IKARO: plataforma de benchmarking de algoritmos de fuzzing para sistemas embebidos

Maialen Eceiza
Industrial Cybersecurity,
IKERLAN Technology Research Center,
Basque Research and Technology
Alliance (BRTA)
P. J. M. Arizmendiarrieta 2, E-20500
Arrasate-Mondragón, Spain
meceiza@ikerlan.es

Jose Luis Flores
Industrial Cybersecurity,
IKERLAN Technology Research Center,
Basque Research and Technology
Alliance (BRTA)
P. J. M. Arizmendiarrieta 2, E-20500
Arrasate-Mondragón, Spain
jlflores@ikerlan.es

Mikel Iturbe
Dpto. de Electrónica e Informática
Mondragon Unibertsitatea
Goiru 2, E-20500
Arrasate-Mondragón
miturbe@mondragon.edu

***Resumen*—La presencia de sistemas embebidos conectados aumenta progresivamente representando un desafío el poder actualizar vulnerabilidades una vez han sido desplegados en campo. Por ello, es necesario optimizar la búsqueda de vulnerabilidades antes de su despliegue. En este sentido, el Fuzzing es una de las técnicas que mejores resultados ha proporcionado para detectar vulnerabilidades de forma automatizada. Sin embargo, la propia naturaleza de los sistemas embebidos hace que existan una serie de condicionantes propios que dificultan la aplicación del Fuzzing en este ámbito. Los condicionantes principales son la ausencia de mecanismos de seguridad en hardware, o incluso la ausencia de un sistema operativo en algunos casos. A consecuencia de esto, una misma clase de errores presenta respuestas diferentes en diferentes tipos de sistemas embebidos. Ante esta problemática, se ha diseñado una plataforma llamada IKARO, que integra diferentes tipos de sistemas y que permite, por un lado, caracterizar los diferentes tipos de errores o vulnerabilidades de seguridad en los sistemas embebidos y por otro, una evaluación objetiva de los diferentes métodos de Fuzzing definidos en la literatura.**

***Index Terms*—Fuzzing, Sistema Embebido, IKARO**

I. Introducción

En los últimos años, el número de dispositivos embebidos conectados ha aumentado de forma muy considerable y además, lo ha hecho en términos de variedad y velocidad de despliegue. Tanto es así, que para 2025 se espera que en el mundo haya 75 mil millones de dispositivos IoT conectados [1]. Esta proliferación tan rápida de dispositivos (que viene motivado principalmente por el mercado) provoca que sea necesario disponer de ciclos de vida de desarrollo cada vez más cortos. Por ello, es más necesario disponer de tecnologías y técnicas más eficientes para el desarrollo, testeo y búsqueda de vulnerabilidades de estos sistemas. El impacto que puede tener el descubrimiento de una vulnerabilidad en un sistema embebido ampliamente desplegado es muy elevado al existir millones de dispositivos conectados en campo. La superficie de ataque que representa un fallo en una clase de dispositivo que está desplegado de forma tan masiva es enorme, como ya se vió en el caso del ataque Mirai [2].

Sin embargo, la superficie de ataque masiva no es el único gran inconveniente de este tipo de sistemas, ya que existen condicionantes que dificultan aún más asegurar este tipo de sistemas. El primer condicionante está asociado a la imposibilidad total o temporal de realizar actualizaciones del sistema por diferentes motivos, como son la ausencia de conectividad exterior, la falta de recursos para ello o bien la imposibilidad de realizar actualizaciones periódicas en el instante adecuado. Por ejemplo, en entornos industriales, los costes asociados a una parada de producción para instalar una actualización pueden no ser asumibles. El segundo está asociado con la criticidad del entorno donde operan ya sea el militar, el médico o aquellos entornos donde la seguridad física (*safety*) es importante. En estos ámbitos cada actualización debe ser planificada y verificada abriendo una ventana de exposición elevada hasta la actualización. Otro de los condicionantes está asociado a la variedad del hardware de los diferentes sistemas desplegados; la diversidad de los sistemas provoca que los ciclos de testeo sean mas complejos y largos, extendiendo la ventana de exposición ante ataques.

En estas condiciones, una de las etapas del ciclo de vida del desarrollo de productos que adquiere mayor importancia es la del testeo del sistema. En el ámbito del software tradicional no orientado a sistemas embebidos, esta etapa ha madurado y ha ido incorporando una serie de técnicas como el Fuzzing, que permite detectar de forma bastante rápida y eficiente bugs que pueden conllevar posibles vulnerabilidades de seguridad en unos tiempos de ejecución razonables.

El Fuzzing es una técnica que consiste en generar una serie de patrones de datos deliberadamente mal formados hacia un sistema a testear (*System Under Test*, SUT) y recoger las respuesta del mismo para bien valorar si se ha activado alguna vulnerabilidad o para mejorar la propia generación de patrones en sucesivos ciclos. El éxito de esta técnica con mas de 40 publicaciones, libros y herramientas publicadas en sistemas de proposito general [3] ha provocado una proliferación no sólo en el número o la variedad de los mecanismos de Fuzzing utilizados, sino de la clase de SUT que se desea testear. Arrastrado por este éxito, la aplicación del Fuzzing está extendiendose al campo de los sistemas embebidos con el objetivo de aprovechar las fortalezas que este tipo de técnicas dispone en el ámbito embebido.

El problema de la aplicación de este tipo técnicas en el campo de los sistemas embebidos es la propia naturaleza de éstos. Estos sistemas tienen una serie de características de hardware y software que difieren en diferentes grados de los sistemas de propósito general. Fundamentalmente, los

sistemas embebidos pueden tener carencias muy relevantes en los ámbitos de hardware y software. Desde el punto de vista del hardware, los sistemas embebidos no cuentan con mecanismos de protección como son una unidad de protección de memoria (*Memory Protection Unit*, MPU) o ni siquiera una unidad de gestión de memoria (*Memory Management Unit*, MMU). En consecuencia, la explotación de fallos de desbordamiento de buffer o de pila no generan excepciones y son, en el mejor caso, detectados posteriormente. Desde el punto de vista del software, los sistemas embebidos en los casos mas restringidos no cuentan con un sistema operativo, o bien cuentan con uno con capacidades limitadas.

En estas condiciones, es necesario evaluar la efectividad y la aplicabilidad de las diferentes técnicas de Fuzzing de una manera objetiva, esto es, es necesario evaluar cada técnica bajo diferentes condiciones de hardware y software para conocer la aplicabilidad de estas técnicas en el ámbito de los sistemas embebidos.

Con el objetivo principal de evaluar de forma efectiva y fiable las diferentes técnicas de Fuzzing se ha desarrollado IKARO, una plataforma hardware y software que agrupa diferentes categorías de sistemas embebidos donde es posible evaluar cada método bajo diferentes condiciones. Además, la plataforma puede ser utilizada para caracterizar el comportamiento de los sistemas embebidos ante diferentes clases de errores.

El objetivo principal de este artículo es presentar la plataforma esperimental IKARO. Para ello el artículo esta organizado siguiendo la siguiente estructura: en la sección II se describen los sistemas embebidos y se define la clasificación de sistemas embebidos que se utilizará, en la sección III se describe el funcionamiento y las propiedades del Fuzzing, en la sección IV se detallan las características y las funcionalidades de la plataforma esperimental que se ha diseñado y finalmente, en la sección V, se concluye el artículo.

II. Sistemas Embebidos

Los sistemas embebidos son soluciones de hardware y software que han sido diseñados para cumplir una tarea específica y que interactúan con su entorno mediante periféricos. Sus recursos están limitados a la realización de la tarea para la cual han sido diseñados. El ámbito de actuación determina una combinación muy particular de componentes de hardware y del software necesario para poder hacer uso de estos recursos. Como consecuencia, existe una proliferación muy amplia de combinaciones de hardware y software en el campo de los sistemas embebidos, en contraste con los sistemas de propósito general, donde existe una combinación muy estandarizada de componentes y software.

Esta diversidad de sistemas tiene un efecto inmediato en el comportamiento de éstos ante diferentes clases de errores. La ausencia de mecanismos de hardware que permitan realizar funciones de seguridad de apoyo al software (como realizan clásicamente la MPU o la MMU) provoca que fallos de desbordamiento de buffer o sobrepasamiento en las operaciones de coma flotante sean prácticamente indetectables. Es por tanto necesario, clasificar los sistemas para poder caracterizar los errores y evaluar de una forma objetiva las capacidades de los métodos de Fuzzing que encuentran estas vulnerabilidades. Además, hay que tener en cuenta que la propia limitación de los recursos de hardware es otro factor que determina el comportamiento de los sistemas ante diferentes situaciones de error.

II-A. Clasificación de los sistemas embebidos

Los sistemas embebidos se pueden clasificar utilizando distintos criterios como el área de aplicación o los requisitos funcionales entre otros. Desde el punto de vista del testeo de seguridad, el criterio más útil para la clasificación depende de las capacidades técnicas del hardware en seguridad como en procesamiento. De esta forma, es posible dividir los sistemas embebidos en tres grupos principales [4]:

- Sistemas de bajas prestaciones: estos sistemas disponen de un único núcleo y la memoria RAM tiene una capacidad inferior a 1 MB. Se caracterizan por no tener sistema operativo dada la limitación de recursos. A pesar de esto, son capaces de completar la tareas sencillas, sin la necesidad de muchos recursos. Esta clase de sistemas no suele disponer de mecanismos de gestión de memoria (MMU) aunque algunos de ellos sí pueden poseer excepcionalmente mecanismos básicos de protección (MPU). La ausencia de un sistema operativo provoca que estos mecanismos no se utilicen por el impacto en la memoria limitada que ya disponen y por la delegación en manos del equipo de desarrollo los controles necesarios. Como consecuencia es la categoría de sistemas que hace más complicada la tarea de detectar el instante preciso en el que se producen los errores. Este efecto si no es adecuadamente calibrado provoca un aumento de falsos positivos en los patrones de datos del Fuzzer, y por ende un falso negativo.
- Sistemas de medias prestaciones: son sistemas que disponen de uno o dos núcleos y memoria RAM de entre 1 MB, y 1 GB. Desde el punto de vista de seguridad en muchos de ellos se dispone de una MMU pero no de una MPU. Esta clase de sistemas sí suele disponer de sistema operativo, pero con capacidades limitadas y ajustadas para poder operar en el hardware en particular. La limitación de las capacidades no solo afecta a las funcionalidades estándar de un sistema operativo sino que también afecta a los servicios de seguridad que no siempre hacen uso del único mecanismo de gestión de memoria que disponen (MMU). A consecuencia de ello, a pesar de disponer de un sistema operativo (aunque limitado) y ser capaces de ejecutar más tareas no son capaces de proporcionar una reacción adecuada cuando se producen diferentes clases de errores.
- Sistemas de altas prestaciones: son los sistemas que más recursos tienen, disponen de varios núcleos y tienen la memoria RAM superior a 1 GB, y al igual que en el caso anterior disponen de MMU. Estos sistemas se caracterizan por tener un sistema operativo completo y son capaces de realizar tareas más complejas, incluso pueden realizar algunas tareas que realizan los dispositivos de propósito general. Además, el sistema operativo proporciona soporte que junto con los mecanismos de hardware permiten detectar más clases de errores y en el instante concreto en el que se producen.

Esta clasificación permite establecer una serie requisitos para el diseño de una plataforma que trate de reunir estas

categorías de sistemas materializadas en una serie de dispositivos en particular que se caracterizan de la forma que se han detallado. Ésto permite no solo evaluar, como se ha indicado previamente diferentes Fuzzers, sino también caracterizar el comportamiento de los errores en estas categorías de sistemas como ya señalan algunos autores como Muench y col. [4].

III. Fuzzing

El Fuzzing es una técnica muy eficaz para encontrar bugs en los sistemas que se propuso por primera vez en 1990 por Barton Miller y col. [5]. Su funcionamiento básico se basaba en generar un patrón de datos al azar, y enviar este al sistema testeado (*System Under Test*, SUT) para posteriormente analizar la respuesta recibida para determinar si el patrón de datos ha tenido éxito encontrando un bug.

Actualmente esta técnica ha ido mejorando en diferentes aspectos. En primer lugar, haciendo uso de toda la información posible que se puede extraer del SUT a través de diferentes técnicas como la instrumentación o el análisis estático. En segundo lugar, modificando el SUT para incorporar mecanismos de generación de información adicional, técnica conocida como instrumentación, o bien haciendo uso de técnicas de ejecución simbólica. En tercer lugar, aprovechando los recursos del sistema operativo y la instrumentación para marcar e identificar zonas de memoria específicas. Por último, se ha trabajado en el propio proceso de generación de datos incorporando diferentes mecanismos de mejora y/o aprendizaje en base a la información obtenida para generar patrones de datos mas precisos para explorar el código fuente en todas sus ramas.

Muchas de estas técnicas han sido mayoritariamente pensadas asumiendo que se ejecutan en sistemas de propósito general donde existen multitud de posibilidades para aplicar los diferentes mecanismos adicionales que permiten disponer de un Fuzzer con mayores capacidades de exploración. En cambio, en los sistemas embebidos existen no solo una serie de limitaciones de hardware sino que no es posible realizar instrumentaciones del código fuente o bien aprovechar los recursos del sistema operativo para identificar accesos a memoria u otros tipos de señales. Estas limitaciones junto con las restricciones del uso de mecanismos de hardware nos indica que es posible algunos de estos métodos no puedan ser directamente aplicables. Es necesario considerar que los Fuzzers trabajan en base a la respuesta del SUT, en consecuencia para un Fuzzer es más fácil trabajar con sistemas que proporcionan información sobre el error en el instante que han sucedido.

Esto implica que es necesario re-evaluar de una manera más objetiva y realista estos métodos para diferentes sistemas embebidos y determinar cuan aplicables y eficaces son bajo unas condiciones mas realistas y no asumiendo unas capacidades idílicas como las que disponen los sistemas de proposito general. Este resultado permitiría identificar que métodos son viables, y eficaces para su uso y bajo qué condiciones deberían operar.

IV. Plataforma experimental IKARO

El diseño de una plataforma de benchmarking debe perseguir unos objetivos claros y medibles, en este caso, el diseño de la plataforma, bautizada con el nombre de IKARO, persigue los siguientes objetivos principales:

- **Caracterización de los errores**. Consiste en analizar y estudiar todos los parámetros externos que se pueden recoger, para observar cómo responden a un conjunto de errores deliberadamente creados, en diferentes categorías de sistemas embebidos (de bajas, medias y altas prestaciones), los cuales se pueden dar en el desarrollo de productos. Este conjunto de errores está previamente clasificado y categorizado de forma que se conoce cual es el efecto que debe provocar y por tanto se puede conocer que efecto tienen en cada uno de los sistemas (Muench y col. [4]).
- **Evaluación objetiva de métodos de Fuzzing en sistemas embebidos**. Partiendo de una plataforma que disponga de sistemas embebidos de tres categorías, y con el conocimiento adquirido se pretende evaluar los diferentes métodos de Fuzzing en el ámbito embebido. De esta forma será posible clasificar los métodos de esta forma:
  - Métodos que son aptos sin ninguna clase de modificación.
  - Métodos que son aptos pero requieren alguna clase de modificación.
  - Métodos que no son aptos bajo ninguna circunstancia, porque no hay ninguna clase de modificación que permitan que puedan funcionar correctamente en los sistemas embebidos.

  Para todos aquellos métodos que si pueden funcionar con o sin modificaciones es posible realizar un ranking del rendimiento de los mismos y determinar cual es el mejor, pero también bajo qué condiciones de error. Además de estos, se determinará si el método es igual de eficiente en los sistemas de distinta categoría. La literatura muestra que algunos métodos son mejores que otros en la búsqueda de determinados tipos de error [4], por lo que se espera que sea necesario combinar diferentes tipos de métodos en función de la clase de error.

IV-A. Arquitectura de la plataforma IKARO

Para poder cumplir con estos objetivos se ha diseñado una plataforma de experimentación con una serie de características que trata reunir las características de las diferentes categorías de los sistemas embebidos que se han descrito en la sección II, esto es, de bajas prestaciones, de medias prestaciones y de altas prestaciones. Los elementos que se han elegido para cada una de las categorías son los siguientes:

- **STM32F429ZI** [6]: es un sistema embebido de STMicroelectronics que posee un único núcleo **Cortex M4** ARM de 32 bits, con una memoria **RAM de 256 KB**, una memoria persistente de tipo **FLASH de 2 MB** y que opera con un reloj de **168 MHz**. Por lo tanto, será un dispositivo de bajas prestaciones. Desde el punto de vista de seguridad no dispone de MMU, pero si dispone de una MPU, aunque su uso depende del entorno de desarrollo. Este sistema proporciona por tanto el entorno más adverso para la ejecución de Fuzzers pero es a la vez un entorno que nos puede proporcionar información muy útil sobre el comportamiento de los diferentes tipos de

errores en entornos tan reducidos, ya que se espera que sean difíciles de detectar y por tanto de caracterizar [7].

- **BeagleBone Black** [8]: es un sistema embebido de prestaciones medias de Texas Instruments, posee una CPU **ARM Cortex A8** de 32 bits, con una memoria **RAM de 512 MB**, una memoria persistente **FLASH de 2 GB** y que opera a **1 GHz**. Desde el punto de vista de los mecanismos hardware de seguridad dispone de una **MMU**, pero no de una MPU, ni ningún otro mecanismo de seguridad, pero con la ventaja de que es posible ejecutar un sistema operativo limitado sobre él [9]. Se espera que proporcione mas información que en el caso anterior ante la presencia de diferentes clases de errores, y donde un mayor conjunto de métodos de Fuzzing puedan operar con él.
- **Raspberry Pi 4 Modelo B** [10]: es un sistema embebido desarrollado por la fundación Raspberry Pi es de un sistema de altas prestaciones, posee una **ARM Cortex A72 Quadcore** de 64 bits de alto rendimiento, una memoria que puede variar entre 1 GB y 8 GB, siendo elegido de **4 GB de RAM**, una memoria persistente mediante tarjetas SD, y que opera a **1.5 GHz**. Al igual que en el caso anterior dispone de una MMU, pero no de una MPU como mecanismos de seguridad. Esta clase de sistemas se pueden a llegar a comportar como un sistema de propósito general en algunos aspectos, ya que se trata de una categoría de sistema con alta capacidad de cómputo. Por tanto, se espera que la mayoría de los métodos de Fuzzing funcionen.

Además de estas caracterśiticas se ha dotado a la plataforma de otra serie de características funcionales adicionales:

- **Interfaces de propósito general y específico**. Los sistemas de propósito general disponen de interfaces Ethernet, sobre los cuales opera con el stack TCP/IP. En cambio en los sistemas embebidos es habitual encontrar otro tipo de interfaces y protocolos como el CAN bus, donde se opera con unas tramas muy reducidas y con una complejidad inferior. La plataforma se ha diseñado proporcionando una interfaz Ethernet y también con una interfaz CAN, como exponente válido de las interfaces industriales de sistemas embebidos. La plataforma proporciona conectores físicos para todos los sistemas y en cada uno de ellos para las dos interfaces consideradas.
- **Sistema de control energético**. Como ya se ha indicado previamente el comportamiento de los sistemas embebidos puede llegar a ser muy diferente y uno de los aspectos donde se diferencian es cuando se producen determinados tipos de errores. Dependiendo del error, en los sistemas embebidos de altas prestaciones el sistema operativo finaliza la ejecución del proceso y se devuelve el control a la *shell*, en estos casos el sistema operativo del dispositivo puede ser capaz de detectar el error y activar una señal que lo indique. En el caso de los sistemas embebidos, y particularmente para aquellos de recursos muy restringidos, el sistema simplemente se queda en un estado donde la única salida es resetear el sistema, ya que no existe ninguna otra forma de recuperar el control. Para ello, se ha integrado un sistema embebido adicional de altas prestaciones que permite controlar el suministro energético de cada placa y por tanto posee la capacidad de encender, apagar o simplemente resetear el sistema. Esto ha supuesto la integración de una serie de relés de control en la plataforma. El núcleo de este sistema de monitorización es una placa Raspberry Pi 4, la cual tendrá el control de los sensores de energía y los relés. De esta forma, si se detecta un comportamiento anómalo en el consumo energético se activará el relé relacionado con el sistema que está teniendo ese comportamiento y se pordrá resetear el mismo. Controlar el consumo energético permitirá detectar un comportamiento errático en los casos en los que el sistema no es capaz de detectar el error. De esta forma, si uno de los sistemas se queda bloqueado se activará el relé correspondiente para que el sistema sea reseteado. Este sistema de monitorización se encuentra dentro de la plataforma IKARO y se comunicará por un lado con los elementos de control y por otro lado con el algoritmo de fuzzing, de esta forma el algoritmo recibirá los datos de monitorización. La detección en tiempo real de los errores será posible, ya que el cambio en el consumo energético presentará un patrón característico. En cambio, sin la monitorización el error se puede detectar tarde la detección del erros puede ser tardía (dada la falta de elementos de protección), ya que en algunos casos las señales de error de activan cuando termina la ejecución, o directamente el sistema puede continuar con la ejecución sin ser capaces de detectarlo.

El resultado de las premisas anteriores se ha materializado en un diseño por bloques como se muestra en la Figura 1, donde se pueden observar los diferentes subsistemas: el sistema de monitorización y control energético, los diferentes subsistemas a caracterizar, y el conjunto de métodos de Fuzzing a evaluar. La elección de los subsistemas que se van a caracterizar se ha realizado en base a la clasificación de los sistemas embebidos de la sección II, y considerando las interfaces de comunicación (Ethernet y CAN).

Finalmente, para facilitar el conocimiento del estado del sistema cuando está operando se ha dotado a su vez a IKARO de un sistema de visualización que permita conocer cual es el estado de la plataforma en un momento dado (ver Figura 1).

IV-B. Funcionamiento de la plataforma IKARO

La plataforma IKARO proporciona un banco de ensayos y experimentación básico para los dos objetivos mencionados previamente se han integrado y desarrollado una serie bibliotecas de código fuente de dos tipos:

- **Biblioteca de caracterización**. Para poder realizar este proceso, se han desarrollado muestras de código fuente que explotan no sólo con los fallos indicados por Muench [4], sino un conjunto más amplio de muestras de código que implementan más tipos de errores. La biblioteca estará formada por errores que se dan en binarios. El resultado de estas muestras es la creación de una biblioteca de programas cada uno con un tipo de fallo diferente para poder ser analizado en cada uno de los sistemas, y analizar las respuestas de cada uno de los sistemas a estos fallos. Estos códigos estarán implementados en lenguaje C y serán los códigos que se

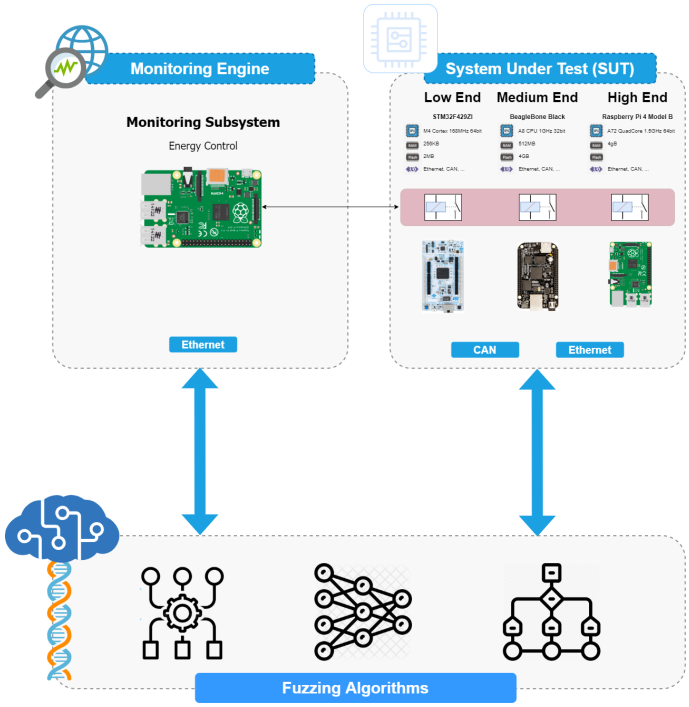

Figura 1. Arquitectura de IKARO.

introducirán en los distintos dispositivos para caracterizar cual es el comportamiento de cada uno de ellos ante los errores que hay dentro de esos códigos. La ejecución de esta biblioteca permitirá detectar y analizar cual es el comportamiento de cada uno de los sistemas ante cada error.

- **Biblioteca de métodos de Fuzzing**. Se trata de una biblioteca que contiene todos los métodos de Fuzzing publicados cuyo código fuente es públicamente accesible. Cada uno de los algoritmos de Fuzzing tienen sus propios requisitos y están escritos en un lenguaje diferente. A la hora de ejecutar los algoritmos es necesario tener en cuenta cuales son los requisitos y el lenguaje de cada uno de los métodos. Además, la ejecución de estos algoritmos se realizará en entornos de virtualización que cumpla con los requisitos que exigen los Fuzzers, pero cuyas capacidades han de ser las mismas en todas las ejecuciones para no influir en los resultados. La plataforma IKARO es completamente independiente del algoritmo, por lo tanto no dependerá del algoritmo, se utilizará un traductor de protocolos para adaptar las entradas generadas por los algoritmos a los dispositivos. De esta forma, si en un futuro se quiere probar en otros dispositivos únicamente habrá que cambiar el traductor.

A partir de la plataforma y este conjunto de bibliotecas es posible proceder al análisis pormenorizado de cada uno de los objetivos señalados y por tanto poder disponer una valoración real de la idoneidad de los diferentes métodos de Fuzzing en sistemas embebidos. Además, tanto la biblioteca de caracterización como la de los métodos de Fuzzing son independientes de los dispositivos. Por lo tanto, en un futuro existe la opción de cambiar alguno de los dispoditivos o añadir un dispositivo con otras características, ya que como la plataforma es modular cambiar cualquiera de los elementos de uno de los módulos no afectará a la relación de ese módulo con los demás.

V. Conclusiones

La necesidad de disponer de una plataforma de benchmarking específica es fundamental para poder determinar qué métodos de Fuzzing son adecuados y por ende poder integrarlos en los ciclos de vida de desarrollo de software para sistemas embebidos para minimizar el número de vulnerabilidades en el momento del despliegue. Del mismo modo, es importante la propia caracterización de todos los fallos posibles y poder conocer a priori cómo se van a comportar cada uno de los sistemas en diferentes situaciones.

IKARO ofrece una plataforma hardware sencilla donde no sólo se pueden realizar ambas categorías de ensayos, permitirá realizar más clases de ensayos futuros. De esta forma, en un futuro se pondrán añadir nuevos dispositivos a la plataforma con unas características diferentes, con el objetivo de ampliar los dispositivos que se caracterizan y comprueba el funcionamiento de los algoritmos de Fuzzing en ellos. Además, la plataforma dispone adicionalmente de una capa de software que se ha materializado en dos bibliotecas fundamentales: una con aplicaciones con diferentes tipos de fallos y otra con los métodos o algoritmos de fuzzing disponibles. El conjunto de ambas partes proporciona un ecosistema adecuado para la caracterización de fallos en sistemas embebidos y la evaluación de métodos de Fuzzing de una forma más objetiva.

Finalmente, con el objetivo de difundir en la medida de lo posible todos los desarrollos se publicarán en un futuro en modo abierto todas las bibliotecas utilizadas para que todos los investigadores y empresas interesadas puedan hacer uso de los recursos utilizados.

Agradecimientos

Maialen Eceiza recibe financiación mediante el programa Bikaintek (exp. núm. 20-AF-W2-2019-00006) del Departamento de Desarrollo Económico, Sostenibilidad y Medio Ambiente del Gobierno Vasco.

Mikel Iturbe es parte del grupo de Sistemas Inteligentes para Sistemas Industriales, financiado por el Departamento de Educación, Política Lingüística y Cultura del Gobierno Vasco.

Referencias

[1] Statista: "IoT: number of connected devices worldwide 2012-2025", en *recurso online* https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/, 2019, visitado el 05-10-2020.

[2] C. Kolias, G. Kambourakis, A. Stavrou, y J. Voas: "DDoS in the IoT: Mirai and other botnets", en *Computer*, 50(7), 80-84, 2017.

[3] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo y Wenqian Liu: ."A systematic review of fuzzing techniques."en *Computers & Security*, vol. 75, 02-2018.

[4] Marius Muench, Jan Stijohann, Frank Kargl, Aurelien Francillon y Davide Balzarotti: "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Device", en "2018 Network and Distributed System Security Symposium (NDSS 2018)".

[5] B. P. Miller, L. Fredriksen, y B. So, "An empirical study of the Reliability of UNIX Utilities en "Communications of the ACM, vol. 33, no. 12, pp. 32–44, 1990.

[6] STMicroelectronics:"STM32F0 Series."en *recurso online* https://www.st.com/en/microcontrollers-microprocessors/stm32f0-series.html, visitado el 19-10-2020.

[7] Majid Salehi, Danny Hughes y Bruno Crispo: "μSBS: Static Binary Sanization of Bare-Metal Embedded Devices for Fault Observability."en "23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)".

[8] Brian Benchoff:"BeagleBone Green, Now Wireless.[en] *recurso online* https://hackaday.com/2016/05/21/beaglebone-green-now-wireless/, 2016, visitado el 20-10-2020.
[9] Gerald Coley:"BeagleBone Rev A3 System Reference Manual", en *Beagleboard. org, Tech. Rep*, 2012.
[10] Raspberry Pi: Raspberry Pi 4 Computer, Model B.[en] *recurso online* https://www.raspberrypi.org , 05-2020, visitado el 08-10-2020.