Library	Bug Type	ID	Severity
`openssl`	Buffer overflow	CVE-2022-3602	high
`openssl`	Buffer overflow	PR 19166	N/A
`openssl`	Use-after-free	CVE-2023-0215	moderate
`openssl`	Denial of Service	PR 19918	N/A

Tab. 3.8: Bugs encontrados durante la amplificación de casos de test.

En Figura 3.3 se presenta la cobertura en función del tiempo lograda para los tres sujetos seleccionadas. Las líneas punteadas verticales indican cuándo se iniciaron las distintas campañas, correspondientes a los diferentes hosts que componen la test suite (línea 14 en Algorithm 6). Las líneas punteadas horizontales indican el nivel de cobertura lograda por una ejecución no-amplificada de la test suite. Al medir cobertura no tomamos en consideración el código correspondiente a los casos de test, solo consideramos el código funcional dentro de la librería. En Tabla 3.8 se exhibe la información relevante a los bugs identificados durante las campañas.

Podemos observar que se obtuvo un incremento de cobertura de 600 líneas en `OpenSSL`, y más de 300 líneas para `libxml2`. En el caso de `opus` no pudimos lograr ningún incremento por sobre el nivel de cobertura alcanzada por la test suite. Al inspeccionar la cobertura obtenida por la test suite de `opus` sobre la librería, vemos que la misma logra una cobertura de código del 93 %, lo cual implica que la cobertura estaba casi saturada desde el comienzo. Por otra parte, esta test suite consistía de cinco casos de prueba, de los cuales uno ejercitaba los nueve puntos de amplificación, razón por la cual no se observa ningún cambio de host en Figura 3.3c. Algo similar ocurre para `libxml2`, donde el primer caso de test seleccionado dejó un solo punto de amplificación sin ejecutar, y solo ocurrió un cambio de host a lo largo de toda la campaña.

Al observar Figura 3.3a, vemos que durante la amplificación de cada host se alcanza una meseta local (como es esperable en este tipo de testing [11]), el cual es superado al cambiar de host durante la primera mitad de la campaña global. Esto destaca el hecho de que una vez provista una especificación de la librería, la misma se puede reutilizar durante la amplificación de distintos hosts, y que esto sirve para mejorar la cobertura de las campañas ejecutadas. Lo que es más, estas campañas de amplificación adicionales conllevan un costo muy bajo para el usuario, requiriendo únicamente una ejecución de un host alternativo. Esto no quita que hacia el final de la campaña de amplificación de casos de test se vuelva a llegar a una meseta de cobertura, inclusive luego de cambiar de host.

En términos de bugs encontrados, descubrimos 4 bugs en `OpenSSL`, dos de los cuales habían sido descubiertos mediante auditorías manuales del código. Entre los bugs encontrados está la vulnerabilidad relacionada con el procesamiento de `PunyCode` (CVE-2022-3602), referida en Sección 1.5. Se encontró también una vulnerabilidad de severidad moderada previamente desconocida resultante de un *use-after-free*, a la cual se le asignó el identificador CVE-2023-0215. No se reportaron falsos positivos durante ninguna de las campañas.

4. CONCLUSIONES

La técnica presentada en este trabajo permite aplicar fuzz testing guiado por cobertura sobre una librería dada, a partir de una aplicación host que la utilice. Esto nos permite testear al sujeto bajo condiciones que se asemejen a aquellas en las cuales sería utilizado en un entorno productivo. Este testeo no está circunscripto a las funciones que integren la API de la librería, sino que podemos hacer uso del contexto construido por la aplicación para testear regiones del código más profundas. Inclusive permitiéndonos testear partes del código que difícilmente sean consideradas por otras técnicas, como el fuzz testing basado en la utilización de harnesses manuscritos.

La amplificación de casos de test, siendo estos casos particulares del concepto de host, nos permite también testear *todos* los puntos de amplificación identificados por el usuario. Esto es destacable, ya que durante la preparación de los experimentos pudimos advertir que al tratar con hosts reales, los mismos ejecutan un pequeño subconjunto de las funcionalidades provistas por la librería. Si bien a lo largo del trabajo medimos la calidad del proceso de testing casi exclusivamente en base a la cobertura de código lograda, vale destacar que al utilizar esta técnica estamos efectivamente explorando una vecindad de los estados de ejecución de un programa. Esto implica que al amplificar una test suite, no solo estamos logrando una mayor cobertura de código, sino que también estamos utilizando como semilla para la campaña invocaciones a funciones que los desarrolladores deliberadamente consideraron importantes (ya sea por ser casos de uso típicos, o patológicos).

En los experimentos llevados a cabo demostramos que la amplificación es efectiva, pudiendo aumentar de manera significativa la cobertura de código por encima de la lograda por la ejecución base. En el caso de la amplificación de casos de test, pudimos someter a fuzz testing a grandes porciones de código de la librería, para las cuales escribir harnesses hubiera resultado altamente laborioso. Lo que es más, muchas de las funciones testeadas por nuestra herramienta no forman parte de la API de las respectivas librerías, y difícilmente hubieran sido tenidas en cuenta por desarrolladores a la hora de escribir harnesses. En cuanto a la capacidad de nuestra técnica para exponer bugs, podemos resaltar que encontramos siete crashes en un sujeto de prueba (`hstlib`), y tres crashes y un hang en otro (`OpenSSL`). Estos bugs no solo fueron expuestos por nuestra herramienta, sino que eludieron a otro fuzzers/harnesses, ya que los mismos no pudieron ser encontrados por fuzzers ejecutados de manera continua como parte del proyecto `OSS-Fuzz`.

4.1. Peligros de validez

Existen diversas consideraciones a tener en cuenta al interpretar los resultados presentados que pueden afectar la validez de los mismos.

Por un lado, la especificación de los puntos de amplificación y sus restricciones es una tarea semi-automática, y la misma tiene una alta injerencia en la tasa de falsos positivos reportados por la herramienta. En nuestros experimentos el proceso de identificar los puntos de amplificación fue asistido por la herramienta descripta en Sección 2.7, y el proceso de identificar las restricciones fue iterativo. Para cada sujeto, se planteó una primera aproximación de las restricciones, luego se ejecutó el host correspondiente indicándole a la instrumentación que serialice los argumentos iniciales (Sección 2.4). Si las restricciones

especificadas por el usuario no son lo suficientemente fuertes, lo más probable es que la serialización falle, lo cual permite reiterar sobre la lista de restricciones y ajustarlas hasta que la misma resulte adecuada. Esto fue necesario debido a que no teníamos familiaridad con los sujetos siendo testeados. Si esta técnica fuera implementada en la práctica por los desarrolladores de una librería, por ejemplo, los mismos contarían con el conocimiento suficiente como para correctamente identificar y restringir los puntos de amplificación sin necesidad de llevar a cabo este proceso iterativo.

Otro problema que surge con respecto a la reproducibilidad de los resultados es el no-determinismo inherente al proceso de fuzzing. Esto concierne no solo a los operadores de mutación y selección de semillas inherentes al fuzzing basado en mutaciones, sino que también tiene que ver con el comportamiento del SUT. Una premisa de CGF es que el programa siendo testeado es determinístico, razón por la cual se le atribuye únicamente al input la varianza en la cobertura lograda en diferentes ejecuciones. En la práctica, los sujetos considerados pueden comportarse de manera no-determinística, ya sea por la utilización de tiempos límite para terminar la ejecución del fork siendo amplificado (ver Sección 2.5), o por tratarse de programas con múltiples hilos de ejecución cuyo comportamiento depende del *scheduler* del sistema operativo. Para mitigar estos problemas es que ejecutamos cada campaña múltiples veces y promediamos los resultados. Adicionalmente, en todos los casos donde sea posible compilamos los hosts y las librerías sin soporte para multi-threading. A fines de lograr reproducibilidad, llevamos a cabo los experimentos dentro de containers que puedan ser reproducidos en diferentes entornos.

4.2. Trabajo futuro

Creemos que existen varias mejoras que se podrían hacer sobre lo presentado en este trabajo. La continuación que resulta más natural es la de inferir, dada la lista de puntos de amplificación, las restricciones pertinentes a los mismos. Esto eliminaría la mayor fuente de esfuerzo manual que todavía require esta técnica.

Por otro lado, en la implementación actual solo estamos amplificando la primera invocación a cada punto de amplificación. Pero en muchos casos existen múltiples llamados a un mismo punto de amplificación, y dado que cada uno se da en un contexto potencialmente diferente y con distintos argumentos, puede resultar beneficioso experimentar con distintas maneras de seleccionar cuál invocación se va a testear. Una dificultad asociada con esto sería el requisito de tener una política que nos permita seleccionar cuál de todas las invocaciones amplificar, y tener que serializar *todos* los argumentos provistos al punto de amplificación a lo largo de la ejecución a modo de semillas, lo cual impondría un gran costo en términos de espacio a la implementación.

Por último, existen targets que fueron excluidos de este trabajo debido a la complejidad asociada a las restricciones que debían respetar los argumentos de los puntos de amplificación. Tal fue el caso de varias librerías cuyo único punto de entrada recibía una estructura que codificaba la acción a desarrollar[1]. Lo mismo ocurrió con targets escritos en `C++`, ya que si bien los objetos en dicho lenguaje compilan a estructuras en `LLVM`, los invariantes que deben cumplir estas escapan a la expresividad del lenguaje implementado. Por estas razones, otra línea de trabajo futuro podría ser implementar soporte para restricciones más complejas, que permitan extender la aplicabilidad de esta técnica a targets

[1] A modo de ejemplo, ver: `https://android.googlesource.com/platform/external/libhevc/+/refs/heads/main/test/decoder/main.c#563`

más complejos, o escritos en otros lenguajes de programación.

Bibliografía

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[2] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 290–301, New York, NY, USA, 1994. Association for Computing Machinery.

[3] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: Fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 975–985, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[5] Marcel Böhme, Valentin Manès, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 14th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 970–981, 2020.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery.

[8] Richard A. Eyre-Todd. The detection of dangling references in c++ programs. *ACM Lett. Program. Lang. Syst.*, 2(1–4):127–134, mar 1993.

[9] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 337–350, New York, NY, USA, 2021. Association for Computing Machinery.

[10] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*, WOOT'20, USA, 2020. USENIX Association.

[11] Wentao Gao, Van-Thuan Pham, Dongge Liu, Oliver Chang, Toby Murray, and Benjamin I.P. Rubinstein. Beyond the coverage plateau: A comprehensive study of fuzz blockers (registered report). In *Proceedings of the 2nd International Fuzzing Workshop*, FUZZING 2023, page 47–55, New York, NY, USA, 2023. Association for Computing Machinery.

[12] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings 11*, pages 330–347. Springer, 2015.

[13] Github. Codeql. `https://codeql.github.com/`, 2021. Accessed: 2023-01-11.

[14] Harrison Green and Thanassis Avgerinos. Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1070–1081, New York, NY, USA, 2022. Association for Computing Machinery.

[15] David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. *SIGPLAN Not.*, 38(5):168–181, may 2003.

[16] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, Bellevue, WA, August 2012. USENIX Association.

[17] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. FuzzGen: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2271–2287. USENIX Association, August 2020.

[18] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

[19] LLVM. Libfuzzer. `https://llvm.org/docs/LibFuzzer.html`. Accessed: 2023-01-11.

[20] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, dec 1990.

[21] Rohan Padhye, Caroline Lemieux, and Koushik Sen. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 398–401, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.

[23] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 47(9):1980–1997, 2021.

[24] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: Network fuzzing with incremental snapshots. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 166–180, New York, NY, USA, 2022. Association for Computing Machinery.

[25] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, June 2012. USENIX Association.

[26] Kostya Serebryany. OSS-Fuzz - google's continuous fuzzing service for open source software. Vancouver, BC, August 2017. USENIX Association.

[27] Prashast Srivastava and Mathias Payer. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 244–256, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019.

[29] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press, 1981.

[30] Yichen Xie and Alex Aiken. Context- and path-sensitive memory leak detection. *SIGSOFT Softw. Eng. Notes*, 30(5):115–125, sep 2005.

[31] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. Intelligen: Automatic driver synthesis for fuzz testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 318–327, 2021.

[32] Mingrui Zhang, Chijin Zhou, Jianzhong Liu, Mingzhe Wang, Jie Liang, Juan Zhu, and Yu Jiang. Daisy: Effective fuzz driver synthesis with object usage sequence analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 87–98, 2023.