Auditoría de Software Orientada a Compiladores

Caso de Estudio: Solidity

Matías Ariel Ré Medina
Dr. Ing. José María Massa

Una tésis presentada para el título de
Ingeniería en Sistemas

Ciencias Exactas
UNICEN
Argentina
Junio 2019

Auditoría de Software Orientada a Compiladores

Caso de Estudio: Solidity

Matías Ariel Ré Medina
Dr. Ing. José María Massa

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum tincidunt est et quam aliquam, sit amet fermentum dui tincidunt. Donec fringilla ipsum fermentum nibh ultrices mattis. Sed eu metus ultricies, malesuada diam pulvinar, lacinia velit. Sed maximus justo eget mi aliquam vestibulum. Mauris eget magna id risus sodales ullamcorper non sed orci. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Suspendisse egestas augue in volutpat tincidunt. Vivamus ac urna finibus, imperdiet enim quis, elementum odio. Phasellus vel pretium ipsum. Morbi non ipsum convallis, consectetur nisi vitae, tempus sem. Fusce sodales, augue vitae posuere dignissim, odio nibh volutpat enim, sed placerat risus elit quis mi. Ut pretium convallis tellus, at faucibus nulla. Aliquam porta neque in orci vehicula, eu ultrices nunc consequat. Suspendisse sodales diam sit amet nisi luctus, eget ullamcorper elit consectetur. Curabitur nec libero velit.

Praesent ligula mi, convallis eget turpis eu, tempor faucibus nunc. Praesent consequat tortor ac ligula dapibus ultricies. Duis id accumsan metus. Phasellus vel diam in velit dignissim vulputate. Integer sodales ullamcorper lorem eget suscipit. Phasellus gravida tempus facilisis. Sed mi sem, convallis posuere elit in, aliquam laoreet tellus. Etiam finibus feugiat purus, gravida condimentum sapien maximus a. Pellentesque ac risus turpis. Sed a nibh diam. Etiam viverra nisl auctor nulla efficitur fringilla. Vivamus vitae arcu a ante lobortis ullamcorper. Proin et lorem consectetur, aliquet nisi ac, tempus sem. Aenean ut massa id eros efficitur tincidunt.

Nam eu condimentum sem. Etiam iaculis et nunc a sollicitudin. Nulla vitae quam magna. Aliquam vestibulum, nibh ut accumsan malesuada, libero est facilisis ante, in aliquet mi velit quis nibh. Suspendisse pharetra sapien vitae massa sodales fringilla. Sed varius nulla eget lorem dignissim lacinia. Phasellus hendrerit imperdiet magna id hendrerit. Curabitur porttitor leo nec luctus tempus. Maecenas hendrerit, mauris ac auctor tempor, magna nulla accumsan mi, et dictum quam tellus a nunc. Etiam auctor dui ut lectus maximus, sed accumsan nunc facilisis.

Índice general

Capítulo 1

Introducción

1.1. Prólogo

Considerando que el mundo de la tecnología informática es un campo relativamente nuevo, que día a día crece exponencialmente, hay que destacar que dentro de él también se encuentran campos como el de la seguridad informática, que son mucho más recientes.

La explotación de vulnerabilidades existentes y nuevas permite el acceso no autorizado a los bienes de una empresa, siendo un problema de seguridad muy grave. *Una gran proporción de todos los incidentes de seguridad de software son causados por atacantes que explotan vulnerabilidades conocidas.*[65]

> *Romper algo es más fácil que diseñar algo que no se puede romper.*
>
> – Gary McGraw

Por eso es fundamental que se realice la comprobación de las aplicaciones, redes, sistemas nuevos y ya presentes, en búsqueda de vulnerabilidades para asegurarse que nadie sin acceso autorizado haya accedido previamente ni lo haga en el futuro.

Los análisis de seguridad comúnmente no llegan a cubrir el total de la infraestructura de una empresa. Hay dos principales razones por las cuales esto sucede: la inmensidad de las mismas y los plazos breves de tiempo disponibles para el trabajo[53][102][52][49]. No obstante, los mecanismos utilizados son efectivos, lo suficiente como para identificar vulnerabilidades conocidas, y comprobar cómo un atacante podría acceder a sus sistemas.

Las técnicas de testeo empleadas en el ciclo de vida del desarrollo seguro de un software se pueden distinguir en cuatro categorías: **(1)** *pruebas de seguridad basadas en modelos que se basan en los requisitos y los modelos de diseño creados durante la fase de análisis y diseño*, **(2)** *pruebas basadas en código y análisis estático en el código fuente y bytecode creado durante el desarrollo*, **(3)** *pruebas de penetración y análisis dinámico en sistemas en ejecución, ya sea en un entorno de prueba o producción*, así como **(4)** *pruebas de regresión de seguridad realizadas durante el mantenimiento*[65]. A pesar de que algunos mecanismos eran utilizados específicamente en el mundo de la seguridad informática, *dejando de lado la revisión de código por supuesto*, desarrolladores y DevOps están utilizando cada vez más estrategias como *fuzzing y análisis estático de código* para probar la calidad de su software[92][48].

1.2. Motivación

En las carreras universitarias las cuestiones de seguridad no se tratan con profundidad y de una manera enfocada al problema, sino desde los aspectos subyacentes que permiten entender los problemas de seguridad y sus posibles soluciones. Es por ello que los graduados que decidan dedicarse a la seguridad informática, deben especializarse por su cuenta a través de cursos, o mediante el aprendizaje profesional que se da a través de la resolución de problemáticas de los clientes.

Esta propuesta surge por un interés personal originado gracias a las materias *Lenguajes de Programación y Diseño de Compiladores*, y al incremento que ha habido últimamente en desarrollo de nuevos lenguajes, que poseen propósitos y contextos distintos[96][93].

1.2.1. Contexto

Los *Smart Contracts*, son programas que poseen una ejecución completamente verificable y observable. Esto permite que exista la certeza de que la ejecución del mismo no pueda ser alterada, abriendo una nueva posibilidad de casos de usos que en las plataformas de cómputo tradicionales no existían.

Ethereum Network fue desarrollada para ser una plataforma de Smart Contracts, siendo la primera que posee un lenguaje (del estilo bytecode) con característica *Turing complete* (permite que un el lenguaje pueda llegar a programarse para realizar cualquier tipo de operación) que corre en una máquina virtual llamada *Ethereum Virtual Machine (EVM).*

Si bien hay diversos lenguajes de programación que son compilados a la representación en bytecode para EVM, el que es oficialmente desarrollado y posee financiación por parte de la *Ethereum Foundation es Solidity*[103].

Solidity, que si bien se puede percibir como un lenguaje medianamente similar a Javascript en cuanto a sus aspectos sintácticos y en menor medida semánticos, nació de la necesidad de tener un lenguaje de alto nivel orientado a desarrollar *Smart Contracts* que permita interactuar con la Ethereum Network.

Los Smart Contracts hoy en día manipulan y almacenan caudales de dinero de gran magnitud, es por eso que es inevitable que la seguridad en estos casos se haga presente.

Ha habido ya muchos casos registrados de pérdidas de miles de millones de dólares, debido a descuidos a la hora de desarrollar y por no entender este muy reciente *"paradigma"*:

- Uno de los clientes más populares utilizado para facilitar a los usuarios la interacción con la red, congeló fondos valuados en $100 millones de dólares debido a un bug en su código[78].
- En julio del 2017, días después de que un hacker obtuviera más de 7 millones de dólares explotando una vulnerabilidad, debido a otro error en el mismo cliente, otro hacker obtuvo acceso a fondos de algunas cuentas, valuado en un total de $37 millones de dólares [86].
- Un ejemplo de un error de diseño del lenguaje con impacto a gran escala es el caso del famoso llamado, en este ambiente, "Reentrancy bug". Permitía a un atacante retirar una gran cantidad de veces su balance de un contrato, volviendo a llamar a la misma funcionalidad en medio de su ejecución, logrando así multiplicar sus fondos[69].

Así es como que desde el lado de los estudios de los lenguajes de programación, parece de suma relevancia poseer lenguajes y compiladores correspondientes que funcionen de manera esperada, sin permitirle a los desarrolladores la posibilidad de cometer errores catastróficos.

En este contexto un error en la generación del bytecode podría detener el funcionamiento de una red de miles de máquinas virtuales, o financieramente impactar de formas inesperadas en el contrato desarrollado.

1.3. Objetivos

Realizar una investigación de las estrategias y metodologías existentes para auditar compiladores, brindando primero una introducción a la auditoría de software, luego una introducción a auditoría específicamente de compiladores, comentando las técnicas más populares.

Finalmente evaluar como caso de uso una auditoría al lenguaje ***Solidity*** *y a su compilador* ***solc****, explicando el proceso y herramientas utilizadas, mostrando los resultados obtenidos.*

Capítulo 2

Marco teórico

En esta sección se presentarán los conceptos teóricos subyacentes en los que está basado este trabajo de investigación, que se utilizan a lo largo de este documento y son la base para comprender el marco en torno a él.

En la las primeras secciones, 2.1 a 2.6, se habla de las distintas estrategias, metodologías, y técnicas que envuelven el proceso de auditar un software. En las secciones 2.7 y 2.8 se introduce al compilador como un pilar importante en el proceso del desarrollo de software, y se explica su anatomía. Por último, en la secciones 2.9, 2.10 y 2.11, se presenta el concepto de una estructura blockchain, los smart contracts, y el lenguaje de interés Solidity, quién posee su propio compilador como parte del proceso de su utilización.

2.1. Auditoría de software (seguridad)

Uno de los *aclamados* padres de la seguridad informática del software, comenzó preguntándose por qué los creadores de Java fallaron al aplicarle seguridad al lenguaje en su momento. Buscando sobre cómo aprender sobre ello, se descubrió para su sorpresa que hasta ese momento ningún libro se había escrito al respecto, siendo que se sitúa a mediados de los 90'.

Luego de eso, en el 2000 publicó un libro llamado **Building Secure Software** (construyendo software seguro), junto a John Viega, que terminó siendo el *primer libro* en el mundo relacionado a seguridad en software.

Desde entonces se han publicado muchos otros libros[24][57][14][36][39][31][26][54], y se han aplicado estándares como **Building Security In Maturity Model**[62] (BSIMM, al estilo CMMI que ahora es el estándar de facto para medir iniciativas de seguridad de software), así como también el **Estándar de Verificación en Seguridad de Aplicaciones**[106] (ASVS de OWASP) y su versión abierta del **Software Assurance Maturity Model** (Modelo de Madurez de la Seguridad del Software) llamado **OpenSAMM**[107].

Sin embargo, la mayoría de los artículos que se pueden encontrar relacionados acerca de la seguridad en software comienzan describiendo políticas robustas previo al desarrollo del producto, y desde allí avanzan implementando mejores prácticas y programación segura. El gran problema de ello es que no todos los sistemas que quieran incorporar seguridad hoy en día pueden aplicar esta estrategia, ya que por cuestiones de recursos, o incluso sentido común, no se justificaría comenzar el desarrollo íntegramente desde cero. Es por eso que soluciones como estas sólo tienden a evitar las vulnerabilidades y no tratar las posibles ya existentes, para proyectos que ya estén creados.

Si bien ha habido un gran incremento en la aplicación de seguridad a la hora de desarrollar un software, no existen muchos materiales que traten la búsqueda de vulnerabilidades.

Vulnerabilidad de software versus bug
Entendiendo el término bug como un error, equivocación, o descuidos en programas que resultan en comportamientos inesperados y no deseados, una vulnerabilidad es un bug con implicancias en seguridad, permitiendo a un atacante abusar de ella para obtener alguna especie de beneficio, ya sea conseguir accesos privilegiados, provocar la caída de un servicio, tomar el control de un sistema o adquirir información sensible. Utilizar esa falla para violar las políticas de seguridad de un proyecto es lo que se llama explotar una vulnerabilidad, y quién intenta explotarla es llamado atacante.

2.2. Auditando

Auditar una aplicación es el proceso de analizar su código (en el formato de su fuente o binario) para descubrir vulnerabilidades que algunos atacantes podrían explotar. Realizando este proceso, se pueden identificar y cerrar agujeros de seguridad que de otra manera pondrían en un riesgo innecesario datos sensibles y recursos de negocios.

El contenido de la presente sección está desarrollado y profundizado en el libro *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*[21] (El arte del asesoramiento de seguridad de software: Identificando y previniendo vulnerabilidades de software).

A continuación ejemplos de algunas situaciones que son de relevancia en las que se realiza una auditoría:

Situación	**Descripción**	**Ventaja**
Auditoría in-house (pre-release).	Una compañía de software realiza auditorías de código a un nuevo producto antes de su release.	Fallas de diseño e implementación pueden ser identificadas y remediadas antes de que el producto salga al mercado, ahorrando dinero en desarrollo y desplegando actualizaciones. También le ahorra a la compañía una potencial situación vergonzosa.
Aditoría in-house (post-release).	Una compañía de software realiza auditorías de código a un nuevo producto luego de su release.	Vulnerabilidades de seguridad pueden ser encontradas y corregidas antes de que partes con intenciones maliciosas las descubran primero. Este proceso permite obtener tiempo para realizar testeos y otros chequeos, al contrario que estar haciendo una publicación en apuros en respuesta a un vulnerability disclosure (la acción de hacer pública una vulnerabilidad).
Comparación de productos por terceros.	Un tercero realiza auditorías a varios productos que compiten en un contexto dado.	Un tercero que pueda ser objetivo, puede proveer información útil a consumidores y asistirlos en seleccionar el producto más seguro.

Evaluación por terceros	Un tercero realiza una auditoría de software de manera independiente para un producto de un cliente.	El cliente puede obtener un gran entendimiento de la seguridad relativa a un producto que pretende desplegar. Esto prueba ser el factor decisivo entre la compra de un producto u otro.
Evaluación preliminar por terceros.	Un tercero realiza una auditoría de software de manera independiente de un producto antes de que salga al mercado.	Capitalistas de riesgo pueden obtener una idea de la viabilidad de una futura tecnología con propósitos de financiamiento. Proveedores pueden también realizar este tipo de evaluación para asegurar la calidad de un producto que pretenden llevar al mercado.
Investigación independiente.	Una compañía de seguridad o una firma de consultoría realiza una auditoría de software de manera independiente.	Proveedores de productos de seguridad pueden identificar vulnerabilidades e implementar medidas de protección en scanners y otros dispositivos de seguridad. La investigación independiente también funciona como una industria que está atenta y provee una manera para que los investigadores y compañías de seguridad establezcan credibilidad profesional.

2.2.1. Auditando versus black box testing

La idea del black box, o testeo de caja negra, es evaluar un sistema solamente manipulando sus interfaces, en el cual la estructura, el diseño o la implementación interna del objeto que se está analizando no es conocido por el profesional de validación (tester). Es llamado así, porque el software a ojos del tester es como una caja en la cual el contenido no se puede ver.

Desde la perspectiva de seguridad, y no tanto asociado a las definiciones formales, como la que podría proveer a los profesionales informáticos el **International Software Testing Qualifications Board** (Comité Internacional de Certificaciones de Pruebas de Software), lo que se busca es comunicarse con estas interfaces de maneras inesperadas para la lógica contenida en el software. Es decir, si se está testeando con esta metodología un servidor web (web server), se enviarán variantes de peticiones de protocolo **HTTP** (Hyper Text Transfer Protocol) reales, malformadas, e incluso modificadas para ser de un tamaño mayor al posiblemente esperado.

Cualquier comportamiento inesperado, o el mero detenimiento del software analizado es considerado algo de gran seriedad. El hecho de automatizar este proceso, para acelerar el hallazgo de más fallas, es llamado ***fuzz-testing***, del cual se hablará con profundidad más adelante.

A menudo, analizando código manualmente se pueden encontrar vulnerabilidades que realizando tareas automatizadas probablemente pasen por alto, y es por eso que existe la necesidad de realizar asesoramientos a las estructuras del código de manera inteligente, en adición a simplemente correr herramientas automatizadas observando sus resultados. Es necesario poder analizar código, y detectar caminos y/o vulnerabilidades que una herramienta puede llegar a pasar de largo.

Afortunadamente, la combinación de auditar manualmente y el uso de black box testing provee una combinación útil en cortos plazos de tiempo.

2.2.2. Auditoría de código y el ciclo de desarrollo de software

Cuando se consideran los riesgos de exponer una aplicación a potenciales usuarios maliciosos, el valor de un asesoramiento de seguridad está claro. Sin embargo, hay que saber exactamente cuándo realizarlo. Generalmente se puede realizar una auditoría en cualquier momento del ciclo vital del desarrollo/diseño de sistemas[11] (Systems Development Life Cycle, de ahora en más **SDLC**), pero el costo de identificar y corregir las vulnerabilidades encontradas no va a ser lo mismo para cualquier etapa de él.

Según la opinión del autor, los desarrolladores parecen pasar su vida entera encerrados en el SDLC, no importa si utilizan métodos ágiles, DevOPs, cascada, etc., siempre hay un análisis, una etapa de diseño, otra de implementación o código, y otra de testing (no siempre presente). Finalmente existe por lo general una etapa de operaciones, donde hay monitoreo.

Desde el punto de vista profesional, se observa una diferencia destacable entre lo que se conoce como un desarrollador puro y un ingeniero de sistemas. La diferencia reside en que los primeros, simplemente desarrollan piezas de software por lo general, sin seguir un proceso (o siguiéndolo sin necesidad de comprenderlo profundamente), mientras que los últimos están capacitados para diseñar y aplicar procesos, incluyendo también seguridad en ellos.

En relación de cómo la seguridad afecta al SDLC se puede decir lo siguiente[16][22][18][82]:

- **La seguridad es invasiva.**
  Si un desarrollador hoy en día no está de manera rigurosa involucrando la seguridad en su código, cuando ésta aparezca, que inevitablemente lo hará, lo sentirá invasivo, ya que deberán hacerse muchos cambios, y por naturaleza el humano es resistente al cambio (hay una gran rama de investigación bajo el acrónimo RTC que explica muy bien esta temática).

- **La seguridad no puede ser aplicada como un parche al final del proyecto.**
  Gary McGraw, previamente mencionado, y uno de los padres de la seguridad de la seguridad informática, comenzó siempre diciendo que la seguridad debe construirse en conjunto con el software, no agregarlo como algo después.

- **El sistema debe ser desarrollado con la seguridad activa por defecto.**
  El término **security by default** (seguro/seguridad por defecto) proviene de proveer un sistema o software, con la configuración más segura que se puede obtener por defecto. Esto no hace que el software sea impenetrable, sino que ahorra a los usuarios el tener que estar pendientes de una configuración para prevenirse, cuando puede que realizarla sea de una complejidad relevante.

- **El software debe ser escrito de manera segura desde el comienzo.**
  En el proceso hay que darle a la seguridad una alta prioridad, como aspectos claves del núcleo del proyecto, no como *features* (características adicionales) que van a ser agregados luego. Conviene atender estos aspectos desde el comienzo.

- **No subestimar la resistencia del equipo de desarrollo.**
  No subestimar la resistencia de los desarrolladores, ya que el hecho de tener que implementar seguridad en los sistemas hace que se incremente la complejidad y se retrase el tiempo para alcanzar la funcionalidad. El autor opina desde su experiencia que los desarrolladores en la industria están acostumbrados a que les paguen por cosas que funcionen.

- **Una relación cercana con el experto adecuado y la unidad de gestión desde el principio es un requerimiento que no debe ser negociado.**

Todo proceso de desarrollo de software sigue este modelo hasta cierto grado. El clásico modelo en cascada tiende a moverse hacia una interpretación estricta, en donde el tiempo de vida del sistema solamente itera una sola vez sobre el modelo. En contraste, nuevas metodologías, como desarrollo ágil[47], tienden a enfocarse en refinar la aplicación realizando repetidas iteraciones de las fases del SDLC. Entonces la forma en la que el modelo SDLC se aplica puede variar, pero los conceptos básicos y las fases son lo suficientemente consistentes para los propósitos de esta discusión.

Entendiendo que aplicar seguridad en distintas etapas del proceso puede llevar a hallazgos en cada una de ellas, es natural poder separar en categorías cada uno de estos hallazgos. Para ver una representación de cómo sería ver la Figura 2.1. Si lo que se encuentra corresponde a la etapa de

Figura 2.1: Security in the SDLC, Jim Manico, dotSecurity 2017.

diseño, tienden a ser fallas en las especificaciones y la arquitectura del sistema; vulnerabilidades de implementación son fallas técnicas a bajo nivel en la construcción real del software, y finalmente, en la categoría de vulnerabilidades operacionales entran las fallas que suceden por deployments y configuración del software en un entorno en particular.

Sin ir más lejos, los costos para solucionar un bug en algunas de estas etapas, tanto para proyectos ágiles o en cascada, empeoran mientras más tarde se encuentran.

A continuación se mencionan algunos hechos relevantes y destacados por su impacto, relacionados a bugs de fallas de seguridad. El Mariner 1 de la NASA fue el primer intento de Estados Unidos de enviar una nave espacial a Venus. Poco después de su lanzamiento en 1962, se desvió del rumbo debido a un error de software. Un empleado de la **NASA** se vio obligado a mandar a la nave a autodestruirse. Se quemaron $ 18 millones de dólares debido a un guión (-) faltante en el código[60].

En febrero del 2018 los servicios web de **Amazon** no estuvieron disponibles durante 4 horas y afectaron a innumerables sitios. Aunque el costo financiero no está claro, las estimaciones de cuando el sitio de Amazon se cayó en 2016 durante 20 minutos fue de $ 3.75 millones. El incidente de este verano fue 12 veces más largo e involucró a muchos otros sitios web.

AT&T actualizó su software para llamadas de larga distancia en enero de 1990. Sin embargo, no se percataron de que el sistema no podría mantenerse al día con la velocidad del nuevo programa. Las llamadas de larga distancia cayeron durante 9 horas. Se perdieron 200,000 reservas de aerolíneas y hubo 75 millones de llamadas telefónicas perdidas. El costo total estimado para AT&T fue de $ 60 millones [83].

El Instituto de Ciencias de Sistemas de **IBM** ha informado que *el costo de reparar un error después de la publicación del producto fue de cuatro a cinco veces más que uno descubierto durante el diseño, y hasta 100 veces más que uno identificado en la fase de mantenimiento.*

2.2.3. Costos en el tiempo

Phil Crosby, autor que contribuyó a las prácticas de la gestión de la calidad, en su libro **Quality is free** (La calidad es gratis) explica cuánto cuesta la mala calidad a largo plazo en un proyecto. Simplificadamente se puede interpretar en el gráfico de la Figura 2.2. El cual intenta demostrar que mientras más temprano, y en etapas más internas se puedan localizar los problemas, más

económico será el costo/tiempo de resolverlos, ya que se involucrará menos gente en el proceso.

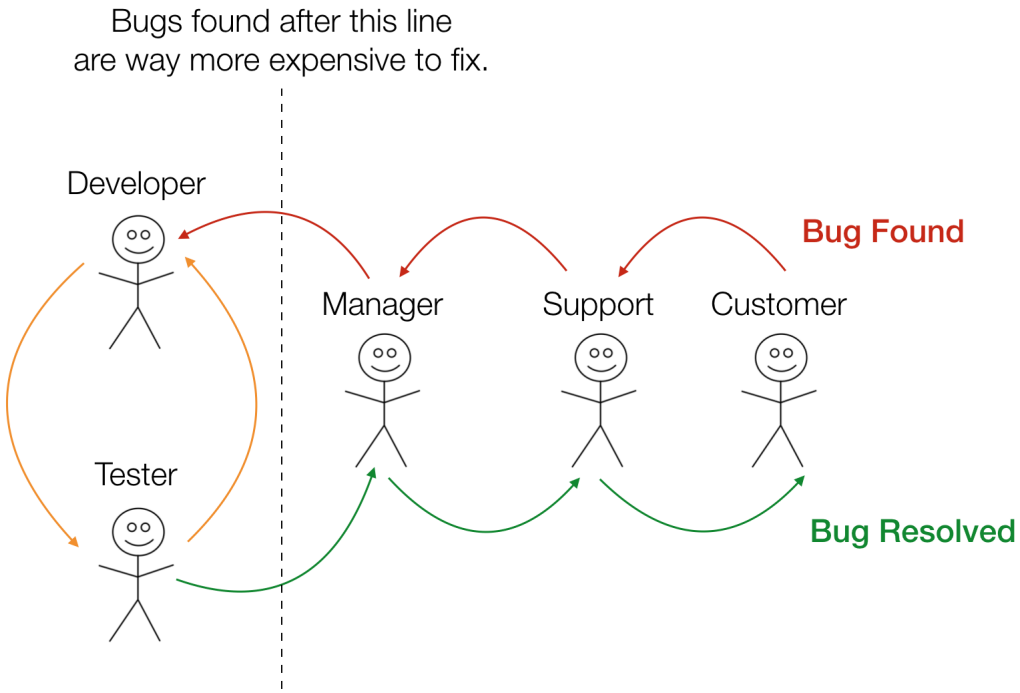

Figura 2.2: Cost of a software bug, Emanuil Slavov.

Para un ciclo de desarrollo en cascada, según Caper Jones, especialista en metodologías de ingeniería de software, y asociado con el modelo de punto de función de estimación de costos, en su libro Applied Software Measurement: Assuring Productivity and Quality[33] el costo para corregir un bug en las distintas etapas se ve descrito por el gráfico de la Figura 2.3

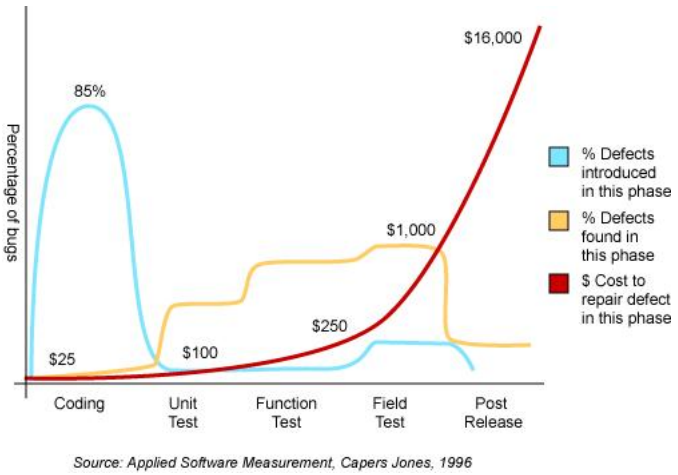

Figura 2.3: Applied Software Measurement, Caper Jones, 1996.

Steve McConnell autor de muchos libros de texto de ingeniería de software conocidos, como **Code Complete**[15], **Rapid Development**[6], y **Software Estimation**[23], en su primero también realiza un análisis en la dirección de Caper Jones. En la Figura 2.4, de uno de sus libros se puede observar el costo de detectar un defecto en determinadas etapas.

En la Figura 2.6 se encuentra una visualización de un artículo sumamente detallado, llamado The Agile Difference for SCM[17] (La diferencia ágil para la administración de la cadena de suministro), comparando los costos para los distintos modelos: XP, Boehm y Ágil.

Scott W. Ambler un ingeniero de software, consultor, autor de varios libros centrados en el kit

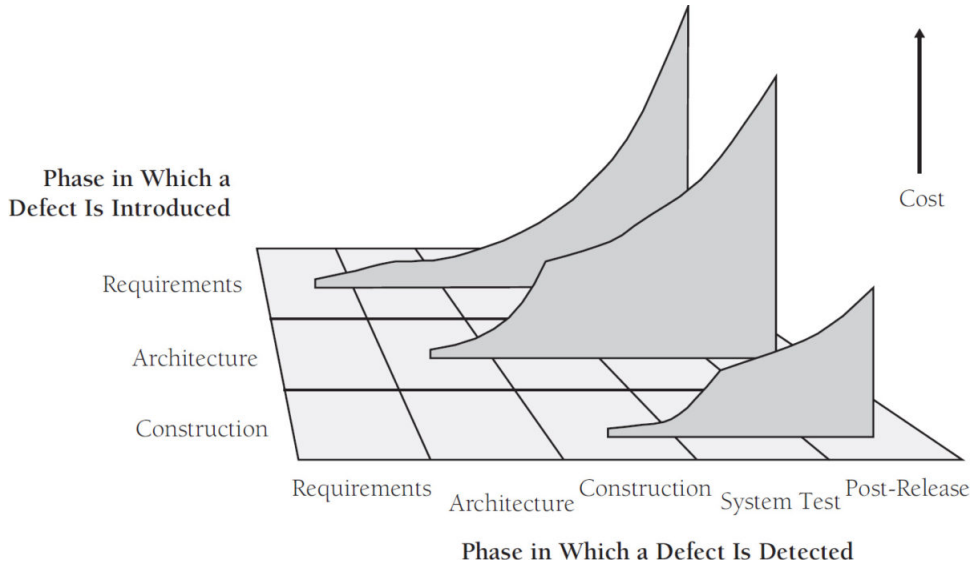

Figura 2.4: Costo de encontrar defectos por fase, Steve McConnell en Code Complete, 2004.

de herramientas de Disciplined Agile Delivery, el proceso Unificado, el desarrollo de software Agile, el Lenguaje de modelado unificado y el desarrollo del Modelo de madurez de capacidades. En su publicación **Why Agile Software Development Techniques Work: Improved Feedback**[20] (Por qué las técnicas ágiles de desarrollo de software funcionan: Feedback mejorado) se puede observar otro gráfico en relación a los costos con respecto a lo avanzado que se está en el desarrollo del proyecto:

A esta altura se puede asumir que el lector tiene una interpretación, al menos visual, respaldada de distintas fuentes con años de trayectoria de lo que cuesta posponer la búsqueda de problemas y solucionarlos en etapas tempranas.

Ahora, hay que entender que estos estudios no contemplan errores de seguridad, ya que tienden a ser superiores, porque si son vulnerabilidades explotables, tienen un impacto directo y dañino, en la empresa y sus consumidores; los impactos directos de reputación también no fueron cuantificados en los estudios anteriores.

Las violaciones a la seguridad cuestan al rededor 600 mil millones de dólares al año globalmente[98]. El 46 % de los incidentes de ciberseguridad del 2017 son debidos a personas internas a la organización[97]. El 30 % de los profesionales de seguridad esperan un gran y efectivo ataque dentro de los próximos 90 días.[101]

La mayoría de las fallas en los mecanismos de seguridad están relacionadas a funcionalidad faltante o incorrecta, y la mayoría de las vulnerabilidades están relacionadas a comportamientos adversos no intencionales.

Michael Felderer, et. al., explican muy bien en su sección de Security Testing[64] en Advances in Computers, 2016: Las pruebas (funcionales) normalmente se centran en la presencia de algún comportamiento correcto, pero no en la ausencia de un comportamiento adicional, que está implícitamente especificado por requisitos negativos. Las pruebas rutinariamente omiten las acciones ocultas y el resultado son comportamientos peligrosos de efectos secundarios que se envían con un software. La Figura 2.7 siguiente ilustra esta naturaleza de efectos secundarios de la mayoría de las vulnerabilidades de software que las pruebas de seguridad tienen que enfrentar[64].

El círculo representa la funcionalidad prevista de una aplicación, incluidos los mecanismos de seguridad, que generalmente se define mediante la especificación de requisitos. La forma amorfa superpuesta en el círculo representa la funcionalidad real e implementada de la aplicación. En un sistema ideal, la aplicación codificada se superpondría completamente con su especificación, pero en la práctica, este casi nunca es el caso. Las áreas del círculo que la aplicación codificada no cubre representan fallas funcionales típicas (es decir, comportamiento que se implementó incorrectamente y no se ajusta a la especificación), especialmente también en los mecanismos de seguridad. Las áreas que quedan fuera de la región circular representan una funcionalidad no intencionada y potencialmente peligrosa, donde residen la mayoría de las vulnerabilidades de seguridad. La falta de coincidencia entre la especificación y la implementación que se muestra en la figura que conduce

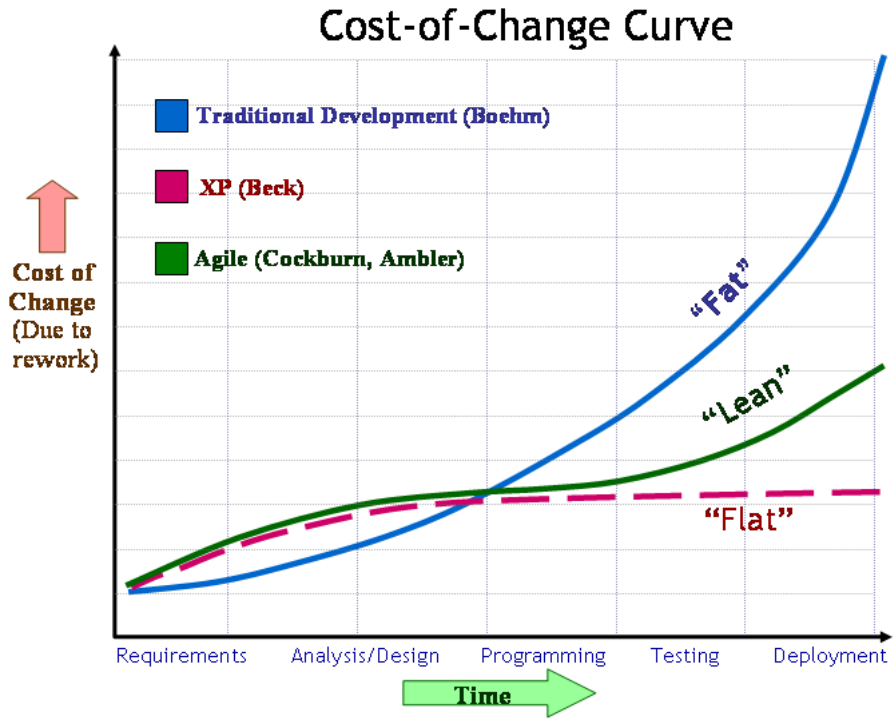

Figura 2.5: Curvas del costo de un cambio, CM Crossroads.

a fallas en los mecanismos de seguridad y vulnerabilidades se puede reducir teniendo en cuenta los aspectos de seguridad y especialmente las pruebas de seguridad en una etapa temprana y en todas las fases del ciclo de vida del desarrollo de software.

Dentro de la misma investigación, y basándose de manera abstracta a partir de técnicas de prueba de seguridad concretas mencionadas anteriormente, se clasifican en la siguiente figura de acuerdo con su base de prueba dentro del SSDLC.

Las pruebas de *seguridad basadas en modelos* se basan en requisitos y modelos de diseño creados durante la fase de análisis y diseño. Las *pruebas basadas en código y el análisis estático* se basan en el código fuente y el bytecode creado durante el desarrollo. Las *pruebas de penetración y el análisis dinámico* se basan en sistemas en ejecución, ya sea en un entorno de prueba o de producción. Finalmente, las *pruebas de regresión de seguridad* se realizan durante el mantenimiento.

2.3. Proceso de revisión de software

Cuando se tiene que comenzar con el proceso de revisar, desde la perspectiva de seguridad, un software que jamás se ha visto, teniendo una ventana de tiempo generalmente acotada, hay que entender cómo utilizar los recursos y poder cubrir bien las partes que son más relevantes a la seguridad.

Adoptando un proceso pragmático, flexible y basado en resultados se podrá obtener un balance para descubrir fallas de diseño, lógicas, operacionales y de implementación.

Hay dos frases que se destacan mucho de uno de los libros más importantes que he el autor ha tenido la oportunidad de leer en esta investigación, y son *"la revisión de código es un proceso fundamentalmente creativo"*, y *"hacer revisión de código es una habilidad"*.

Entendiendo esto entonces, la mejor manera de realizarlo es armando una lista de todo lo que probablemente pueda salir mal, y al hacer la revisión tratar de entender al desarrollador pensando en situaciones que no haya podido anticipar. Es una especie de estudio del perfil (profiling) del equipo de desarrollo, y entender que no sólo pueden tener fallas a nivel código sino también conceptuales que en el mismo proyecto pueden traer consecuencias a la forma de encarar otras partes del mismo proyecto, o futuros. Por ejemplo, puede que para un *input* (entrada) de usuario no se estén realizando los chequeos necesarios pero que no tenga ningún impacto real de seguridad; lo que no significa que en otros lugares del mismo proyecto esté aplicado de una forma diferente,

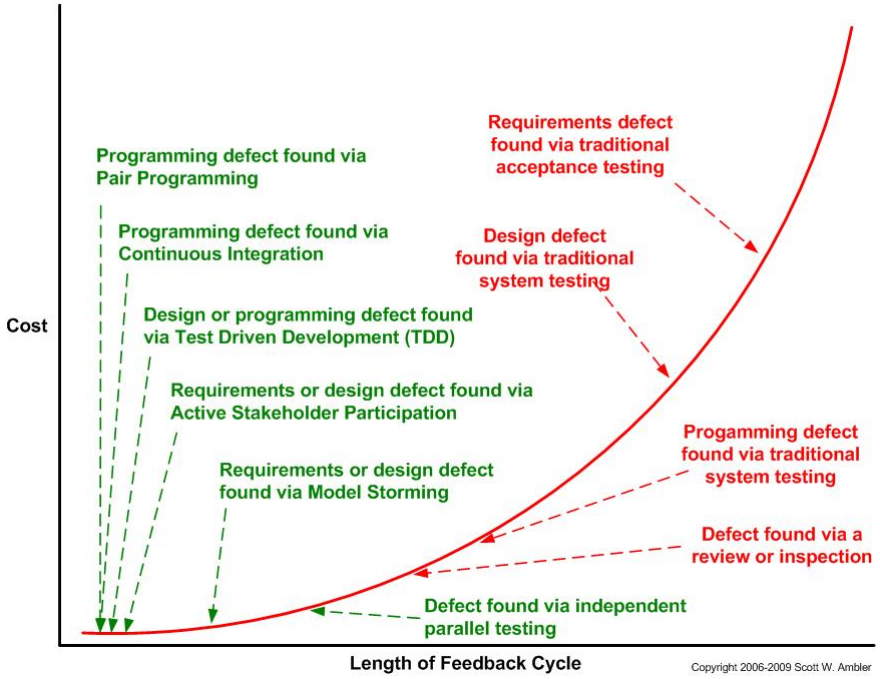

Figura 2.6: Longitud y costo de un defecto dentro del ciclo de desarrollo, Scott W. Ambler

pudiendo así explotar finalmente la falla, ya que ha sido conceptual o de diseño.

Tener un proceso es algo realmente valioso más allá de que hay más factores involucrados que simplemente seguir unos pasos. No todos los que realicen el mismo proceso obtendrán los mismos resultados, pero sí probablemente los haga más efectivos. Un proceso le da estructura a la auditoría, permite mantener un nivel de prioridades y ser consistente en el análisis. También de esta manera, junto con documentación permite que los asesoramientos sean algo compatible desde la perspectiva de negocios, lo cual no es algo menor, y es poco común.

El proceso de revisión que se mencionará a continuación, del cual hablan Mark Dowd, John McDonald, y Justin Schuhes, sobre la *identificación y prevención de vulnerabilidades de software*, es abierto, adaptable a los requerimientos de cada equipo para prácticas reales, y está dividido en 4 etapas:

- **Pre asesoramiento:** recolección de información inicial y documentación; planeamiento y alcance de la auditoría.
- **Revisión:** fase principal del asesoramiento. No está necesariamente estructurada en distintas etapas del ciclo de desarrollo. De hecho, estas etapas son objetivos simultáneos alcanzados por el uso de distintas estrategias.
- **Documentación y análisis:** documentación de procesos, hallazgos, y análisis sobre los mismos para evaluar riesgos y posibles métodos de solución.
- **Reporte y soporte de soluciones:** esta fase es básicamente para darle un seguimiento a quienes van a actuar en base a los hallazgos reportados.

2.3.1. Pre asesoramiento

En esta etapa se trata de obtener la mayor cantidad de información posible, porque a partir de ella se va a dictaminar cómo comenzar y qué acercamiento tomar frente a la auditoría.

Hay que definir bien el alcance del proyecto, tal vez sea buscar la vulnerabilidad más impactante del proyecto como también obtener la mayor cobertura del código posible sin la necesidad de comprobar que sean explotables o no las fallas encontradas. Esto no sólo va a depender de quién esté realizando la auditoría sino también del propósito de la misma.

Como forma tangible del software en cuestión se puede proveer tanto el código, como el binario, o ambos. La combinación del código y el binario, o un setup en el cual desde el código se pueda

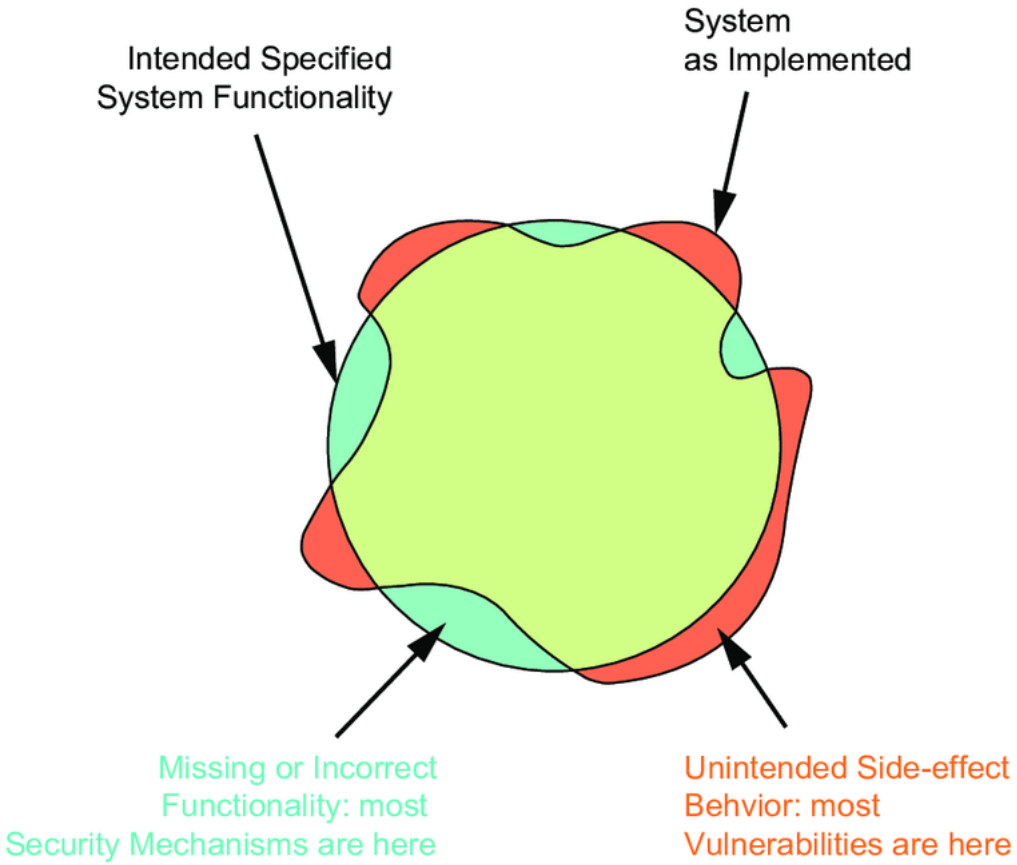

Figura 2.7: Security Testing: A Survey, Michael Felderer.

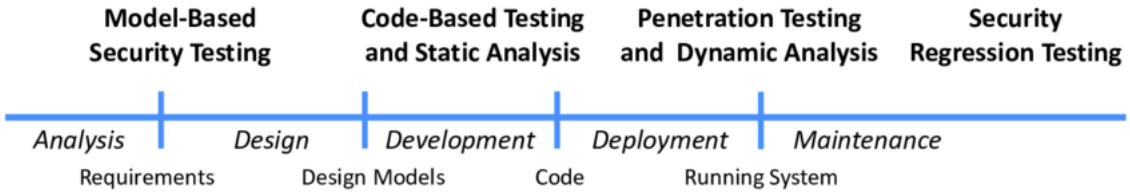

Figura 2.8: Técnicas de prueba de seguridad en el ciclo de vida de desarrollo de software seguro.

llegar al binario, son lo que hacen una revisión más eficiente (no siempre se puede llegar del código a un ejecutable). Sin un binario, ni la posibilidad de compilar, generalmente se realizan análisis estáticos, y en las potenciales fallas encontradas resultan difíciles de comprobar su real explotación. Sin el código fuente, no hay otra alternativa que realizar análisis en ejecución e ingeniería inversa (analizar el código assembler del ejecutable).

Existen otras alternativas, como poseer el binario con información adicional de debugging (símbolos por ejemplo) que facilita la ingeniería inversa, usualmente entregado por empresas con software propietario que quieren simplificar la revisión. Y el caso más extremo, usualmente más utilizado al realizar auditorías web, es cuando no se entrega código ni binario, en donde sólo se realizan técnicas externas como *black box* y *fuzzing* (*).

Fuzzing es una forma de descubrir errores en el software al proporcionar entradas aleatorias a programas para encontrar casos de prueba que causan un *crash* (detenimiento abrupto). Aplicando técnicas de *fuzzing* a programas, se puede obtener una vista rápida de la solidez general y ayudar a encontrar y corregir errores críticos. Es una técnica de *black box*, que no requiere acceso al código fuente, pero aún puede usarse contra el software para el que se posee dicho fuente, ya que posiblemente encuentre los errores más rápido, evitando la necesidad de revisar muchas líneas de código. Una vez que se detecta un bloqueo, si se tiene el código fuente, debería ser mucho más fácil de solucionar o explotar una vulnerabilidad.

En el caso de proyectos open source, como el que se verá en el caso de estudio, es donde se posee el mejor escenario. Para estos, siempre es mejor realizar una revisión manual de código, y como tal se debe comenzar con un *threat model* (modelado de amenazas) o al menos entrevistas a los desarrolladores para tener un entendimiento de la arquitectura de la aplicación, su superficie de ataque, así como también las técnicas de implementación.

2.3.2. Revisión

La tendencia principal a la hora de comenzar es generalmente seguir un modelo en cascada, lo cual no siempre es el mejor camino o el único, como ya se ha hablado anteriormente, más si se está hablando en términos de ser tiempo-eficiente identificando vulnerabilidades en alto como bajo nivel. Parecerá obvio pero un auditor se encuentra más preparado para juzgar la seguridad del diseño o implementación, al finalizar la auditoría que al principio. Esto es, hablando en términos de entenderlo de manera abstracta.

No siempre se puede comenzar por el diseño o haciendo un *threat model*, a veces la documentación no existe, está incompleta, vieja; los desarrolladores pueden también no estar disponibles.

El método para realizar la revisión es un simple proceso iterativo. Particularmente se planea qué estrategia utilizar, y dependiendo de ella la selección de técnicas u objetivos. Después se ejecuta la estrategia seleccionada tomando notas, y de vez en cuando reflexionar sobre el manejo del tiempo, para no obsesionarse particularmente con algo que demande demasiado. Finalmente entender qué es lo que se ha aprendido y ver cómo utilizarlo, repitiendo estos pasos hasta el final de la auditoría.

Hay tres maneras generalizadas para realizar la evaluación: *top-down*, *bottom-up*, e híbrida; las primeras dos son análogas a los tipos de descomposición de componentes en el diseño de software, y la última una combinación de ambas, alternando, dependiendo cuanta información se tenga del contexto dado.

En la etapa de planeamiento se elige qué tipo de estrategia se utilizará, en las próximas secciones se detalla brevemente las recomendadas por estos autores. La selección, la preparación para la misma y demás detalles se realizan dependiendo si se trabaja en equipo, con metas/objetivos específicos, y preparación de la toma de notas/documentación que se va a acontecer. También se pone sobre la mesa hacer un chequeo general del estado de la evaluación, re-evaluar si la estrategia seleccionada está funcionando, y conducir *peer-reviews* para tener más objetividad.

2.3.3. Documentación y análisis

En esta etapa, luego de que el trabajo difícil ha terminado, se documentan o formalizan las notas que se tomaron con los hallazgos.

Principalmente se puede describir el tipo de amenaza, el componente afectado, el módulo (línea de código) donde se encontró, a qué clase de vulnerabilidad pertenece, una descripción con el resultado y el contexto para reproducirla. Finalmente el riesgo o impacto que produce.

Se decidió no entrar en detalles en cómo es cada una de las categorías ya que son bastante explicativas por sí solas, y depende de la dedicación que se le quiera abocar a cada una, tanto como agregar mas o utilizar menos.

2.3.4. Reporte y soporte de soluciones

Finalizar una auditoría no consta solamente de entregar lo hallado de manera organizada. Se cumple un rol muy importante a la hora de entregar, ya que sin alternativas, soluciones o sin mantener un contacto para ayudar a remediar los problemas reportados nos estaría faltando una parte muy importante de nuestro trabajo.

Dependiendo del tipo de auditoría que se esté realizando, esta etapa puede implicar situaciones más complejas. En caso de una investigación independiente, en la circunstancia de haber encontrado vulnerabilidades explotables que pueden perjudicar a otros usuarios, se debe iniciar un disclosure (divulgación) responsable. Generalmente se desarrollan exploits (programa particularmente diseñado y/o utilizado para abusar vulnerabilidades en un determinado sistema) antes de contactar al proveedor de ese software/servicio, y una vez contactado, en el caso de respuesta, se otorgan 30-90 días para solucionar el problema antes de hacerlo público.

Recorrer el código Al autor le parece importante dejar expresado brevemente la importancia que tiene saber recorrer un código, porque es lo que se va a estar haciendo la mayor parte del tiempo, y dependiendo de la manera en la que se haga va a condicionar la velocidad y dificultad de su progreso.
Generalmente, lo que es más efectivo es revisar funciones de manera aislada, y hacer seguimiento al flujo sólo cuando es absolutamente necesario.

2.4. Estrategias de auditoría de código

Las estrategias se pueden resumir a las siguientes tres categorías:

2.4.1. Compresión de código (CC):

Estas estrategias analizan el código directamente para descubrir vulnerabilidades y mejorar el entendimiento de la aplicación.

En esta categoría se encuentran las siguientes metodologías: seguimiento de *inputs* maliciosos, analizar un módulo, analizar un algoritmo, analizar una clase u objeto y seguimiento de cosas de interés por usar *black box*.

Exceptuando por el seguimiento a los resultados encontrados realizando *black box*, todas ellas son difíciles y lentas, pero todas las comprensiones de los impactos hallados serán altísimas.

2.4.2. Puntos candidatos (CP):

Se requieren dos pasos: Primero crear una lista de problemas potenciales mediante un proceso o mecanismo y luego examinar el código en busca de ellos.

En la lista mencionada se encuentran las siguientes estrategias: herramientas automatizadas de análisis del código; puntos candidatos de enfoque general, léxicos simples, binarios simples, generados por *black box*, y específicos de aplicación. Estas estrategias se destacan por ser rápidas, fáciles de realizar pero la comprensión de su impacto suele ser bajo.

2.4.3. Generalización de diseño (DG):

Estas técnicas, son más flexibles, intencionadas para analizar potenciales problemas de medianos a alto nivel en la lógica y el diseño.

En ella se encuentran las siguientes sub-estrategias: modelado del sistema, testeo de hipótesis, deduciendo propósito y funcionalidad, y chequeo de conformidad de diseño.

Son de nivel moderado a difícil, velocidad media, pero con altísima comprensión del impacto.

2.5. Tácticas de auditoría de código

En esta sección se presenta un análisis de las tácticas más relevantes de auditoría de código, este análisis está basado mayormente en el capítulo *6.4.9* del libro *The Art of Vulnerability Assessment* previamente mencionado.

2.5.1. Análisis de flujo interno

Muchos caminos poseen secciones similares de código, por lo tanto analizar los que sean relevantes no es tanto trabajo como parece, además de que es totalmente posible leer varios caminos en simultáneo.

Se pueden ignorar las fallas de chequeo de errores ya que no son relevantes a la seguridad, pero hay que tener extremo cuidado cuando se descartan partes con este criterio, ya que es bastante común descuidarse en dos áreas en particular: ramas de chequeos de errores, las rutas que el código sigue cuando los chequeos de validación resultan en error; y rutas de código patológicas, funciones con pequeñas rutas que no resultan con una terminación abrupta de su funcionalidad (*crash*).

2.5.2. Análisis de subsistema y dependencias

No sólo es necesario revisar los módulos que interactúan directamente con los datos ingresados por los usuarios, también es importante entender los subsistemas y las dependencias que estos mismos utilizan. Algunos ejemplos son las maneras que utilizan para hacer manejos de memoria, APIs de sistema, subsistemas que manejan guardado de datos, parsers de cadenas de caracteres, manejos de buffers de datos, etcétera.

2.5.3. Releer código

Releyendo el código es una de las únicas maneras para lograr terminar de entenderlo, si se quiere se puede interpretar como un proceso iterativo, en donde las primeras pasadas pueden utilizarse para concentrarse en vulnerabilidades más evidentes.

2.5.4. Chequeando con lápiz y papel

Si hay partes del código a las cuales se les puede hacer un seguimiento con diversos tipos de datos de entrada, se puede trabajar sobre lápiz y papel como se enseña en la facultad, para ver si responde acorde. A muchos les sorprendería los resultados que se obtienen con este simple, pero efectivo mecanismo.

2.5.5. Casos de test

Realizar casos de test es algo muy útil, y no solo se pueden realizar llamando al binario desde otro programa (wrapper) con diversas entradas, sino que se pueden realizar casos de test como haría uno mismo si estuviera desarrollando el software. También, en la experiencia del autor, ha partido de casos de test provistos por el proyecto y se ha pivotado hacia una perspectiva más de seguridad para encontrar cosas relevantes.

2.6. Herramientas para la auditoría

Existen una gran cantidad de herramientas para facilitar el proceso. Herramientas como navegadores de código[129][257][260][258], analizadores de código estático[123][140][141][130][163][117], analizadores del software en ejecución/debuggers [241][155][207][266][212], navegadores de binarios/disassemblers[119][122][161][118][156], fuzzers[114][240][157][158][116], y más.

El autor no entrará en detalle, ya que describir todos carece de sentido, y que no todos sirven para todos los contextos, además hablará con detalle más adelante respecto a los que fueron utilizados en el caso de estudio.

2.7. Auditoría de compiladores

El foco del caso de estudio está dado en un software que su vez es un compilador.

2.7.1. Arquitectura

Los compiladores cumplen un rol muy importante en el desarrollo de software. Su principal función es convertir un código fuente en un lenguaje adecuado para su manipulación por humanos en un código ejecutable, cercano a una computadora real o virtual, correspondiente a una o varias arquitecturas.

Los compiladores se encuentran organizados por etapas sucesivas, apreciadas en la Figura 2.9 a continuación, y entre cada una de ellas se manipulan datos de naturalezas específicas. En general, los compiladores pueden verse como traductores, donde cada traducción requiere procesos de análisis y síntesis.

El desarrollo de los compiladores sigue un esquema conocido como “Teoría de Compiladores” el cual tiene su raíz junto con el nacimiento de la computación misma y más precisamente con los primeros lenguajes de programación de alto nivel. Esta teoría ha sido consolidada por varios autores[3][12][5][3][27][13][10][8][44][2][1][37][35][28][40][46][50][12][41][84] y con el surgimiento de nuevos lenguajes de programación[38][61][95][105][108][58][51][94][99][89][267] contínuamente está siendo

Figura 2.9: Arquitectura. Ray Toal, Loyola Marymount University.

puesta a prueba sin que esto haya significado modificaciones esenciales en el esquema. El caso del lenguaje Rust, de la fundación Mozilla[243] cuyo compilador se encuentra documentado[242] en los términos de la Teoría de Compiladores, es una prueba fehaciente que las nuevas construcciones sintácticas y semánticas sobre seguridad de memoria, closures y paralelización pueden ser soportadas sin problemas por dicha teoría.

Ambas etapas están compuestas de fases internas que serán descritas en las siguientes secciones.

2.7.1.1. Componentes

A continuación brevemente se describen los componentes para un compilador que genera lenguaje ensamblador. En casos en los que los compiladores están diseñados para lenguajes de más alto nivel, o hacia máquinas virtuales, algunas de las etapas descritas son obviadas.

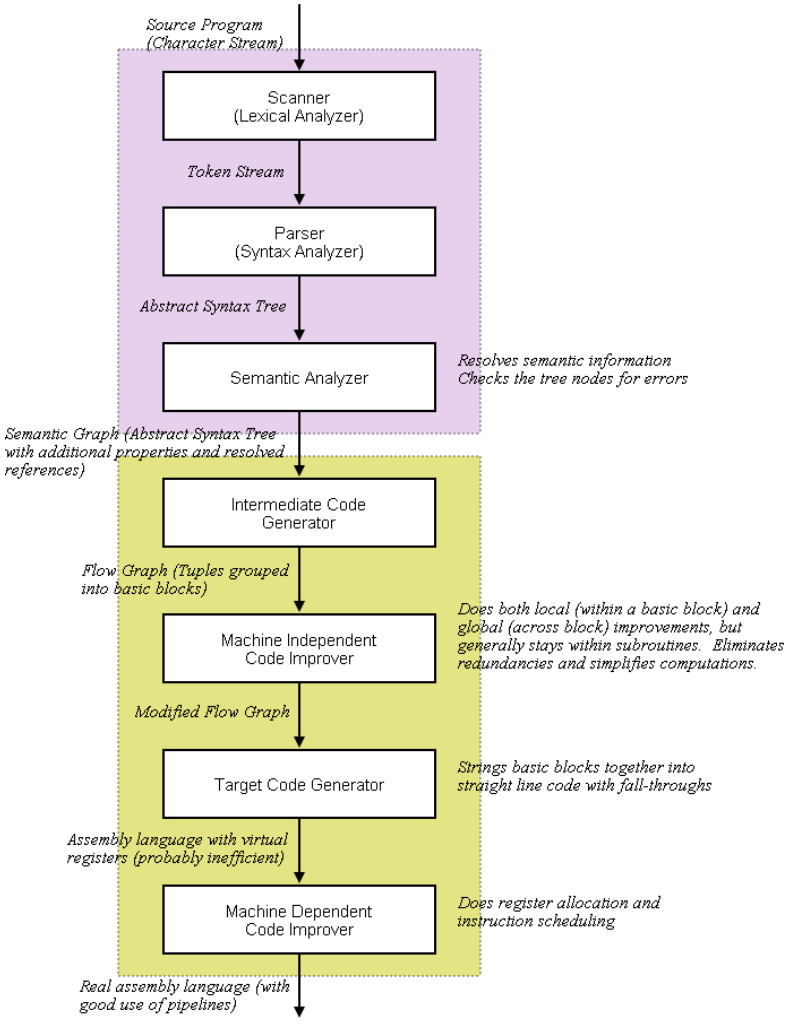

Figura 2.10: Componentes de un compilador. Ray Toal, Loyola Marymount University.

2.7.1.2. Análisis léxico (Scanner)

El scanner convierte el flujo de caracteres correspondiente al código fuente del programa en un flujo de tokens. Desde el código del recuadro 2.1 se puede llegar al conjunto de tokens mostrado en la Figura 2.11.

```
1  #define ZERO 0
2  unsigned  gcd (
3      unsigned int x,// Algoritmo Euclideo
4          unsigned y) {
5              while ( /* hello */ x> ZERO) {
6                  unsigned temp=x;
7                  x=y%x;y  = temp;
8              }
9          return y;
10 }
```

Listing 2.1: Código de ejemplo

`unsigned` `ID(gcd)` `(` `unsigned` `int` `ID(x)` `,` `unsigned` `ID(y)` `)` `{` `while` `(` `ID(x)` `>` `INTLIT(0)` `)` `{` `unsigned` `ID(temp)` `=` `ID(x)` `;` `ID(x)` `=` `ID(y)` `%` `ID(x)` `;` `ID(y)` `=` `ID(temp)` `;` `}` `return` `ID(y)` `;` `}`

Figura 2.11: Conjunto de tokens. Ray Toal, Loyola Marymount University.

Los escáneres se ocupan de cuestiones tales como:

- Sensibilidad de mayúsculas y minúsculas (o insensibilidad)
- Si los espacios en blanco son significativos o no
- Si las nuevas líneas son significativas
- Si los comentarios pueden anidarse

Los errores que pueden ocurrir durante el escaneo, llamados errores léxicos, incluyen:

- Encontrar caracteres que no están en el alfabeto de idioma
- Demasiados caracteres en una palabra o línea
- Un caracter no cerrado o cadena literal
- Un final de archivo dentro de un comentario

2.7.1.3. Análisis sintáctico (Parsing)

El árbol también puede ser guardado en forma de un string (cadena de caracteres) como es mostrado en el recuadro **??**

```
1 (fundecl unsigned gcd
2   (params (param unsigned x) (param unsigned y))
3   (block
4     (while
5       (> x 0)
6       (block (vardecl unsigned temp y) (= x (% y x)) (= y temp)))
7     (return y)))
```

Listing 2.2: AST

Técnicamente, cada nodo en el AST se almacena como un objeto con campos con nombre, muchos de cuyos valores son nodos en el árbol. Hay que tener en cuenta que en esta etapa de la compilación, el árbol es definitivamente solo un árbol como se ve en la Figura 2.13. No hay ciclos.

Los errores que pueden ocurrir durante el análisis, llamados errores de sintaxis incluyen cosas como las siguientes, en C:

- `42 = x * 3`
- `i = /5`
- `j = 4 * (6 - x;`

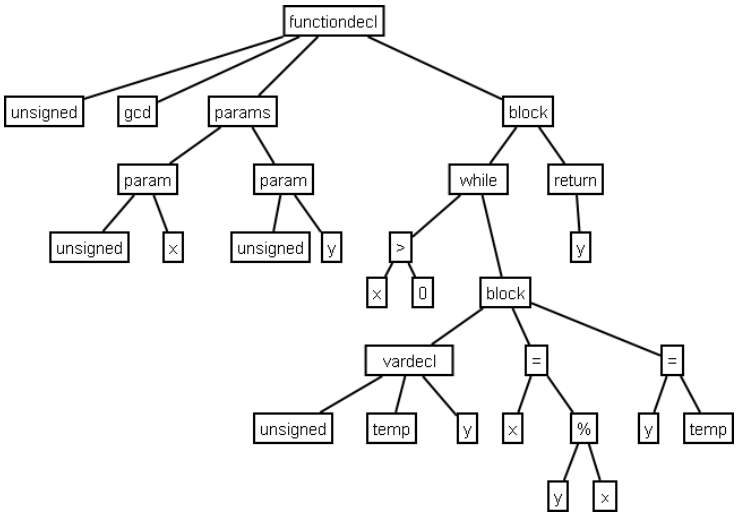

Figura 2.12: Árbol sintáctico abstracto. Ray Toal, Loyola Marymount University.

2.7.1.4. Análisis semántico

Durante el análisis semántico se deben verificar las reglas de legalidad y, al hacerlo, atar las piezas del árbol de sintaxis (resolviendo las referencias de los identificadores, insertando operaciones de conversión para coerciones implícitas, etc.) para formar un gráfico semántico.

Continuando con el ejemplo anterior, el AST se puede apreciar como en la Figura 2.14.

Obviamente, el conjunto de reglas permitidas es diferente para cada idioma. Los ejemplos de que se pueden ver en un lenguaje similar a Java incluyen:

- Múltiples declaraciones de una variable dentro de un ámbito.
- Referencia a una variable antes de su declaración.
- Referencia a un identificador que no tiene declaración.
- Violar las reglas de acceso (público, privado, protegido, ...).
- Demasiados argumentos en una llamada de método.
- No hay suficientes argumentos en una llamada de método.
- Tipo de desajustes (hay toneladas de estos).

2.7.1.5. Generación de código intermedia

El generador de código intermedio produce un gráfico de flujo formado por *tuplas* agrupadas en bloques básicos. Para el ejemplo anterior, se ve en la Figura 2.15.

2.7.1.6. Mejoras de código independientes de la arquitectura

La mejora de código que se realiza en el gráfico semántico o en el código intermedio se denomina optimización de código independiente de la arquitectura. En la práctica hay una gran cantidad de optimizaciones conocidas[100][85] (mejoras), pero ninguna realmente se aplica a nuestro ejemplo de ejecución.

2.7.1.7. Generación de código

No se pretende que el lector interprete el assembler, sino que se aprecie que mediante una optimización la lógica del programa se mantiene y su salida es más pequeña.

La generación de código produce el código de destino real, o algo cercano. Esto es lo que se obtiene al ensamblar con gcc 6.3 orientado a x86-64, sin ninguna optimización.

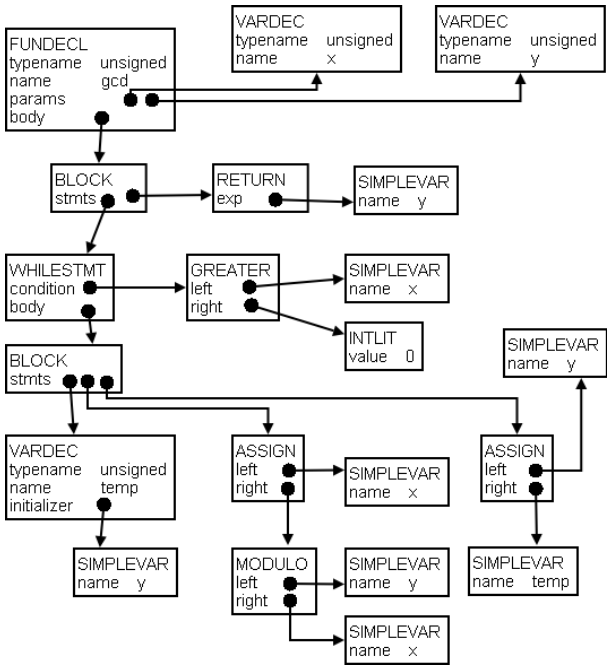

Figura 2.13: Vista de objetos del AST. Ray Toal, Loyola Marymount University.

```
gcd(unsigned int, unsigned int):
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
.L3:
    cmpl    $0, -20(%rbp)
    je      .L2
    movl    -20(%rbp), %eax
    movl    %eax, -4(%rbp)
    movl    -24(%rbp), %eax
    movl    $0, %edx
    divl    -20(%rbp)
    movl    %edx, -20(%rbp)
    movl    -4(%rbp), %eax
    movl    %eax, -24(%rbp)
    jmp     .L3
.L2:
    movl    -24(%rbp), %eax
    popq    %rbp
    ret
```

Listing 2.3: Ejemplo assembler sin mejora

2.7.2. Mejora de código dependiente de la arquitectura

Por lo general, la fase final en la compilación es limpiar y mejorar el código objetivo. Para el ejemplo anterior, se obtiene lo siguiente al configurar el nivel de optimización en -O3:

```

gcd(unsigned int, unsigned int):

        testl   %edi, %edi
        movl    %esi, %eax
        jne     .L3
        jmp     .L7
.L5:
        movl    %edx, %edi
.L3:
        xorl    %edx, %edx
```

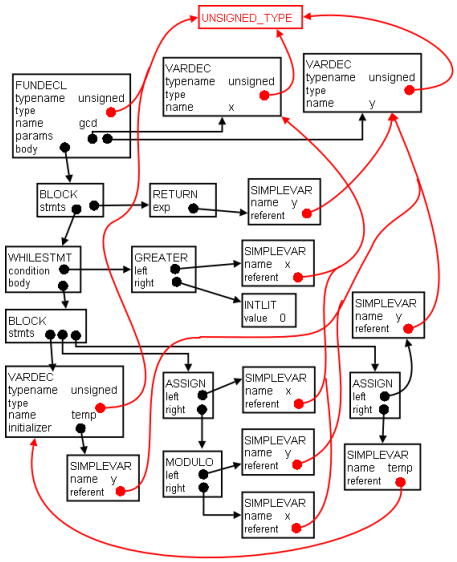

Figura 2.14: AST con análisis semántico. Ray Toal, Loyola Marymount University.

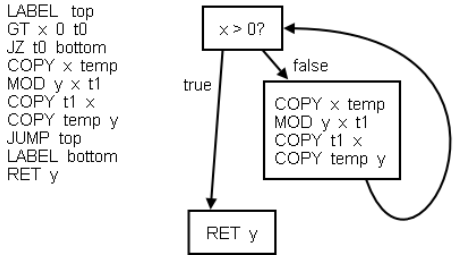

Figura 2.15: Gráfico de flujo intermedio. Ray Toal, Loyola Marymount University.

```
12          divl    %edi
13          movl    %edi, %eax
14          testl   %edx, %edx
15          jne     .L5
16  .L1:
17          movl    %edi, %eax
18          Ret
19  .L7:
20          movl    %esi, %edi
21          jmp     .L1
```

Listing 2.4: Ejemplo assembler con mejora

2.8. ¿Por qué compiladores?

Siempre se pone el foco y la responsabilidad del lado del lenguaje en el que los desarrolladores programan, pero más allá de testear, se está totalmente confiando en que el compilador que se utiliza no posea fallas de seguridad, y que como mínimo no vaya a introducir nuevas en el código desarrollado.

La seguridad informática es un desafío bastante amplio impuesto sobre el sistema. Sólo se necesita una parte que sea insegura y todo el sistema se vuelve inseguro. Sería muy dificultoso para un compilador corregir automáticamente código inseguro, ya que para ello, debería tener en su concepción una gran cantidad de aspectos pragmáticos aún no conocidos en el momento de su construcción. Pero más allá de lo anterior, el compilador está colocado perfectamente en una posición que permite ayudar a un programador o ingeniero de software a escribir sistemas seguros. La razón es porque el compilador es el único programa que tiene la posibilidad de mirar (casi) todas las líneas de un software. Obviamente lenguajes compilados como C/C++ pasan por

un compilador, pero lenguajes semi-interpretados como Java también tienen un compilador para generar su bytecode e incluso código assembler es típicamente pre-procesado por el compilador. Las únicas excepciones podrían ser los lenguajes puramente interpretados, con algunos otras casos muy particulares[72], aunque la interpretación puede verse como una suerte de compilación en línea, compilación bajo demanda o compilación JIT (*Just in Time*)[43][40].

¿Hasta dónde debería uno confiar que una declaración en un programa está libre de Caballos de Troya? Tal vez es más importante confiar en las personas que desarrollaron el software directamente.

Lo describe claramente Ken Thompson, desde 1984 en su artículo **Reflections on trusting trust** (reflecciones en confiar en la confianza):

> *"La moral es obvia. No podés confiar en un código que no creaste en su totalidad. (Especialmente el código de compañías que emplean a personas como yo). Ninguna verificación o escrutinio a nivel de fuente te protegerá de usar código no confiable. Al demostrar la posibilidad de este tipo de ataque, elegí el compilador de C. Podría haber elegido cualquier programa de manejo de software, como un ensamblador, un cargador o incluso un microcódigo de hardware. A medida que el nivel del programa disminuye, estos errores serán cada vez más difíciles de detectar. Un bug de microcódigo bien instalado será casi imposible de detectar."*
>
> – Ken Thompson

El "ataque"del cual habla está separado en dos etapas, y consta de cómo hipotéticamente modifica un compilador para generar backdoors (puertas traseras). En la primera etapa, el código en C contiene código visiblemente obvio para modificar la lógica dado un patrón determinado, de un programa de acceso, y como bien dice: *Tal descarado código no pasaría desapercibido por mucho tiempo. Incluso la lectura más casual del código fuente del compilador de C levantaría sospechas.*

En la segunda etapa es donde la modificación de un compilador con fines maliciosos se pone interesante. En adición a agregar código al compilador de C para corromper un programa en particular también agrega código al compilador para corromperse a sí mismo. Es decir, que si se compilara con ese compilador maligno otro compilador en base a un código limpio, se agregaría la lógica necesaria al nuevo compilador para que cada vez que procese un software de acceso le inserte un *backdoor*.

No necesariamente tiene que ser con intenciones malignas, así como hay errores de seguridad en la mayoría de los programas, un compilador no está exento a esto, y las mismas fallas podrían generar comportamientos inesperados en la compilación.

2.8.1. Optimizaciones y comportamiento inesperado

Investigadores del laboratorio de Ciencias de la computación e inteligencia artificial del MIT publicaron un paper (*Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior*) analizando el problema optimización de código inestable, el cual es código que el compilador quita porque incluye comportamiento indefinido. Dicho código es el que puede comportarse de maneras inesperadas, como dividir por cero, una desreferencia de puntero nulo y *buffer overflows* (rebalse de búferes).

A diferencia de otro código, los desarrolladores de compiladores son libres de decidir cómo lidiar con este tipo de comportamientos. En algunos casos eligen eliminar ese código completamente, lo cual puede llevar a vulnerabilidades si el código en cuestión posee chequeos de seguridad.

Estos investigadores estudiaron una docena de compiladores C/C++ más comúnmente utilizados para observar cómo lidian con código indefinido. Encontraron que, con el tiempo, los compiladores están poniéndose más agresivos en cómo lidian con ese código, usualmente sólo quitándolo, incluso por defecto o en pequeños niveles de optimización. Ya que C/C++ es bastante liberal respecto a permitir comportamiento indefinido, es más susceptible a errores y amenazas de seguridad como resultado de código inestable.

Tanto es el hecho que existe una categoría de seguridad específicamente para estos casos, llamada *optimización insegura de compilación*. A continuación un ejemplo.[215]

```
1 void GetData(char *MFAddr) {
2     char pwd[64];
3     if (GetPasswordFromUser(pwd, sizeof(pwd)))
4      if (ConnectToMainframe(MFAddr, pwd)) {
5             // Interaction with mainframe
```

```
6         }
7       }
8       memset(pwd, 0, sizeof(pwd));
9 }
```

Listing 2.5: Código vulnerable

El código en el ejemplo del cuadro 2.5 se comportaría correctamente si se ejecutara de manera literal, pero si el código se compila utilizando la opción general de optimización, como las de los compiladores de C++ Microsoft Visual C ++ ® .NET o GCC 3.x, la llamada a memset() se eliminará como un almacenamiento inactivo porque el búfer *pwd* no se usa después de que su valor se sobrescribe. Debido a que el búfer pwd contiene un valor sensible, la aplicación puede ser vulnerable a un ataque si los datos se quedan en la memoria. Si los atacantes pueden acceder a la región correcta de la memoria, pueden usar la contraseña recuperada para obtener el control del sistema.[209]

Otro ejemplo más impactante que terminó afectando a muchos sistemas Linux/GNU, fue un error introducido en el kernel debido a una optimización. Si bien los detalles técnicos sobre esto son un poco complejos, en general, lo que sucede puede explicarse fácilmente. El código vulnerable se encuentra en la implementación de una librería. Básicamente, lo que sucede aquí es que el desarrollador inicializó una variable (**sk** en el fragmento de código a continuación) a un cierto valor que puede ser *NULL*. El desarrollador verificó correctamente el valor de esta nueva variable par de líneas más tarde y, si es 0 (*NULL*), simplemente devuelve un error. El código se ve así:[34]

```
1 struct sock *sk = tun->sk;  // initialize sk with tun->sk
2 ...
3 if (!tun)
4     return POLLERR;  // if tun is NULL return error
```

Listing 2.6: Código vulnerable 2

Este código se ve perfectamente bien, y lo es, al menos hasta que el compilador toma esto en sus manos. Mientras optimiza el código, verá que la variable ya ha sido asignada y eliminará el bloque *if* (la comprobación si *tun* es NULL) completamente del código compilado resultante. En otras palabras, el compilador introducirá la vulnerabilidad al código binario, que no existía en el código fuente. Esto hará que el kernel intente leer / escribir datos desde la dirección de memoria *0x00000000*, que el atacante puede *mapear* a la zona de usuario, controlando el flujo de ejecución.

Vulnerabilidades en intérpretes

Los intérpretes no son el foco de discusión en esta investigación, pero el contenido aplica de igualmente manera a ellos también.

Todas las vulnerabilidades alguna vez reportadas tienen asignado un identificador único llamado CVE (Common Vulnerability Enumeration). En el sitio oficial de la lista se puede encontrar que intérpretes como Ruby[148], Python[147] y PHP[146] poseen 13, 45 y 599 vulnerabilidades reportadas hasta el momento de esta redacción.

2.9. Tecnologías blockchain

En esta sección se enumeran tecnologías cuya estructura principal es del tipo blockchain, con las que se trabajó en esta tesis y forman parte de las principales razones por la que fue impulsada esta investigación, dada la popularidad que han obtenido en los últimos años.

2.9.1. Bitcoin

La idea de tener una moneda digital no es nueva. Antes de las *cryptocurrencies* (criptomonedas), existieron muchos intentos de crear una. El principal desafío que la mayoría encontraba era solucionar el problema de *double spend* (doble gasto). Un bien digital tiene que de alguna manera ser utilizado sólo una vez para prevenir que se copie y efectivamente falsifique.[104]

Diez años antes de las *cryptocurrencies*, el concepto había sido introducido por un ingeniero en computación llamado Wei Dai. En 1998, publicó un paper donde discutió una propuesta llamada *B-money*. Discutió la idea de una moneda digital, que podía ser enviada junto con un grupo de pseudónimos irrastreables. Ese mismo año, otro intento bajo el nombre *Bit Gold* fue escrito por Nick Szabo. Bit Gold incluyó la posibilidad de crear una moneda digital descentralizada. La idea

de Szabo fue motivada por las ineficiencias que se encuentran hoy en día en el sistema financiero tradicional (FIAT), como requerir metales para construir monedas, y para reducir la cantidad de confianza que hay que tener para realizar transacciones. Si bien ambos proyectos nunca fueron oficialmente ejecutados, fueron parte de la inspiración de *Bitcoin*[66].

En el 2008 Satoshi Nakamoto publicó un artículo (*white paper*) llamado *Bitcoin: A Peer-to-Peer Electronic Cash System*, describiendo la funcionalidad de la red blockchain de Bitcoin: *una red sin permiso tolerante a fallos bizantinos* (resistencia de un sistema informático tolerante a faltas) *criptográficamente segura.*

> *"Mucha gente descarta automáticamente las e-currency como una causa perdida por todas las compañías que fallaron hacerlo desde 1990. Espero que sea obvio que lo que las llevó a la perdición fue su naturaleza de tener un sistema central controlado. Creo que esta es la primera vez que estamos intentando un sistema descentralizado, no basado en la confianza."*
>
> – Satoshi Nakamoto.

2.9.2. Estructura blockchain

La blockchain es un libro de registro distribuido punto a punto, seguro, y se utiliza para registrar transacciones. El contenido del registro solo se puede actualizar agregando otro bloque vinculado al anterior. También se puede ver como una plataforma donde las personas pueden realizar transacciones de todo tipo sin la necesidad de un árbitro central o de confianza.

La base de datos creada se comparte entre los participantes de la red de manera transparente, por lo que todos pueden acceder a su contenidos. La gestión de la base de datos se realiza de forma autónoma utilizando redes punto a punto y un servidor de timestamping, es decir que permite demostrar que una serie de datos han existido y no han sido alterados desde un instante específico en el tiempo. Cada bloque en una blockchain está organizado de tal manera que hace referencia al contenido del bloque anterior.[112]

Los bloques que forman una blockchain contienen lotes de transacciones validadas por los participantes en una red. Cada bloque viene con un hash criptográfico de un bloque anterior en la cadena.[112]

> Bitcoin y Blockchain no son sinónimos. Blockchain es la estructura que Bitcoin posee como base para impulsar sus aplicaciones.

2.9.3. Smart Contracts

Un smart contract es un código de computadora corriendo sobre una blockchain que posee un conjunto de reglas bajo las cuales las partes de ese contrato acceden para interactuar uno con el otro. Si-y-cuando esas reglas se cumplan, el acuerdo es automáticamente impuesto. El código del smart contract facilita, verifica, y hace cumplir la negociación o performance de un acuerdo o transacción. Es la forma más simple de automatización descentralizada.

Es un mecanismo que envuelve bienes digitales de dos o más partes, donde algunas o todas las partes depositan bienes en el contrato y los bienes son automáticamente redistribuidos entre esas partes dependiendo de una fórmula basada en determinados datos, que no son conocidos al momento de la iniciación del contrato.

El término se presta a confusiones porque un smart contract no es inteligente, ni debería ser confundido con un contrato legal:

- Un smart contract es tan inteligente como las personas que lo programaron con la información que tenían disponible en ese momento.
- Mientras que tienen el potencial de poder ser contratos legales no deben confundirse con contratos legales aceptados por las cortes y la ley. Sin embargo, probablemente se vaya a ver una fusión de estos dos en el futuro.

Si bien la descripción anterior es acertada, y una definición que se puede observar en muchas fuentes, el autor cree que es demasiada específica. Un smart contract es algo más abstracto.

Entonces, se puede resaltar lo más importante de la siguiente manera: son auto verificables, auto ejecutables y resistentes a la manipulación; pueden transformar responsabilidades legales en procesos automatizados, garantizar un gran grado de seguridad, reducen la dependencia de intermediarios confiables y poseen costos de transacción bajos.

2.9.4. Ethereum

En el 2013 un jóven llamado Vitalik Buterin[210], co-fundador de la Bitcoin Magazine propuso **Ethereum** en un paper llamado *'Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform'*. Preocupado por las limitaciones de Bitcoin, comenzó a trabajar en lo que él pensaba que sería una blockchain maleable que pueda realizar varias funciones además de ser una red punto a punto. Ethereum[111] nació como una nueva blockchain pública con funcionalidades adicionales, en comparación con Bitcoin.

> *"Bitcoin es excelente como dinero digital, pero su lenguaje de scripting es demasiado débil para que se puedan construir aplicaciones avanzadas de forma seria."*
>
> – Vitalik Buterin

Lanzado oficialmente en 2015, Ethereum ha evolucionado hasta convertirse en una de las aplicaciones más grandes de la tecnología de blockchain, dada su capacidad para respaldar los *smart contracts* utilizados para desarrollar aplicaciones descentralizadas. La plataforma ha logrado reunir una comunidad activa de desarrolladores que la han visto convertirse en un verdadero ecosistema.

Procesa la mayor cantidad de transacciones diarias gracias a su capacidad para soportar contratos inteligentes y aplicaciones descentralizadas. Su capitalización de mercado también se ha incrementado significativamente en el espacio de las criptomonedas.

En la Figura 2.16 se muestra una cronología para repasar visualmente el proceso.

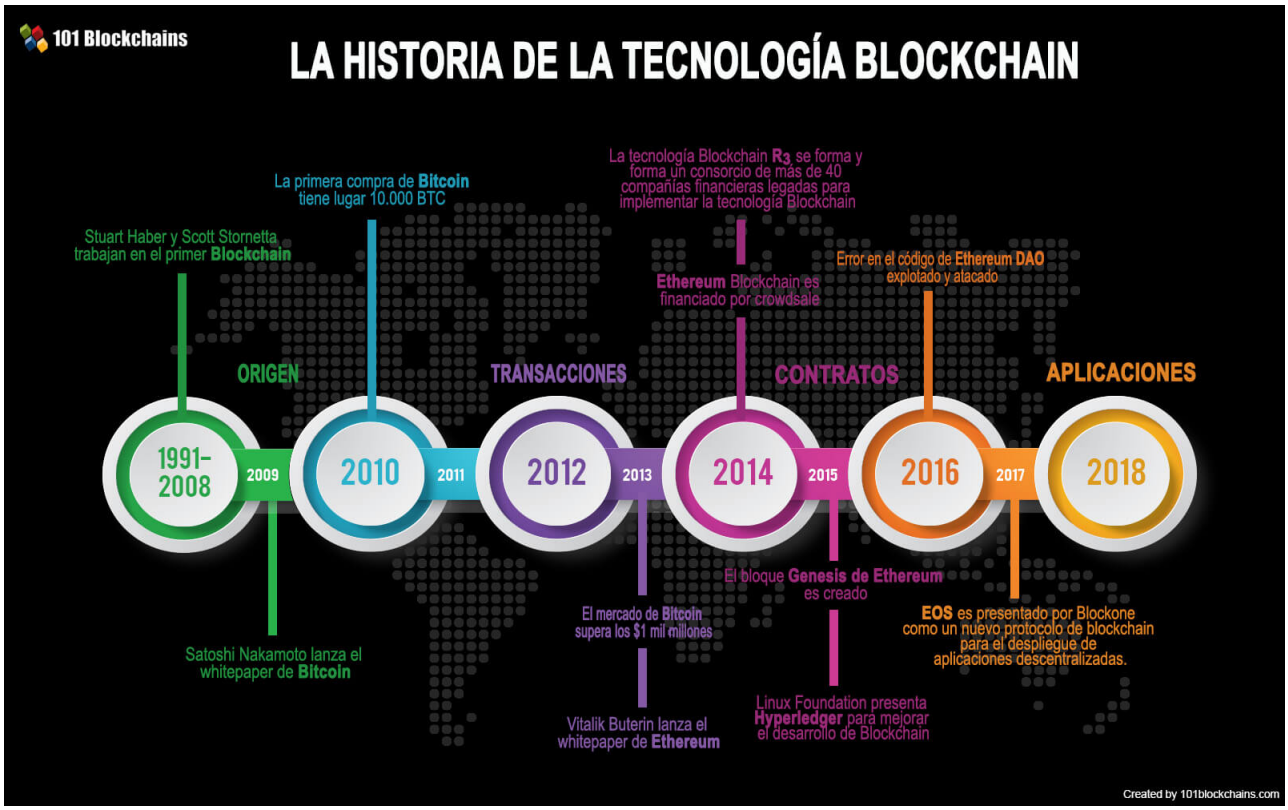

Figura 2.16: Cronología de tecnologías blockchain por 101blockchains.

2.9.5. Ethereum Virtual Machine

La Ethereum Virtual Machine (**EVM**) es el contexto en el cual los smart contracts de la red de Ethereum viven y se ejecutan. Posee un stack de registros de 256 bits, diseñada para correr el código

exactamente como fue desarrollado. Es el mecanismo de consenso fundamental para Ethereum. La definición formal de la EVM está especificada en el Ethereum Yellow Paper.

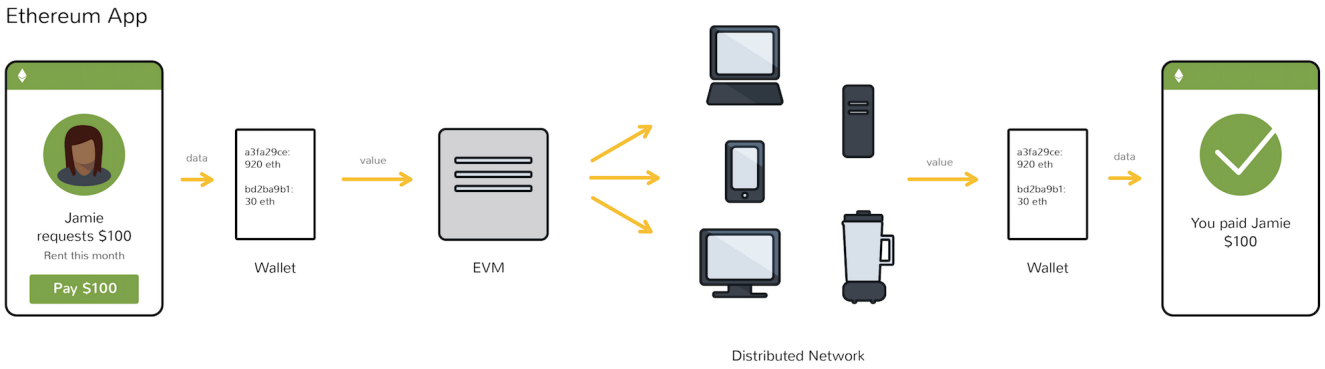

Figura 2.17: Diagrama de despliege, por Coindesk.

2.9.6. ¿Cómo es Ethereum diferente a Bitcoin?

Si bien hay varias similitudes entre Ethereum y Bitcoin, también hay diferencias significativas:

1. Bitcoin opera sólo como cryptocurrency, mientras que Ethereum ofrece diversos métodos de interacción, incluyendo cryptocurrency (Ether), smart contracts y la Ethereum Virtual Machine.

2. Están basados en distintos protocolos de seguridad: Ethereum eventualmente utilizará un sistema *proof of stake* (PoS) en contraposición al sistema *proof of work* (PoW) utilizado por Bitcoin, aunque en este momento utilizan el mismo hasta que el cambio suceda. *Explicar las diferencias y alcances de PoS vs PoW escapa del contexto de la investigación.*

3. El tiempo promedio de generación de un bloque en Ethereum es de 12 segundos contra 10 minutos de Bitcoin. Esto se traduce a más confirmaciones de bloque, lo que permite a los mineros de Ethereum a completar más bloques y recibir más Ethers.

4. Ethereum usa un sistema de cuentas en donde los valores en Wei (mínima unidad de representación de un Ether, `1x10^18`) son debitados de una cuenta hacia otra, al contrario del sistema de UTXO (*Unspent Transaction Output*) de Bitcoin, que es más análogo a gastar dinero y recibir cambio. Ver Figura 2.18.

5. Ethereum posee una máquina virtual que permite escribir smart contracts con un lenguaje bytecode *turing completo*.

6. La moneda de Bitcoin se representa mediante la sigla BTC, y la de Ethereum mediante ETH.

2.9.7. Lenguajes de programación para la EVM

Los smart contracts de Ethereum pueden ser escritos en **Solidity**[153] (un lenguaje de bibliotecas que posee similitudes con C y JavaScript), **Serpent**[152] (similar a Python, pero deprecado), **LLL**[151] (un lenguaje bajo nivel similar a Lisp), **Mutan**[211] (basado en Go, pero también deprecado), **Vyper**[264] (orientado a la investigación, fuertemente tipado, basado en Python, recursivo), y recientemente la empresa **BlockStack**[145] presentó **Clarity**[165] (tiene intencionado optimizar la predictibilidad y la seguridad).

Esto convierte a Solidity en el lenguaje más utilizado, o de facto, a la hora de desarrollar smart contracts en **Ethereum Network**[159].

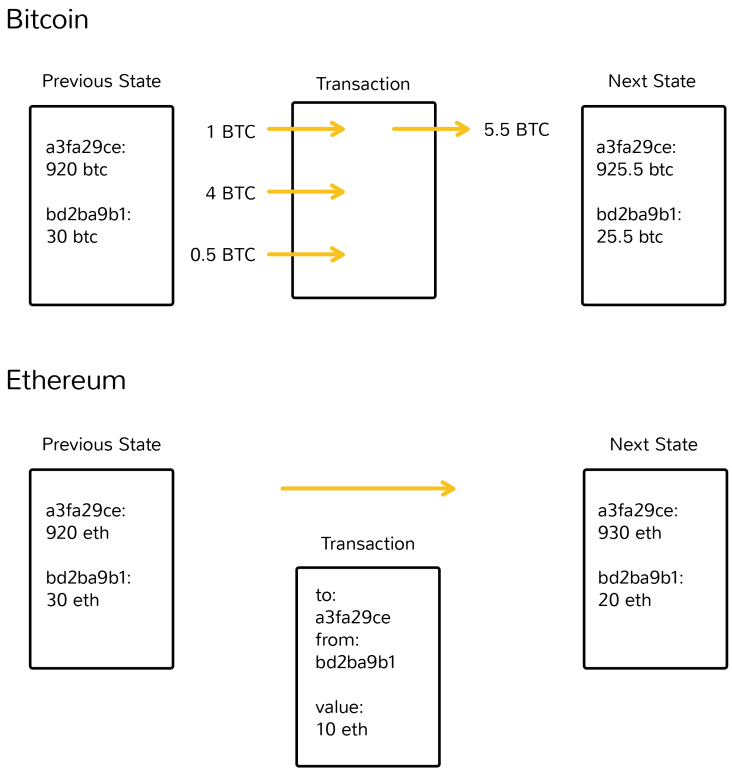

Figura 2.18: Diferencia del estado de las transacciones por Coindesk.

2.10. Auditando blockchains

Entonces, repasando un poco ahora que se posee la terminología necesaria: se puede desarrollar un smart contract en Solidity, que es compilado a bytecode, y si todo es correcto, será puesto en funcionamiento (*deployado*) de manera inmutable a la blockchain. Allí todos los nodos que poseen máquinas virtuales tendrán una copia de ese contrato, y de ahora en más deberán llegar a un consenso en cuanto a si los interacciones que lo involucran y sus modificaciones de estado son correctos.

Entendiendo esto, se pueden definir las siguientes maneras de auditar una tecnología blockchain que utiliza smart contracts:

- **Análisis estático al código**
  Análisis al código mediante la automatización de herramientas que buscan patrones comunes susceptibles a errores y fallas de seguridad.

- **Análisis dinámico del contrato**
  Analizar mediante cualquier medio el comportamiento del contrato, interactuando en vivo en la blockchain. Generalmente se replica en un entorno seguro, como en una red de testeo, conocidas como testnets.

- **Ingeniería inversa al bytecode deployado**
  Analizar directamente el código resultante de la compilación, es decir el bytecode, lenguaje que interpreta la EVM. Generalmente se utiliza para observar si se introdujeron errores en el proceso de deployment, o para los contratos cuyo código fuente no se posee.

- **Verificación formal**
  Es el mecanismo mediante el cual se prueba la correctitud de un contrato, basada en la verificación formal de lo que se supone que debería hacer. Una manera de realizar análisis estático pero sumamente compleja. Generalmente lo realiza quién lo desarrolla.

- **Ejecución simbólica**
  Mecanismo para analizar un contrato con el fin de entender qué entradas estimulan qué partes de él.

Algunas de estas técnicas implican analizar directamente el bytecode como output del compilador de Solidity por ejemplo, o el bytecode del contrato deployado en la blockchain, por más que se posea el fuente.

¿Por qué se ofrece esta alternativa? Porque es lógico asumir luego de las declaraciones anteriores, que no se puede confiar en la manera en la que los compiladores optimizan o interpretan el código. Por eso es necesario comprobar que la salida interpretada por la EVM tiene de hecho la intención con la que fue diseñado.

2.11. ¿Por qué el compilador de Solidity?

En el momento de la escritura de esta sección, la moneda que maneja la Ethereum Network ether, es decir 1 **ETH**, equivale a un total de **US$**266.29. El volúmen de transacciones de ETH en dólares de las últimas 24 horas es de **US$**8.584.415.157 o 939.202 BTC. Es decir, con la cotización del dólar en Argentina, equivale a un total de **AR$**372.282.118.457,18[142] El **ether** no es el único bien que posee muchas transacciones diarias. También existen los tokens, que se pueden comprar con monedas. Una moneda opera independientemente, mientras que un token posee un uso específico en el ecosistema de un proyecto. Son creadas sobre la red de Ethereum, gestionadas mediante smart contracts. Algunos ejemplos y sus volúmenes en las últimas 24 horas de esta redacción son:

- Dai: DAI $22,038,619 USD
- Maker: MKR $1,514,750 USD
- OmiseGo: OMG $119,081,815 USD
- Basic Attention Token: BAT $66,837,473 USD
- 0x: ZRX $17,960,636 USD
- Augur: REP $6,113,229 USD
- Usd Coin Classic: USDC $177,029,191 USD
- Paxos Standard Token: PAX $128,207,385 USD
- Status: SNT $16,693,466 USD
- Golem: GNT $1,843,572 USD
- Decentraland: MANA $10,516,180 USD

Esto no incluye otras plataformas de Ethereum paralelas como **EOS**, **Tron**, **IOST** o más, en donde algunos de sus volúmenes en las últimas 24 horas supera los miles de millones de dólares en sólo alguna de las aplicaciones que corren, como por ejemplo **TRONBet** para Tron, **dice2.win** para ETH, o **EOSJacks** para EOS[144].

El interés que ha habido en este ambiente por la cantidad de dinero que manejan ha aumentado en gran cantidad desde sus comienzos. En el sitio de Ledger, fabricantes de billeteras digitales en hardware, muestran un diagrama de tiempo[113] con todos los hackeos que han recibido los sitios de intercambio (**exchanges**) desde sus comienzos. Los costos en pérdidas superan el billón de dólares en costos.

2.11.1. Smart contract hacks

En esta sección se presentan tres ejemplos de *hackeos* de alto impacto, de los más populares que han acontecido desde el momento en el que se empezó a interactuar con Smart Contrats en la red de Ethereum.

2.11.1.1. El DAO hack

Una **DAO** es una Organización Autónoma Descentralizada o una organización que se ejecuta a través de smart contracts. Las decisiones se toman digitalmente mediante la votación de los miembros de la organización, eliminando la necesidad de documentos y personas que gobiernan y, en consecuencia, un sistema de control descentralizado.

En este caso, la DAO comprendía una serie de contratos inteligentes destinados a democratizar cómo se financiaban los proyectos de Ethereum. Un hacker, al darse cuenta de una vulnerabilidad, robó USD$ 3.6 millones de Ethers mediante la explotación de una función (*fallback*) en el código que fue expuesto a una vulnerabilidad conocida como *reentrancy* (re-entrada). Para recuperar los fondos, se tuvo que tomar una dura decisión, atentando contra la fe de los usuarios, que condujo a la creación de Ethereum Classic y Ethereum como dos cadenas separadas.[67]

2.11.1.2. Multi-firma Wallet hack de Parity

La empresa **Parity** hizo "billeteras" (*wallets*) de software multi-firma (multi-sig) para la gestión de la criptomoneda Ether. Estas wallets multi-sig eran contratos inteligentes disponibles en una base de código abierto que requerían más de una firma digital (clave privada) antes de que el Ether asociado con ellos pudiera ser aprobado para su transferencia. Un hacker desconocido robó 150,000 Ethers, alrededor de USD$30.000.000 en ese momento[74].

2.11.1.3. Wallets de Parity congeladas por un usuario

Apenas unos meses después del hackeo en julio de 2017, un usuario explotó accidentalmente una vulnerabilidad en el código de la biblioteca de Parity para las wallets multi-sig, congelando más de 513.774,16 ETH, lo que representaba en esa época más de USD$106.864.992[70].

En conclusión, *no solo se tiene la importancia de auditar un compilador por las razones mencionadas anteriormente, sino que al auditar un compilador como el de Solidity, se estaría auditando un compilador cuyo código emitido e inmutable corre en todas las máquinas virtuales de todos los nodos de una red. Es decir que si se encuentra la manera de, vulnerar los contratos para obtener su balance, o la manera de emitir un mensaje que detenga el funcionamiento de toda la red, sería catastrófico.*

Capítulo 3

Estado del arte

Teniendo en cuenta que no existe una teoría o mecanismo formalmente establecido para la auditoría específica de compiladores, más allá de lo presentado en el capítulo 2, podría pensarse que existen varias maneras de realizar dicha tarea.

Una de las maneras más sencillas de auditar el proceso, de transformar el código fuente a un ejecutable, sería generar binarios para el mismo código fuente de un lenguaje, mediante distintos compiladores, y comparar los resultados.

También se podría generar un mismo compilador, con distintos compiladores, para una misma arquitectura, y observar si las distintas optimizaciones aplicadas sobre el mismo compilador hacen que las generaciones realizadas por el resultante difieran tras realizarle un análisis estático.

Clarificación con un ejemplo. Como primer paso, se compila el código fuente para **gcc** utilizando dos compiladores distintos, **gcc** y **clang**; por lo tanto, ambos deberían devolver un ejecutable cada uno, en este caso **gcc-gcc** y **gcc-clang**. Luego, una de las maneras más sensatas de corroborar que sus lógicas son idénticas es compilar el mismo archivo con ambos compiladores resultantes, y comparar sus resultados. Como la lógica, en teoría, es la misma, ambos deberían manifestarse de la misma manera, y de hallar alguna diferencia en el programa resultante se comprobaría que alguno falló, probablemente por optimizaciones con comportamiento inesperado.

A continuación se presenta una selección de proyectos con las propuestas más interesantes que se encuentran en el marco de esta investigación. La mayoría de las propuestas están presentadas mediante una combinación de traducciones, adaptaciones, e interpretaciones de los artículos originalmente publicados, incluyendo un análisis de sus herramientas, en el caso de estar públicas.

3.1. CSMith (2009-2013)

CSmith es una herramienta de generación de casos de test randomizados, utilizada para encontrar errores en compiladores durante 5 años. A diferencia de herramientas antecesoras, Csmith genera programas que cubren un gran subconjunto del lenguaje C. Evita los comportamientos indefinidos y no especificados que podrían destruir la habilidad de encontrar automáticamente bugs por código erróneo.

Todos los compiladores que se testearon con esta herramienta se detuvieron inesperadamente (crash), y también silenciosamente generaban código erróneo cuando eran presentados con input válido.

```
1 int foo (void) {
2     signed char x = 1;
3     unsigned char y = 255;
4     return x > y;
5 }
```

Listing 3.1: Código que produjo bug en GCC

El código del recuadro3.1 pertenece a un bug hallado en una versión de GCC incluído con Ubuntu Linux 8.04.1 para x86. En todos los niveles de optimización compilaba esa función de forma que en su ejecución retorne el valor "1"; cuando el resultado correcto es "0". Lo que sucedió fue que el compilador de Ubuntu había sido fuertemente modificado a través de lo que se conoce como "parche", ya que la versión base de GCC no poseía ese bug[42].

Csmith genera un programa en C; luego un código de prueba (conocido como "arnés de prueba") compila ese programa utilizando diversos compiladores, ejecuta los ejecutables, y compara las salidas. A pesar de que esta estrategia ya había sido utilizada previamente [32][9][29], las técnicas de generación de Csmith, para ese momento, avanzaron sustancialmente el estado del arte generando programas aleatorios que son expresivos—conteniendo código complejo utilizando muchas de sus características—mientras también aseguraban que cada programa generado tenga una sola interpretación. Para poder hacer esto, un programa no debe poder ejecutar ningún tipo de comportamientos indefinido, ni debe depender de ninguno de los 52 tipos de comportamiento no especificado que están descritos en el estándar C99[162].

Los autores de la herramienta claman que Csmith es efectiva para buscar bugs, en parte porque genera casos de testeo que exploran combinaciones atípicas del lenguaje. Que sea código atípico no significa que no sea importante, sin embargo; no se encuentra bien representado en las suites de testeo existentes. Los desarrolladores que se aventuran fuera de los caminos más testeados que representan lo que podría denominarsela "zona de confort"del compilador – por ejemplo escribiendo un kernel o sistemas embebidos, utilizando opciones de compilación esotéricas (de uso extremadamente específico), o generando código automáticamente – pueden encontrarse bugs bastante frecuente.

Este proyecto comenzó como un **fork** (bifurcación) de **Randprog**[30], un proyecto ya existente que genera programas aleatorios en C de 1600 líneas de código. En sus trabajos tempranos extendieron y adaptaron Randprog para encontrar bugs en la parte de la traducción de accesos a objetos calificados de volátiles, lo que dió como resultado a un programa de 7000 líneas de código. Los autores convirtieron Randprog en Csmith, un programa C++ de 40,000 líneas para generar programas de C aleatorios. En comparación con Randprog, Csmith puede generar programas de C que utilizan una gama mucho más amplia de características de C, incluido el flujo de control complejo y estructuras de datos como punteros, matrices y estructuras.

El programa Csmith utiliza pruebas diferenciales aleatorias. Las pruebas aleatorias[25], también llamadas fuzzing[4], consisten en un método de test black box en el que las entradas de prueba se generan aleatoriamente. Las pruebas diferenciales aleatorias[9] tienen la ventaja de que no se necesita un oráculo (principio heurístico o mecanismo por el cual podremos reconocer un problema, y determinar si es correcto o no). para los resultados de las pruebas. Explota la idea de que si uno tiene implementaciones deterministas múltiples de la misma especificación, todas las implementaciones deben producir el mismo resultado de la misma entrada válida. Cuando dos implementaciones producen salidas diferentes, una de ellas debe ser defectuosa. Dadas tres o más implementaciones, un evaluador puede usar el voto para determinar heurísticamente qué implementaciones son incorrectas. La Figura 3.1 muestra cómo usaron estas ideas para encontrar errores de compilación.

3.1.1. Resultados obtenidos

Desde el año 2009 al 2013 encontraron 476 bugs en GCC[154] y LLVM[208].

La lista de bugs es accesible desde su repositorio[143] en github, también desde el sitio del departamento de Ciencias de la computación de UTAH.

El proceso de testear al azar es útil pero posee desventajas: no se puede saber cuándo dejar de testear; optimizar las probabilidades no es trivial; generar salidas expresivas que sean realmente correctas no es sencillo; y finalmente, es limitado al lenguaje. Csmith declara ser el ataque que utiliza como técnica al *fuzzing*, como el más extensivo en comparación con los compiladores de su época.

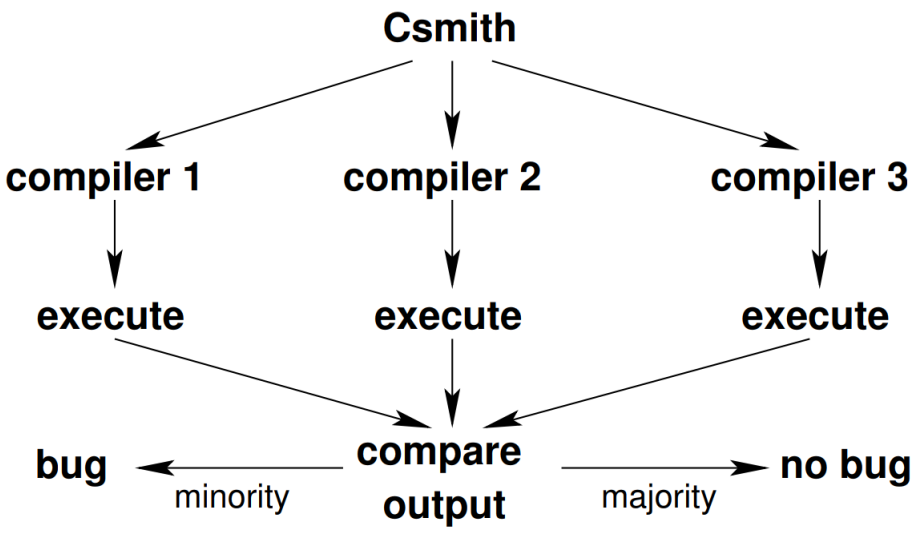

Figura 3.1: Ejecución en distintos compiladores

Resumidamente, la función de CSMith es aplicar técnicas de fuzzing el compilador utilizando programas generados al azar basados en la gramática, del lenguaje C, interpretando los resultados.

3.2. DeepSmith (2018)

Los autores de la herramienta presentan **DeepSmith**[90], como un novedoso enfoque de aprendizaje automático para acelerar la validación del compilador a través de la inferencia de modelos generativos para las entradas del compilador. Su enfoque infiere un modelo, aprendido de la estructura de código real basado en un gran códigos open source. Luego, utiliza el modelo para generar automáticamente decenas de miles de programas realistas. Finalmente, aplican metodologías de pruebas diferenciales establecidas para exponer errores en los compiladores.

Se ha aplicado este enfoque al lenguaje de programación **OpenCL**, exponiendo automáticamente los errores en los compiladores de OpenCL con poco esfuerzo de su lado. En 1.000 horas de pruebas automatizadas de compiladores comerciales y de código abierto, descubrieron errores en todos ellos.

3.2.1. Comparación con CSmith

CSmith se desarrolló a lo largo de los años y consta de más de 41k líneas de código C ++ escritas manualmente. Al unir estrechamente la lógica de generación con el lenguaje de programación de destino, cada característica de la gramática debe diseñarse de forma minuciosa y experta para cada nuevo idioma de destino.

Por ejemplo, adaptar CSmith de C a **OpenCL**[7] - una tarea que parecería ser simple - les tomó 9 meses y 8k líneas adicionales de código. Dada la dificultad de definir una nueva gramática, generalmente solo se implementa un subconjunto del lenguaje.

Su metodología utiliza los avances recientes en *deep learning* (conjunto de algoritmos de *machine learning*) para construir automáticamente modelos probabilísticos de cómo los humanos escriben el código, en lugar de definir meticulosamente una gramática con el mismo fin. Al entrenar una red neuronal profunda en un corpus de código *manualmente escrito*, es capaz de inferir tanto la sintaxis como la semántica del lenguaje de programación. El enfoque de los autores de la herramienta esencialmente enmarca la generación de programas aleatorios como un problema de modelado de

lenguaje. Esto simplifica y acelera enormemente el proceso.

Lo más interesante de esto es que las herramientas *infieren la sintaxis, la estructura y el lenguaje de programación*. Utilizan ejemplos del mundo real, no a través de una gramática definida por expertos. El tamaño promedio de los casos de prueba es dos órdenes de magnitud más pequeño que el estado del arte, sin ningún proceso de reducción costoso, y toma menos de un día para entrenar.

Los autores descubrieron un número similar de errores que el estado del arte, pero también encontraron errores que el trabajo anterior no pudo, cubriendo más componentes del compilador. Y en el modelado de código manualmente escrito, sus casos de prueba son más interpretables que otros enfoques.

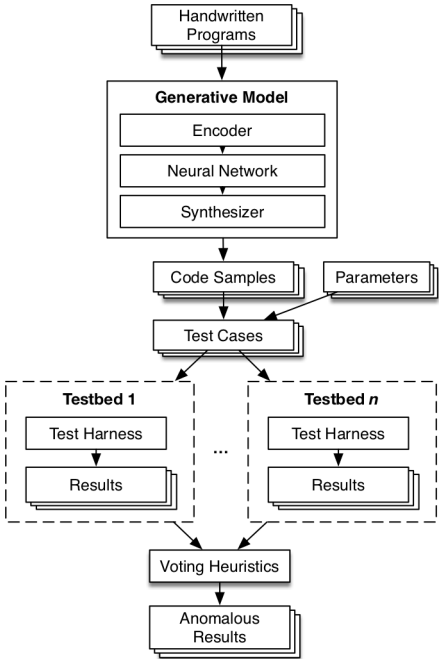

Figura 3.2: Arquitectura de DeepSmith

3.2.2. Extensibilidad del modelado de lenguajes

Una gran parte de la arquitectura de DeepSmith es independiente del lenguaje;ya que solo requiere un corpus, un codificador y un arnés para cada nuevo lenguaje. Esto potencialmente reduce significativamente la barrera de entrada en comparación con los fuzzers basados en gramática anteriores (la gran mayoría). Para explorar esto, deciden intentarlo con un lenguaje reciente, siendo este Solidity.

La razón de seleccionar a Solidity, se debe a que posee menos de cuatro años, carece de gran parte de las herramientas de lenguajes de programación más establecidos y los errores explotables pueden socavar la integridad de la blockchain y provocar transacciones fraudulentas.

3.2.3. Resultados iniciales

La investigación se realizó ejecutando el bucle del arnés y el generador durante 12 horas en cuatro bancos de pruebas: el solc de compilación de referencia de Solidity con optimizaciones activadas o desactivadas, y solc-js, que es una versión compilada por Emscripten del compilador de solc.

Sus resultados se resumen en la tabla que muestra la Figura 3.3.

Los resultados de la investigación demostraron numerosos casos en los que el compilador se bloquea silenciosamente y dos aserciones distintas del compilador. El primero se debe a la falta de

Compiler	±	Silent Crashes	Assertion 1	Assertion 2
solc	−	204	1	
	+	204	1	
solc-js	−	3628	1	1
	+	908	1	1

Figura 3.3: Resultados de testear solc y solc-js.

manejo de errores en las características del idioma (este problema es conocido por los desarrolladores). La fuente de la segunda afirmación es el tiempo de ejecución de JavaScript y se activa solo en la versión Emscripten, lo que sugiere un error en la traducción automática de LLVM a JavaScript.

La extensión de DeepSmith a una segunda programación requirió 150 líneas adicionales de código (18 líneas para el generador y el codificador, el resto para el arnés de prueba) y tomó aproximadamente un día. Dada la reutilización de los componentes básicos de DeepSmith, hay un costo decreciente con la adición de cada nuevo idioma. Por ejemplo, el codificador y reescritura de OpenCL, implementado utilizando LLVM, podría adaptarse a C con cambios mínimos. Dado el bajo costo de la extensibilidad, los autores de la herramienta creen que estos resultados preliminares indican la utilidad de su enfoque para simplificar la generación de casos de prueba.

3.2.4. Trabajos relacionados

La generación aleatoria de casos de prueba es un enfoque bien establecido para el problema de validación del compilador. Los enfoques anteriores se examinan en *Compiler test case generation methods: a survey and assessment*[7], *Survey of Compiler Testing Methods*[19] y se contrastan empíricamente en *An empirical comparison of compiler testing techniques*[63]. La principal pregunta de interés es cómo generar de manera eficiente los códigos que desencadenan errores.

Hay dos enfoques principales: la generación de programas, donde las entradas se sintetizan desde cero; y la mutación del programa, donde los códigos existentes se modifican para identificar comportamientos anómalos.

3.2.5. Programa de generación

En el trabajo fundacional sobre pruebas diferenciales para compiladores, McKeeman et al. presentan generadores actuales capaces de enumerar programas de una variedad de calidades, desde secuencias ASCII aleatorias hasta programas conformes con el modelo C[9]. Los trabajos posteriores han presentado generadores cada vez más complejos que mejoran en alguna métrica de interés, generalmente expresividad o probabilidad de corrección. CSmith[42] es un generador ampliamente conocido y efectivo que enumera programas al vincular funciones de lenguaje combinadas con poca frecuencia. Al hacerlo, produce programas correctos con un comportamiento claramente definido pero una funcionalidad muy poco probable, lo que aumenta las posibilidades de desencadenar un error.

Lograr esto requirió un extenso trabajo de ingeniería, la mayoría no se puede transportar a otros idiomas, e ignorar algunas características del idioma. Los generadores subsiguientes influenciados por CSmith, como Orange3[56], se enfocan en características y tipos de errores más allá del alcance de CSmith, errores aritméticos en el caso de Orange3.

Glade[71] deriva una gramática de un corpus de programas de ejemplo. La gramática derivada se enumera para producir nuevos programas, aunque a diferencia de nuestro enfoque, no se aprende ninguna distribución sobre la gramática; la enumeración del programa es uniformemente aleatoria.

3.2.6. Programa de mutación

La prueba de entradas de módulo de equivalencia (EMI)[55][68] sigue un enfoque diferente para la generación de casos de prueba. Comenzando con el código existente, inserta o elimina instruc-

ciones que no se ejecutarán, por lo que la funcionalidad debe seguir siendo la misma. Si se ve afectado, se debe a un error del compilador. Si bien es una técnica poderosa capaz de encontrar errores difíciles de detectar, se basa en tener una gran cantidad de programas para mutar. Como tal, todavía requiere un generador de código externo. De manera similar a CSmith, EMI favorece los programas de prueba muy largos.

LangFuzz[45] también usa la mutación, pero lo hace insertando segmentos de código que previamente han expuesto errores. Esto aumenta las posibilidades de descubrir vulnerabilidades en los motores de lenguaje de scripting.

La enumeración de programas esqueléticos[88] funciona nuevamente al transformar el código existente. Identifica patrones algorítmicos en piezas cortas de código y enumera todas las posibles permutaciones del uso variable. Comparado con todo esto, su enfoque de fuzzing es de bajo costo, fácil de desarrollar, portátil, capaz de detectar una amplia gama de errores y enfocado por diseño a los errores que es más probable encontrar en un escenario de producción.

3.2.7. Aprendizaje automático

Existe un creciente interés en aplicar el aprendizaje automático a las pruebas de software. Más parecido a su trabajo es Learn & fuzz[76], en el cual una red LSTM se entrena a través de un conjunto de archivos PDF para generar entradas de prueba para el renderizador Microsoft Edge, produciendo un error. A diferencia de las pruebas de compilación, los casos de prueba de PDF no requieren entradas ni procesamiento previo del cuerpo de entrenamiento.

Skyfire[87] aprende una gramática probabilística sensible al contexto sobre un corpus de programas para generar semillas de entrada para pruebas de mutación. Se muestra que las semillas generadas mejoran la cobertura del código de AFL[115] cuando confunden los motores XSLT y XML, aunque las semillas no se usan directamente como casos de prueba. El aprendizaje automático también se ha aplicado a otras áreas, como la mejora de los analizadores estáticos de detección de errores [77][80] la reparación de programas[81][109], la priorización de los programas de prueba[73], la identificación de buffer overruns (sobrecargas de búfer)[75] y el procesamiento de informes de errores[79][59]. Según el conocimiento de los autores, ningún trabajo hasta el momento ha tenido éxito en la búsqueda de errores de compilación al explotar la sintaxis aprendida del código fuente extraído para la generación de casos de prueba. Aparentemente el trabajo de estos autores es el primero en hacerlo.

3.3. Serpent by Zeppelin (Jul 2017)

La empresa Zeppelin, crea herramientas para el desarrollo seguro, deployment y operación de sistemas descentralizados. También ayudan a compañías a securizar sus sistemas blockchain realizando auditorías.

La empresa Augur (un servicio de apuestas descentralizado) los contrató para realizarle una auditoría a Serpent, un compilador de un lenguaje Python-style que compila a EVM. El interés de Augur en esta auditoría está dado por razones propias, tuvieron un inconveniente de seguridad por utilizar este lenguaje. El código del proyecto se puede ver aún en su repositorio de GitHub[110].

> *"Hemos encontrado que el proyecto Serpent es de muy baja calidad. No se ha testeado, hay muy poca documentación y el diseño del lenguaje es muy defectuoso. La serpiente no debe considerarse segura de usar a menos que se solucionen muchos problemas críticos."*
> – Zeppelin Research team

En su publicación[248], la introducción a los resultados del reporte también es utilizada como conclusión al final del mismo. El contenido de ella es una la lista con los problemas apuntando directamente al documento original que posee detalle técnico.

El reporte original menciona que su estrategia fue analizar el código C++ del compilador, sumado a revisar documentación, ejemplos, y herramientas recomendadas para trabajar con los

contratos de Serpent. Realizaron diversos contratos de ejemplo minimales, para verificar y exponer los problemas que encontraron. Analizaron el código assembler tanto en LLL (Lisp Like Language, un lenguaje con un nivel de abstracción superior a la EVM, pero considerado assembler) como en la EVM, así como también su comportamiento en ejecución.

El análisis no explica metodologías ni tecnologías aplicadas a obtener a estos resultados. En base a todo lo observado el autor concluye que fue se trata de un proceso manual, poco automatizado, de un gran equipo. Sin embargo el acercamiento que la empresa tomó, a nivel proyecto, parece mucho más interesante que simplemente crear una herramienta, utilizar una ya existente sin explicar cómo utilizarla, o publicar los resultados sin establecer contacto con el equipo del producto analizado. Tal vez sea una diferencia destacable entre proyectos de grado y un contrato empresarial. Fué la primera auditoría que llamó la atención del autor en su momento, y mostró tanto la importancia como el estado inmaduro del mercado en este tipo de tecnologías.

Lamentablemente la auditoría fue realizada sobre un proyecto que aparenta haber sido descontinuado mientras se publicaba ese reporte. El reporte fue presentado en Julio del 2017, y las últimas actualizaciones a Serpent fueron realizadas meses antes. Actualmente se encuentra obsoleto y sin aplicar los cambios recomendados por el equipo.

3.4. Solidity by Coinspect (Nov 2017)

Nuevamente Augur contrató otro equipo de investigadores, esta vez a Coinspect, para realizar una auditoría al compilador de Solidity.

Coinspect, es un equipo reconocido por sus trabajos específicamente en el ambiente de la cyber seguridad. Entendiendo esto, no es extraño encontrar en la introducción de la publicación[252] un poco más de detalle[259] sobre lo que esperan encontrar o qué piensan en buscar relacionado en términos de seguridad.

El equipo de Coinspect analizó la herramienta Solidity con la posibilidad de encontrar fallas del siguiente estilo:

- Reducción de la seguridad de los contratos desplegados.
- Resultado en comportamiento no determinista.
- La ejecución de código malintencionado o que se bloquee al analizar el código fuente de un contrato en Solidity especialmente diseñado.
- Agotamiento de recursos durante la compilación, ya sea CPU, memoria o disco.
- Código compilado que consume una cantidad no constante de gas (por ejemplo, según los argumentos), donde el programador habría esperado un costo constante.
- Facilitando código malicioso (troyanos en código abierto).

También buscaron vulnerabilidades comunes a vectores en aplicaciones de software:

- Validación de entrada.
- Prevención de denegación de servicio (DoS).
- Prevención de la fuerza bruta.
- Divulgación de información.
- Vulnerabilidades de corrupción de memoria: buffer overflows, format strings por el usuario.
- Integer overflows/underflows.
- Vulnerabilidades de gestión de punteros: double free, user after free.

En cuanto a los resultados, exceptuando uno sólo de los problemas reportados que parece haber sido hallado mediante un proceso manual, el autor se atrevería a decir que el resto pertenecen a una clásica familia resultante de aplicar técnicas de fuzzing.

La diferencia principal que se podría hacer, sobre los análisis de Serpent y Solidity, es que Coinspect se enfoca principalmente en el hallazgo de vulnerabilidades y Zeppelin a problemas de diseño e implementación, los cuales a su vez pueden tener implicancias de seguridad.

Hay que tener en cuenta que que se hace imposible comparar con objetividad los procesos de cada equipo, dado que están meramente basadas en los resultados de cada investigación, y no se están teniendo en cuenta las herramientas disponibles, ni los recursos económicos o el tiempo que poseyó cada equipo al momento de realizarlas.

3.5. Solidity by Ethereum Foundation

Para situarse en contexto, el compilador de Solidity junto con su lenguaje fueron impulsados por la Ethereum Foundation. Su código está disponible en GitHub[256] desde sus comienzos, donde realizan releases periódicamente, tienen su documentación en ReadTheDocs[254], y una amplia comunidad que colabora con el proyecto.

Si bien al momento de esta investigación no parecen aplicar seguridad a su SDLC, en su repositorio dicen realizar testing y fuzzing antes de cada release. Adicionalmente en su documentación se poseen categorías útiles con la intención de reducir las fallas en el código. Algunos ejemplos son patrones de diseño, código de estilo, consideraciones de seguridad, reproducción de testing, y una explicación de cómo utilizar el fuzzer que está incluido en el proyecto.

Varios de los bugs reportados por el equipo de Coinspect siguen sin estar corregidos, y no es posible distinguir en el sistema de reportes de tickets de github, qué bugs corresponden a su suite de testing, y de qué manera se han encontrado. Tal vez muchos errores no llegan a ser publicados mediante commits porque son testeados localmente, y esa información no se encuentra visible al público. El sitio solfuzz.ethdevops.io[253] solía dar estadísticas de los resultados encontrados por el fuzzer incorporado pero en el momento de este análisis se comprobó que ha dejado de funcionar hace tiempo.

Podrá no ser una solución similar a las demás, ya que no se dedican específicamente a aplicar seguridad o auditar el proyecto. Pero todas las soluciones apuntan a lo mismo, securizar los smart contracts através del compilador. Ésta es una tarea que principalmente el equipo de Solidity debería liderar ya que es su responsabilidad tomar los recaudos necesarios a través de su ciclo de desarrollo para proveer un servicio seguro.

3.6. Discusión sobre las soluciones presentadas

Mientras más eficiente sea la herramienta para analizar un compilador, más acoplada a su gramática deberá ser, y como consecuencia serán menos abarcativas. La adaptación de estas herramientas para poder migrar entre lenguajes es sumamente costosa.

En el caso de DeepSmith, los autores de la herramienta lograron aplicar su funcionalidad partiendo desde OpenCL hasta Solidity sin demasiado esfuerzo. Sin embargo, los testeos sobre los compiladores de Solidity fueron de los más breves y menos explorados que realizaron. No se buscaron impactos reales ni se hizo un chequeo sustancial sobre los hallazgos.

La problemática que se trata de presentar no es que el fuzzing no funciona, sino que no forman parte directa de realizar una auditoría. Se presentan como herramientas, analizan los proyectos, en este caso compiladores, y dan por finalizada la investigación. Su propósito no es encontrar y analizar las vulnerabilidades, sino proveer una herramienta para hacerlo.

Las dos propuestas que poseen un valor destacable, son las realizadas por las dos empresas previamente mencionadas, Coinspect y Zeppelin. Aún así son un paso accesorio a las herramientas presentadas, sólo que la herramienta en cuestión es un equipo en vez de un sólo software.

Las sugerencias incorporadas al reporte de Serpent por parte del equipo de Zeppelin fueron de gran motivación para realizar este proyecto.

Al momento de la redacción de este documento, no se encuentra disponible una herramienta para analizar el código de compilador y entender cómo podría influir al lenguaje que ésta interpreta. En el caso de Solidity tampoco hay una gramática oficial definida de manera reutilizable, ya que el scanner no está basado en una gramática estándar, sino que está diseñada su lógica dentro del mismo parser. Teniendo en cuenta lo descrito en el capítulo anterior, se puede concluir que las alternativas provistas e históricas en el estado del arte no son soluciones suficientes o aplicables para concluir un análisis completo, o mejor dicho una auditoría completa, al proyecto en cuestión, Solidity.

Capítulo 4

Método propuesto

TODO:

- Faltan todos los links a las herramientas.
- Falta formatting.
- Agregar gráfico en metodología
- Eliminar este TODO al finalizar.

4.1. Introducción

El objetivo de la metodología descripta en este capítulo consiste en revisar el compilador y lenguaje de Solidity, analizar su diseño general y arquitectura, y reportar potenciales vulnerabilidades de seguridad que puedan llegar a comprometer el código compilado. En este capítulo se describe el trabajo realizado presentando ejemplos de observaciones en áreas específicas del código que presentan problemas concretos, así como también observaciones generales que atraviesan el proyecto entero, que puede mejorar su calidad como un todo.

Se puede interpretar a la metodología como una auditoría abarcativa, no sólo de seguridad, sino en todos los aspectos que permitan encontrar problemas. No es menor resaltar que cualquier problema en este contexto puede ser considerado un potencial impacto de seguridad.

El método propuesto no tiene intención de destacarse por utilizar características novedosas a nivel tecnológico o introducir nuevos procesos. De hecho todo lo que se realizará en esta sección, no es más que, en distintas medidas, una conjunción de los dos capítulos anteriores, y es por eso que parece importante hacer énfasis en este tipo de propuestas, ya que están al alcance del estado del arte.

La razón de centrar el foco de investigación en un proyecto open source, que posee instrucciones sobre cómo construirlo, y una documentación extensa, es para presentar una situación favorable y menos limitante para realizar una auditoría. No obstante, poseer esta situación lo hace más complejo y desafiante, ya que se puede analizar desde todas las perspectivas posibles.

Su código está disponible online, y se ha decidido congelar el repositorio en la última versión estable a la hora de realizar esta investigación.

4.2. Alcance de la auditoría

El código auditado puede encontrarse en su repositorio público de GitHub ethereum/solidity[256], y la versión utilizada para el análisis se encuentra en el *tag* v0.4.24[255], commit: `e67f0147998a9e3835ed3ce8bf6a0a0c634216c5`.

La auditoría contendrá un profundo análisis, diseño e implementación de la herramienta, apuntando a asesorar su calidad e investigar problemas potenciales que pueden surgir mediante su utilización.

Estas actividades incluyen revisar las etapas de *parsing*, análisis, optimización, código de generación. La auditoría cubrirá el código ensamblador generado para la *EVM*, *ABIs* (Application Binary Interface), y Solidity con código intermedio *Yul* de manera *inline*.

No se cubrirá la generación de código *eWASM* (Ethereum Flavored WebAssembly), las características relacionadas al lenguaje *LLL* (lenguaje de bajo nivel para la *EVM* con una sintaxis de expresiones-s), las características experimentales, y las herramientas de verificación formal incluídas en el proyecto.

El proyecto no parece integrar la seguridad como componente activo en su ciclo de desarrollo. Incluye un componente para aplicar técnicas de *fuzzing* y una suite de testeo. Estos no serán analizados con detenimiento pero de todos modos se verá cuál es su real interacción con el compilador.

4.3. Metodología

Para estos casos, siempre es mejor realizar una revisión manual de código, y como tal se debe comenzar con un modelado de amenazas o al menos entrevistas a los desarrolladores para tener un entendimiento de la arquitectura de la aplicación, su superficie de ataque, así como también las técnicas de implementación.

Es por eso que durante toda la auditoría se mantuvo siempre al menos un canal de comunicación abierto con el equipo encargado del proyecto. Se utilizaron servicios de mensajería[ref gitter] para mantener un diálogo específicamente para detalles asíncronos sobre dudas puntuales, frecuentemente relacionadas a la información presentada en el repositorio de GitHub.

GitHub como herramienta fue utilizada en el proyecto como medio de intercambio de información de una manera más acoplada al proyecto, teniendo la posibilidad de continuar involucrando a la comunidad activa.

Finalmente para partes del código difíciles de comprender, se realizaron walkthroughs (guías destinadas a mejorar el entendimiento del proyecto) mediante videollamadas en una herramienta de videoconferencia[ref Meet].

Se utilizó un conjunto de las estrategias de auditoría de código con propósitos bien definidos. Por ejemplo, la generalización de diseño aplicada a analizar la arquitectura del compilador; puntos candidatos para posibles problemas en las etapas de parsing y optimización; y compresión de código basados en hallazgos mediante técnicas de black box o testing manual.

A grandes rasgos, se pretendió:

- Cubrir un análisis general del proyecto en todos los aspectos del mismo, desde cómo está conformado el equipo que lo mantiene hasta las tecnologías que se utilizan.
- Realizar revisión de código en los puntos candidatos y en los lugares en donde se han producido fallos en las pruebas de Black Box.
- Auditar las distintas etapas del compilador a nivel general, buscando que la arquitectura del compilador contemple las etapas estándar.
- Revisar el diseño del lenguaje y su instrumentación para verificar que la instrumentación sea consistente con la especificación del diseño del lenguaje.
- Revisar la configuración de testeo para detectar posibles fallos en los códigos de testeo del propio proyecto.
- Aplicar técnicas de black box para identificar el lugar del fallo y el lugar y la magnitud del impacto del mismo.

- Analizar la salud del proyecto para hallar posibles defectos como compatibilidad hacia atrás, aspectos no contemplados, información inconsistente y sin tratamiento en los repositorios, entre otros.
- Revisión de problemas previamente reportados para asegurar el tratamiento y seguimiento de temas ya identificados.
- Análisis de la documentación para verificar la consistencia entre la documentación y los aspectos de sintaxis, semántica y pragmática del compilador.

Finalizar una auditoría no consta únicamente de entregar lo hallado de manera descriptiva. Se cumple un rol muy importante a la hora de presentar los resultados, ya que sin alternativas, soluciones o sin mantener un contacto para ayudar a remediar los problemas reportados estaría faltando una parte muy importante del trabajo. Frente a los hallazgos se proponen también recomendaciones a aplicar para mejorar la calidad del proyecto.

4.4. Herramientas par el análisis

El método principal de análisis comienza observando y comprendiendo el código (escrito en el lenguaje C ++, versión XXX) del compilador en profundidad. Sin embargo, también se utiliza una serie de herramientas automatizadas que ayudan el estudio en varios aspectos complementarios. Estos se estructuran en 3 categorías principales: (1) visualización de código, (2) análisis estático y (3) prueba / fuzzing. Estas categorías se aplican tanto para el código C ++ del compilador como para la salida EVM que genera.

4.4.0.1. Herramientas C++ (Código del compilador)

Debido a que C ++ es un lenguaje establecido que contiene cierta madurez, existe una gran cantidad de herramientas disponibles. Las herramientas de visualización nos ayudan a comprender el código del compilador, así como a navegarlo. Las herramientas de análisis estático nos ayudan a detectar las dificultades semánticas comunes dentro de él, y las herramientas de fuzzing estresan al compilador con un gran conjunto de pruebas generadas automáticamente.

Visualización de código (C++)

SourceTrail Un explorador de código para C++ y Java. Extremadamente útil para la rápida navegación visual de una gran base de código.

Visual Paradigm Una herramienta para generación de gráficos UML. Permite generar distintos diagramas, como el de clases, basándose en el código de un proyecto.

Ctags Indexador de código. Comúnmente utilizado junto con el IDE para facilitar la navegación del proyecto.

Cscope Navegador de código de fuente. Es similar a Ctags pero permite ir un poco "más allá", y funciona mejor en proyectos grandes donde su código no es conocido.

Clang-tidy Linter basado en Clang para C++ apuntado a diagnosticar y señalar típicos errores de programación. Incluye chequeos para las librerías de boost también.

CppCheck Herramienta de análisis estático para C++ apuntado a detectar errores reales (mínimos falsos positivos).

CppDepend Potente análisis estático destinado a mejorar la calidad del código.

Flawfinder Herramienta para detectar posibles problemas de seguridad en código C++.

CppLint Es una herramienta de línea de comandos para verificar archivos C / C ++ en busca de problemas de estilo siguiendo la guía de estilo C++ de Google.

Scan-build Es una utilidad de línea de comando que permite a un usuario ejecutar un analizador estático sobre su base de código como parte de la realizar una compilación regular.

Lizard Lizard es un analizador de complejidad ciclomática extensible para muchos lenguajes de programación, incluido C/C++ (no requiere todos los archivos de encabezado ni las importaciones de Java). También realiza detección de copy & paste (detección de clon de código / detección de duplicado de código) y muchas otras formas de análisis de código estático.

Prueba / fuzzing (C++)

Grammarinator Genera pruebas aleatorias de acuerdo con una definición de gramática ANTLR. La gramática Solidity.g4 creada por Federico Bond permite un acercamiento menos laborioso para integrar la gramática oficial.

American Fuzzy Lop AFL es un fuzzer orientado a la seguridad que emplea un nuevo tipo de instrumentación en tiempo de compilación y algoritmos genéticos para descubrir automáticamente casos de prueba limpios e interesantes que desencadenan nuevos estados internos en el binario objetivo. Esto mejora sustancialmente la cobertura funcional para el código fuzzeado.

LibFuzzer Es un motor evolutivo de fuzzing, in-process, guiado por la cobertura del proyecto. Se enlaza con la biblioteca o el proyecto bajo testeo, y se le alimenta entradas fuzzeadas mediante un punto de entrada; el fuzzer luego localiza qué áreas del código son alcanzadas, y genera mutaciones a partir de los datos de entrada para maximizar la covertura del código. La información de cobertura del código para libFuzzer es proporcionada por la instrumentación SanitizerCoverage de LLVM.

Instrumentaciones de compilación de Clang Flags de compilación para instrumentar en compilación al binario. AddressSanitizer (detector de error de memoria rápida), MemorySanitizer (detector de lecturas de variables no inicializadas), Fuzzer (alternativa para utilizar libFuzzer), SanitizerCoverage (cobertura de código).

Debugging: gdb/lldb (for debugging) Herramientas de debugging para realizar análisis en ejecución. Se pueden utilizar como línea de comandos, o integrados al IDE.

Dhex Editor hexadecimal por línea de comandos.

Honggfuzz Fuzzer orientado a la seguridad con potentes opciones de análisis. Admite fuzzing evolutivo basado en retroalimentación basado en la cobertura de código (basado en software y hardware).

4.4.0.2. Herramientas EVM (Código generado por el compilador)

El ecosistema Ethereum ya está produciendo una serie de herramientas que son potencialmente útiles para este análisis. Las herramientas de visualización ayudan a comprender el resultado del compilador, que es bastante bajo y difícil de leer de forma natural. Dicha visualización es particularmente importante para evaluar las optimizaciones del código de operación EVM. El análisis estático ayuda a identificar problemas de flujo de control que pueden impactar la EVM, y las herramientas de fuzzing nos permiten probar bytecode fuera de la interfaz ABI ampliamente utilizada actualmente cubierta en muchos frameworks.

Visualización de código (EVM)

Solplay Visualizador de Solidity a varias salidas del compilador, incluyendo procesamiento post-bytecode por otras herramientas. Creada por Zeppelin particularmente para esta auditoría, intencionada para acelerar el uso de otras herramientas de visualización.

Solmap Bytecode visualizado en tiempo real, con la habilidad de seleccionar conjuntos de instrucciones y ver el código en Solidity asociado a ellos. Utiliza la información de mapeo de código de Solidity. También construida por Zeppelin para esta auditoría.

Remix Solidity IDE web. Útil para debuggear salidas del compilador.

Evmdis Disassembler para la EVM que agrupa opcodes en expresiones más legibles. Útil para visualizaciones pero no es 100 % preciso.

go-ethereum/evm Utilidad para desarrolladores de la EVM. Excellente para debuggear la ejecución de la EVM a un bajo nivel.

Evm-tools Herramientas para la ejecución y el desensamblado de la EVM. Tools for EVM execution. Desactualizado, pero útil principalmente con fines educativos. No es lo suficientemente confiable.

Análisis estático (EVM)
Fuera del alcance de la investigación.

Prueba / fuzzing (EVM)

Web3 Ethereum Javascript API.

Geth Implementación en Go de un nodo de ethereum, utilizado para simular transacciones y correr testeos.

Cpp-ethereum Implementación de nodo en Ethereum en C++.

4.5. Formato de los resultados

La manera más oportuna y organizada de presentar los resultados, dada la cantidad y la complejidad que poseen, se concluyó que debería ser compuesta de un título descriptivo, un breve desarrollo de la situación en la que se da, un ejemplo o forma de replicación, recomendaciones para remediar o mejorar esa situación, y para finalizar un breve comentario con el estado actual de dicha cuestión en el proyecto.

El estado actual no es más que una sección comentando la respuesta por parte de los líderes del proyecto, y el estado en el que se encuentra a la hora de escribir este trabajo de investigación, que difiere notablemente de la fecha de los hallazgos. Se incluirán links a issues y pull requests de GitHub en los casos en los que existan referencias directas con el proyecto.

Capítulo 5

Resultados

TODO:

- Agregar referencia al apéndice en Coinspect audit.

La investigación se situó como parte de un trabajo realizado como investigador independiente bajo contrato con la empresa Zeppelin Solutions. Se integró el equipo de auditoría con intención de aplicar perspectivas de seguridad.

Los resultados mostrados han sido extraídos, traducidos y adaptados del reporte original que se publicó en conjunto y puede ser observado en el reporte oficial[251]. Asimismo, la distancia temporal que se tomó para permitirle al equipo encargado del proyecto de Solidity responder a cada descubrimiento, posibilitó agregar un seguimiento, el cual se podrá apreciar debajo de cada situación descripta, como nota de **Actualidad**.

Tal como se comentó en el capítulo 4.5, se prefirió presentar los resultados en este capítulo siguiendo un orden más cercano a objetivo de la auditoría que respecto de la metodología aplicada. Si bien no se muestra explícitamente en cuáles de los resultados se aplicaron las técnicas mencionadas en los puntos descritos en la sección 4 del capítulo 4, las mismas se utilizaron implícitamente y en conjunto para producir los resultados de la auditoría.

Se evitó extender esta sección en demasía, teniendo en cuenta que el detalle entero se puede observar para detalles técnicos en profundidad en el reporte técnico previamente publicado. Se hará hincapié en los hallazgos más interesantes, y en los que estén solapados directamente con perspectivas de seguridad.

5.1. Hallazgos por categoría

5.1.1. Problemas de contexto general

5.1.1.1. Se pueden inyectar direcciones inválidas de bibliotecas en la etapa de linkeo

Un contrato que depende de una biblioteca con funciones públicas tendrá la dirección de la instancia desplegada de la librería en su bytecode. Dado que esta dirección no es conocida por el compilador, tiene que ser provista por el usuario: esto se hace mediante la opción `--libraries` en **solc**. Sin embargo, es imposible no proveer esas direcciones, y dejar el contrato sin estar linkeado; es por eso que una secuencia especial de caracteres, conteniendo el nombre de la biblioteca va a estar presente en el bytecode, previniendo que el contrato sea desplegado sin que la dependencia sea resuelta, utilizando el modo `--link` de **solc**, en el cual el enlace es realizado.

El problema surge en la representación de este `string`: es truncado a 36 caracteres, y los caracteres remanentes del nombre de la librería son eliminados, sin advertencias. Es posible, entonces, tener múltiples librerías que compartan un nombre truncado, haciendo que compartan un prefijo suficientemente largo. Considerar el ejemplo a continuación:

```
library OpenZeppelinStdLibraryArray {
...
}

library OpenZeppelinStdLibraryArrayUtils {
...
}

contract Test {
    function test() public pure returns (uint256) {
        if (OpenZeppelinStdLibraryArrayUtils.isArrayEmpty(arr)) {
        return OpenZeppelinStdLibraryArray.getArrayLength(arr);
        } else {
        return 0;
        }
    }
}
```

El *bytecode* generado al llamar `solc Test.sol --bin` incluirá dos instancias de la secuencia `'__Test.sol:OpenZeppelinStdLibraryArray__'` (la representación truncada del string del nombre de la librería). Cuando se llama a `solc --link --libraries` con direcciones para ambas librerías, el compilador identificará ambas instancias como referencias repetidas `OpenZeppelinStdLibraryArray`, y reemplazará ambas de ellas con esa dirección, ignorando por completo la dirección provista para la otra librería.

Este problema está compuesto de otro, el cual causa a los nombres de bibliotecas truncados que sólo requieran un *matching* de prefijo, incrementando la superficie de ataque. Considerar las siguientes librerías:

```
library OpenZeppelinStandardLibraryArrayCore {
  ...
}

library OpenZeppelinStandardLibraryArrayUtils {
  ...
}
```

El nombre de librería trucado es `OpenZeppelinStandardLibrary`, pero pasando cualquier string que comience con esa secuencia al comando `--libraries` también hará que a ambas bibliotecas se le asignen la misma dirección, a pesar de el hecho de que ninguna biblioteca ni un marcador de posición (placeholder) en el bytecode se asemeje a ese string, por ejemplo: `solc Test.sol --bin | solc --link --libraries ‘‘OpenZeppelinStdLibraryArrayCollection:<address>"`.

Se recomendó considerar remover el tamaño máximo de nombre de librería por completo, o rediseñar la implementación para que permite utilizar librerías con nombres largos.

Actualidad: Existen *issues* abiertos previos al respecto, **#579**[191], **#3918**[204] y **#4429**[186]. El equipo de Solidity respondió “Esta función no está aconsejada, los usuarios deben usar el json estándar que genera las referencias de enlace. Esto es sólo un problema en la interfaz en desuso del compilador”. Solucionado en **PR#5145**[231].

5.1.1.2. Llamadas inseguras al sistema pueden llevar a una ejecución de comandos durante la etapa de testeo.

Comandos no sanitizados dados como argumentos para llamar a `system()` (o funciones similares como `popen()`) pueden permitir a un atacante ejecutar comandos de sistema arbitrarios.

El siguiente código de **isoltest** abre un editor del sistema al encontrar un error mientras testea contratos en Solidity: `/test/tools/isoltest.cpp:236`

```
if (system((editor + " \"" + m_path.string() + "\"").c_str()))
  cerr << ‘‘Error running editor command." << endl << endl;
  return Request::Rerun;
```

El problema con el código anterior es que la llamada al sistema está hecha sobre una combinación de dos variables, y una de ellas no está propiamente sanitizada. **m_path** es el path para el contrato en cuestión pero **editor** proviene de una variable de entorno, la cual controlada por un atacante puede resultar en una ejecución de comandos.

Como un simple ejemplo de cómo esto puede ser explotado, imaginemos el caso en el que un atacante haya accedido de alguna manera a manipular el contenido de la variable de entorno `EDITOR`, y lo haya modificado de la siguiente manera:

```
1    EDITOR='wget http://attacker.site/expl;chmod +x expl;./expl; vim'
```

Listing 5.1: bash version

Cuando un error sea encontrado en algún contrato, un aviso preguntará si se quiere *editar, actualizar expectativas, saltear o salir*. En caso de elegir editar, el comportamiento normal sería que se abra el editor por defecto del sistema o uno especificado manualmente mediante `--editor`.

Teniendo la variable de entorno bajo control, esto resultaría en la ejecución del comando insertado, descargando mediante `wget` un ***exploit***, dándole permisos de ejecución, ejecutándose, y continuando con la edición del testeo utilizando el editor `vim` para evitar sospechas.

Notar que utilizar la opción `--editor` posee el mismo problema.

Se recomendó considerar utilizar un abordaje diferente que no utilice un intérprete directo. Por ejemplo **execv** o **execve**, que trabajan de manera diferente, realizando una bifurcación a nuevos procesos (*fork* de ahora en adelante) y creando el *command string* de una manera que elimina preocupaciones sobre *buffer overflows* o *string truncation*. Más información aquí[149].

Actualidad: El equipo de Solidity respondió: “Cabe señalar que esto no es parte del código de producción. Es solo una parte de la infraestructura de prueba. Queremos ejecutar el editor, por lo que esto siempre resultará en la ejecución del código. Una contramedida sería verificar si ‘EDITOR‘ es la ruta directa de un archivo ejecutable y también imprimir el archivo antes de que se le pregunte al usuario qué hacer con respecto a la falla. Además, si un atacante tiene control sobre las variables de entorno en una máquina de compilación, no hay absolutamente ninguna manera de protegerse contra tal ataque. Una variable de entorno que casi con certeza conducirá a un exploit es, por ejemplo, “CC”: el compilador de C y probablemente haya toneladas más”. La discusión continúa en el issue **#5159**[171].

5.1.1.3. Manejo inseguro de strings en la infraestructura de testeo.

strcpy no chequea por *buffer overflows* cuando copia a destino, y es por esto, que su uso es considerado peligroso para muchos[246] (a pesar de que estos chequeos se puedan realizar manualmente).

`test/RPCSession.cpp:63`

```
1  if (_path.length() >= sizeof(sockaddr_un::sun_path))
2      BOOST_FAIL(``Error opening IPC: socket path is too long!");
3
4  struct sockaddr_un saun;
5  memset(&saun, 0, sizeof(sockaddr_un));
6  saun.sun_family = AF_UNIX;
7  strcpy(saun.sun_path, _path.c_str());
```

Nota: El código actual no parece ser vulnerable, pero el uso de strcpy es fuertemente desalentado cuando alternativas seguras se encuentran disponibles.

Se recomendó considerar utilizar **strcpy_s**, ó **strlcpy** en su reemplazo.

Actualidad: El uso de *strcpy* sigue vigente al menos en esta parte del código.

5.1.2. Auditorías previas

En esta sección se analizaron problemas reportados por auditorías anteriores, observando su estado durante la investigación mediante reproducción en casos de test.

5.1.2.1. La auditoría llevada a cabo por Coinspect posee issues desatendidos.

En el 2017, Coinspect realizó una auditoría[252] al código fuente del compilador de Solidity. La auditoría anterior reveló diez problemas, de los cuales a la hora de realizar la investigación tres permanecieron desatendidos (SOL-005, SOL-010), con la excepción de uno que fue solucionado en simultáneo (SOL-007). La mayoría de ellos fueron tratados como advertencias hasta la versión v0.4.24, y en la versión v0.5.0 se interpretan como errores.

La descripción de los problemas son directas, con un seguimiento a su estado actual y código Solidity utilizado para re-confirmar las declaraciones. Ver Apéndice A[referencia] para leer los detalles de cada problema reportado.

Se recomendó considerar la implementación de correcciones lo más pronto posible, particularmente para problemas que han sido compartidos públicamente.

Actualidad: Comentario del equipo de Solidity: "Los dos restantes requieren la eliminación de funciones del idioma programado para la versión 0.6.0". SOL-005 fue solucionado en **#3324**.

5.1.3. Salud de proyecto

Una verificación del estado de salud del proyecto permite al equipo dar un paso atrás en la ejecución diaria de tareas para evaluar el estado real del proyecto de una manera objetiva. Los beneficios de realizar una verificación de salud del proyecto incluyen: Identificar los problemas antes de que ocurran, lo que puede ahorrar mucho tiempo y dinero.

5.1.3.1. Sólo se emiten advertencias para problemas conocidos de retrocompatibilidad.

Solidity hace lo mejor para preservar la retro compatibilidad al no presentar cambios disruptivos en releases menores, mediante la emisión de notas de deprecación en mensajes de advertencia y sugiriendo cambios cuando detecta problemas potenciales en el código. Este es el procedimiento standard para la mayoría de los proyectos.

Sin embargo, Solidity no es un proyecto standard de software: el código generado por él corre *smart contracts*, los cuales sin inmutables y sus transacciones irreversibles. Medidas deberían ser tomadas para asesorar la seguridad del código de usuario de la manera más sencilla posible. Mientras que plataformas tales como **Etherscan** permiten la verificación del código de un contrato, no emiten las advertencias que fueron expedidas durante la compilación, forzando al usuarios a 1) recordar todos las cuestiones conocidas y chequear que ninguna de ellas esté presente, o 2) compilar el contrato localmente, incrementando significativamente la barrera de entrada para un desarrollador que está leyendo el código. **Etherscan** podría mostrar estas advertencias, pero sería mejor si esta responsabilidad no es transferida, y que dicho código no sea permitido en primer lugar.

Considerar un contrato verificado que visualmente asimila ser inocente[136]. Un problema conocido como referencias sin inicializar, o setear un almacenamiento por defecto, causan en el contrato que una llamada a `applyRaises` también modifique el *owner*(término para referirse a quién controla el contrato) del contrato como un efecto secundario (como puede ser visto en el historial de transacciones). No se muestran advertencias, ni se encontraría en la mayoría de los testeos (dado que la mayoría de los testeos corren de manera independiente por diseño, y generalmente no se chequearía que el owner no haya cambiado después de una llamada a una función de este estilo).

Se recomendó realizar cambios disruptivos (*breaking changes*) al corregir este tipo de errores (almacenamiento no inicializado, funciones de construcción sin la palabra reservada `constructor`,

etc.).

Actualidad: La versión v0.5.0 introdujo breaking changes para estas cuestiones.

5.1.3.2. El bus factor es de 2.

Se conoce como *bus factor* a "la mínima cantidad de miembros de un equipo que tienen que desaparecer repentinamente de un proyecto para que el proyecto colapse debido a la falta de personal competente o con entendimiento."(traducido de Wikipedia[120]). Un factor de bus bajo expone al proyecto a muchos riesgos y hace que el desarrollo sea más lento, mientras que un factor de bus más alto muestra una comunidad más acogedora que difunde el conocimiento y ayuda a los nuevos miembros a asumir responsabilidades y sentirse parte del proyecto. Con solo dos mantenedores activos[139], el factor bus de Solidity es muy bajo. Se requiere un factor de bus más alto para la sostenibilidad a largo plazo del proyecto.

Se recomendó considerar la posibilidad de asesorar a algunos de los contribuidores frecuentes actuales para ayudarlos a convertirse en mantenedores. Ellos pueden, a su vez, ayudar a obtener más colaboradores al documentar sus aprendizajes, reportar problemas para las partes del proceso que son demasiado complicados y difundir la información sobre las buenas maneras de involucrarse.

Actualidad: Desde marzo de 2018[138], al menos dos contribuyentes muy activos han estado aportando constantemente en varias áreas del proyecto. Están en un buen camino para unirse al equipo de mantenedores pronto.

5.1.4. Detalles en torno a la documentación

5.1.4.1. Archivo README para las optimizaciones de Yul incompleto.

Hay un archivo README[206] para el optimizador de *Yul*, que incluye información muy útil, pero está incompleto: algunas etapas de optimización no se explican y algunas secciones están vacías o son demasiado escasas.

Considere reorganizar este documento para explicar primero la arquitectura del optimizador, y luego las diferentes etapas y sus efectos. Una alternativa sería eliminar este archivo README y documentar todo como comentarios sobre el código fuente.

Actualidad: el equipo de Solidity respondió: "El componente aún se encuentra en la fase de investigación y, por lo tanto, no merece la pena documentarlo en esta etapa. Una vez que sea parte del código activo, estará completamente documentado". Actualmente en v0.5.13 se encuentra un documento[268] mucho más detallado.

5.1.4.2. Falta información para poder utilizar el fuzzer contenido en el proyecto.

Algunas distribuciones de Linux fallarán siguiendo los pasos descritos para construir el fuzzer y no hay sección que permita solventar los problemas encontrados.

Se recomendó considerar la elaboración de estas instrucciones, realizar pruebas en diferentes plataformas y especificar soluciones alternativas para compilarlas para cada distribución.

Actualidad: un *pull request* **PR#4360**[226] con información adicional ha sido incorporado.

5.1.5. Situaciones en torno al testeo del proyecto

5.1.5.1. No se encuentra reporte de cobertura de código.

El código sin pruebas unitarias puede tener pequeños errores que son difíciles de detectar en las revisiones de código, que pueden causar vulnerabilidades de seguridad y errores de funcionalidad.

Si no hay un informe sobre la cobertura del *unit test* (test unitario), se desconoce la minuciosidad de la prueba, lo que dificulta encontrar las secciones del código que necesitan atención adicional. Además, cuando se propone un *pull request*, es difícil identificar si todas las rutas de código posibles están cubiertas por pruebas automatizadas.

Se recomendó considerar la posibilidad de generar un informe de *unit test coverage*(test unitario de covertura) para comprender mejor el estado actual de la base de código y automatizar la generación de dicho informe para los *pull requests* y así garantizar que todos los cambios incorporados tengan todas las rutas de código posibles cubiertas.

Actualización: Este *issue*[168] ya ha sido resuelto.

5.1.5.2. Bajo nivel de unit test coverage.

A partir del 26 de Septiembre, la cobertura de pruebas reportada del branch develop[127] fue del 87,91 %.

Este informe de cobertura es para la ejecución combinada de pruebas unitarias con algunas pruebas de integración que cubren varias unidades de código al mismo tiempo. Por lo tanto, el porcentaje de cobertura parece alto, pero en realidad no se posee una visión clara de cuántas afirmaciones se verificaron y determinaron con pruebas unitarias.

Se recomendó considerar medir solo la cobertura de prueba unitaria y aumentarla al menos al 95 %. La cobertura de las pruebas de alto nivel se pueden analizar de manera diferente a través de *user stories* o una lista de verificación de características del lenguaje.

Además, se recomendó considerar la refacción de las pruebas unitarias para asegurarse de que están llamando a una sola función pública, ejerciendo una única rama de código y que terminen de afirmar el comportamiento esperado de esa rama en particular. De esta manera, se puede vincular con confianza la cobertura de la prueba unitaria al número de afirmaciones que se comportan como se espera, con los beneficios adicionales de que las pruebas servirán como una documentación clara de lo que el compilador debe hacer en cada caso, y que seguirá un diseño totalmente comprobable y determinista.

Actualidad: En el momento de esta redacción, aún la mayoría de las secciones del proyecto se encuentran debajo del 95 %, pero a comparación con la primera medición todos los porcentajes fueron en incremento superando aproximadamente el 85 %[128] para todos los casos.

5.1.5.3. No hay una estructura clara de testing.

Aunque el proyecto tiene un número significativo de pruebas que cubren diferentes áreas de la Pirámide de Pruebas, no está claro qué pruebas pertenecen a qué área de la estructura, qué áreas están cubiertas y qué parte de cada área está cubierta.

La base de la pirámide se aborda claramente con pruebas unitarias en los elementos más granulares del código del compilador, pero no hay información de cobertura, como se aborda en otra parte de la auditoría.

El siguiente nivel de la pirámide consiste en compilar un conjunto de contratos conocidos y ejecutarlos contra el cliente `cpp-ethereum` con diferentes versiones de EVM. No hay pruebas de rendimiento, no hay pruebas de uso de gas, no hay pruebas de revestimiento que aborden el estilo del código, no hay pruebas de estrés.

Se recomendó considerar el diseño y la documentación de una estructura piramidal clara para el conjunto de pruebas del proyecto. Con una estructura de este tipo en su lugar, agregar capas a la pirámide y obtener el control de la cobertura de cada nivel debería ser un proceso sistemático y

progresivo.

Actualidad: Se crearon los issues **#5165**[169] y **#5252**[170], los cuales siguen sin resolverse bajo la categoría de testing.

5.1.5.4. No hay pruebas estáticas que impongan un estilo de código consistente.

Un estilo de código coherente es esencial para que el código base sea claro y legible, y para que sea posible combinar contribuciones de personas muy diversas, como es el caso en proyectos open source.

Considere hacer que todos los archivos del proyecto sigan la guía de estilo de código documentada[250] e imponer que cada contribución nueva se adhiera a este estilo de código agregando una comprobación con *linters* que se ejecuten en cada *pull request*.

Actualidad: El equipo de Solidity respondió: *"Hemos comenzado a agregar algunas comprobaciones. El problema principal es que las personas externas no entienden cuándo o por qué fallan las pruebas de estilo de código, tenemos que hacerlo más visible"*. La discusión continúa en el issue **#5241**[**197**].

5.1.5.5. Es muy difícil realizar pruebas localmente.

A pesar de que la sección de "Ejecución de las pruebas del compilador"[137] es muy detallada y clara, la ruta "feliz"para ejecutar todas las pruebas con éxito es muy frágil y es casi imposible de lograr en muchos sistemas operativos / distribuciones comunes (Manjaro, Archlinux, Mint 18 Sarah, Ubuntu Xenial y Bionic, etc). El script de prueba a menudo demora indefinidamente de manera silenciosa en las pruebas de `cpp-ethereum`, y no está claro cuándo se completó con éxito el conjunto de pruebas o si algo salió mal.

Las pruebas no funcionan con ninguna versión de `cpp-ethereum` y la documentación se vincula a un binario específico de `cpp-ethereum` sin ninguna explicación particular de por qué se requiere esta versión.

Estos problemas pueden disuadir a alguien que desea contribuir al proyecto y verificar los cambios a nivel local antes de enviarlos para el análisis de CI (integración contínua).

Se recomendó considerar los siguientes puntos para mejorar la experiencia del desarrollador con respecto a las pruebas:

- Asegurar que la guía "ejecución de las pruebas del compilador"funcione en todas las plataformas compatibles y establezca cuáles son las compatibles.
- Definir claramente qué salida se espera para una ejecución exitosa del 100 % del conjunto de pruebas.
- Proporcionar más información sobre la versión particular de `cpp-ethereum` requerida para las pruebas.

Actualidad: *issue* para el seguimiento de cambios en la documentación **#5166**[190]. Actualmente es un trabajo en proceso.

5.1.6. Problemas en torno a la compilación del proyecto

5.1.6.1. La construcción en algunas distribuciones de Linux falla.

Cuando se trata de construir en Linux, la documentación simplemente indica que se admiten "numerosas distribuciones de Linux"[150]. Cuando se construye en una distribución que no es muy común para los contribuyentes de Solidity, se espera que surjan algunos problemas en el camino.

Por ejemplo, a pesar de que el *script* `install_deps.sh`[164] tiene la capacidad de apuntar a *Archlinux*, no reconoce a *Manjaro Linux* como una distribución *Arch*, lo que da como resultado una distribución de Linux no admitida o no identificada.

Dado que las pruebas de CI para Linux solo se realizan en Ubuntu, depende de los contribuyentes con otras distribuciones descubrir estos errores, lo que se traduce en una experiencia terrible para el desarrollador y podría alejar a los posibles contribuyentes valiosos.

Además, las dependencias también pueden quedar obsoletas o introducir nuevos problemas que hacen que la creación y/o prueba en una plataforma específica sea imposible. Un ejemplo de esto fueron las fallas de compilación con el solucionador (solver) de **CVC4**, cuando se encuentran presentes en el sistema operativo, el compilador intentaría integrarlo a la compilación pero fallaría ya que las interfaces eran inconsistentes debido a las diferentes versiones y no había manera de deshabilitarlas (hasta ahora[221].

Se recomendó considerar la especificación de qué distribuciones de Linux son compatibles para la construcción en la documentación, e introducir pruebas de CI que simplemente aseguren que el compilador pueda integrarse en ellas. Además, agregar una pequeña sección en la documentación que explica cómo usar un contenedor para compilar el compilador en una de las plataformas compatibles.

Actualidad: parte de este problema, relacionado con la construcción en Archlinux y distribuciones similares, se trató en los *issues* **#4377**[202], **#4762**[173] y **#4767**[172].

5.1.6.2. Archivo faltante a la hora de compilar utilizando SANITIZE.

El flag de cmake llamado `SANITIZE` del archivo `EthCompilerSettings.cmake`[125] lee una blacklist (lista negra) de entidades a ignorar desde `sanitizer-blacklist.txt` a la hora de construir el proyecto. Tal archivo se encuentra inexistente en el proyecto, y para poder compilar sin que falle, se debe eliminar la siguiente línea, o crear un archivo vacío con ese nombre.

```
1 -fsanitize-blacklist=${CMAKE_SOURCE_DIR}/sanitizer-blacklist.txt"
```

Se recomendó considerar la incorporación del archivo o verificar si este existe antes de la compilación.

Actualidad: Se eliminó por completo el uso de una blacklist en la rama de *develop* (**PR#4560**[239]) y siguientes releases.

5.1.6.3. Manejo inseguro de variables de entorno en infraestructura de prueba.

Hay varias instancias en las que las variables de entorno se utilizan en las pruebas sin ningún tipo de control o saneamiento. Estas pueden haber sido modificadas por un atacante y, por lo tanto, deben tratarse con el mismo nivel de atención que cualquier otra información no confiable.

`/test/tools/isoltest.cpp:312`

```
1 if (getenv(``EDITOR"))
2     SyntaxTestTool::editor = getenv(``EDITOR");
```

`/test/Options.cpp:67`

```
1 if (!disableIPC && ipcPath.empty())
2     if (auto path = getenv(``ETH_TEST_IPC"))
3         ipcPath = path;
```

`/test/Options.cpp:70`

```
1 if (testPath.empty())
2     if (auto path = getenv(``ETH_TEST_PATH"))
3         testPath = path;
```

Se recomendó considerar revisar cuidadosamente las variables de entorno antes de usarlas. Por ejemplo, si se espera una ruta, verificar que sea realmente una ruta antes de usar esa variable.

Actualidad: El equipo de Solidity respondió: *"Como se explica en el otro issue, no creemos que tenga ningún valor protegerse contra un atacante que tenga control sobre las variables del entorno"*.

5.1.7. Problemas entorno al diseño del compilador

5.1.7.1. Los comentarios se pueden disfrazar de código ejecutable.

Es posible tener comentarios en un archivo Solidity que se verán como código ejecutable en la mayoría de los editores. Al analizar los comentarios, todos los caracteres se omiten hasta que se encuentra un carácter terminador de línea. Esto se puede ver en el código de `skipSingleLineComment` a continuación:

```
Token::Value Scanner::skipSingleLineComment() {
   while (!isLineTerminator(m_char))
       if (!advance()) break;
   return Token::Whitespace;
}
```

El código para `isLineTerminator` comprueba si el carácter actual es igual a `'\n' (hex 0x0a)`:

```
bool isLineTerminator(char c) {
   return c == '\n';
}
```

El problema es que hay caracteres distintos de `'\n'` que representan una nueva línea en UTF-8. Un ejemplo es el carriage return (retorno de carro, `hex 0x0d`), que fue el carácter de salto de línea predeterminado para *MacOS* hasta *MacOS* 9 (lanzado en 1999). Por lo tanto, el analizador considerará todo lo que sigue al *carriage return* como parte de la misma línea, marcándolo como un comentario e ignorando su contenido.

Considerar el siguiente ejemplo de una *wallet* compartida, donde se pueden depositar fondos asociados a un *address*, y luego solo pueden ser recuperarlos con ese *address*:

```
pragma solidity ^0.4.24;

contract SharedWallet {

 mapping (address => uint) pendingWithdrawals;

 function deposit() public payable {
   pendingWithdrawals[msg.sender] += msg.value;
 }

 function withdraw() public {
   uint amount = pendingWithdrawals[msg.sender];
   // Remember to zero the pending refund before
   // sending to prevent re-entrancy attacks
   pendingWithdrawals[msg.sender] = 0;
   msg.sender.transfer(amount);
 }
}
```

Pero, hay una trampa. Un usuario puede ver el código anterior exactamente como se muestra en este documento, pero el carácter de nueva línea utilizado en el último comentario no es como los otros.

Este es el código manipulado:

```
// sending to prevent re-entrancy attacks
 pendingWithdrawals[msg.sender] = 0;
```

Con el hexadecimal equivalente:

```
2f2f 2073656e64696e67 746f 70726576656e74 72652d656e7472616e6379 61747461636b730d
70656e64696e6757697468647261177616c735b6d73672e73656e6465725d 3d 303b0a
```

El analizador reconocerá los primeros dos caracteres `(0x2f2f)` como un token de comentario de una sola línea y consumirá todo lo que quede, hasta el siguiente token de nueva línea `(0x0a)`, faltando el carriage return `(0x0d)` en el camino. Por lo tanto, el compilador nunca procesa la asignación de mapeo.

Lo que realmente sería compilado es esto:

```
//...
function withdraw() public {
  uint amount = pendingWithdrawals[msg.sender];
  // Remember to zero the pending refund before
  // sending to prevent re-entrancy attacks pendingWithdrawals[msg.sender] = 0;
  msg.sender.transfer(amount);
}
```

Lo que el atacante ha logrado es engañar al lector para que piense que la función de retorno permitirá que el participante retire solo lo que se había depositado anteriormente, pero en cambio le permitirá al participante retirar la misma cantidad sin límite, ya que no se realiza la asignación a cero.

Los sistemas de comando Unix que involucran `stdout` se comportarán de manera diferente: `cat`, por ejemplo, no mostrará el comentario en absoluto. Esto se debe a que la línea que sigue al retorno de carro está escribiendo sobre los caracteres de la línea anterior, e incluso `diff` no podrá mostrar la diferencia entre los archivos originales y los malintencionados, sin embargo, un editor para terminales como `vim` podrá; los editores modernos no-terminales lo mostrarán como una nueva línea regular.

El comportamiento actual del compilador permite crear backdoors casi indetectables con poco esfuerzo. Se recomendó agregar todas los distintos tipos de salto de línea no adaptados, reconocidos por el estándar de Unicode[263] a `isLineTerminator`, y probar un comportamiento más inesperado mientras maneja los caracteres válidos y no válidos de UTF-8.

Actualidad: se ha aplicado una solución[217] y se ha publicado en `v0.4.25`. Line feed, vertical tab, form feed, carriage return, NEL, LS y PS ahora se consideran válidos para terminar un comentario de una sola línea.

5.1.7.2. Todos los strings son UTF-8.

Las cadenas en Solidity no solo se usan para mostrar información: por ejemplo, es muy común que sean la clave de un mapeo. Debido a que UTF-8 permite múltiples caracteres invisibles (por ejemplo, ZERO WIDTH SPACE[262]), y para caracteres que se parecen casi a caracteres comunes (por ejemplo, GREEK QUESTION MARK[261]), este uso puede ser extremadamente problemático, y puede llevar a backdoors, exploits, etc. Esto afecta a los principales contratos de control de acceso[213], al igual que muchas otras implementaciones basadas en strings.

Se recomendó considerar agregar un tipo de string que no sea UTF-8 para evitar que estas situaciones surjan en primer lugar.

Actualidad: issue **#5167**[167] creado.

5.1.7.3. Los modificadores se pueden anular sin una sintaxis especial o advertencias.

Los contratos pueden anular cualquier modificador en el árbol de herencia simplemente definiendo uno nuevo con la misma *firma* (*signature*, una combinación única de caracteres para identificar una estructura y así después poder referenciarla). Si bien se produce un error si la firma de reemplazo no coincide, no hay advertencias para el caso en el que lo hacen.

```
contract Ownable {
 address public owner;

 modifier onlyOwner() {
  require(msg.sender == owner);
  _;
 }
}
contract ModifierOverride is Ownable {
 modifier onlyOwner() {
  _;
 }
}
```

Los modificadores (`modifier`) se usan normalmente para el control de acceso, el saneamiento de entradas, etc., permitiendo anulaciones de este tipo que son un riesgo para la seguridad, ya que obliga a los desarrolladores a revisar el código para verificar manualmente cada modificador de manera individual para ver si está anulando otro, y la herencia múltiple hace la tarea aún más engorrosa.

Es bastante común que los desarrolladores declaren un modificador y anulen involuntariamente a su declaración anterior, en algunos casos con graves consecuencias para la seguridad de la aplicación.

Se recomendó considerar agregar una palabra clave como `override`, con una sintaxis similar a la de C++11[214]. Esto asegurará que las modificaciones del modificador sean siempre explícitas, tanto para los desarrolladores como para los revisores de códigos.

Actualidad: El equipo de Solidity respondió: *"Hay varios problemas al respecto. Probablemente se arreglará en v0.6.0"*. Hubo un intento de solucionar el inconveniente en **PR#3737**[220], y continúa en etapa pendiente para testear con la versión v0.6.0. Ver *issues* **#2563**[198] y **#973**[205].

5.1.7.4. Las variables de estado se pueden ver opacadas por otras.

Los contratos que heredan de otros contratos pueden declarar variables de estado con el mismo nombre que las variables `internal` o `public` en un contrato base, utilizando un nuevo slot (ranura) de almacenamiento y ocultando las originales. Esto significa que los accesos desde la base y el contrato derivado se referirán a las instancias declaradas en cada una, a pesar de que el nombre sea el mismo. También habrá una sola función *getter* generada automáticamente, dirigida a la variable del contrato que está en el último gráfico de la herencia. Por ejemplo, en el siguiente código, `baseGetter` devolverá un `address`, `derivedGetter` devolverá un `uint256`, y el *getter* de `x` generado automáticamente devolverá un valor de `uint256`:

```
pragma solidity ^0.4.24;

contract Base {
  address public x;

  constructor() public {
    x = msg.sender;
  }

  function baseGetter() public view returns (address) {
    return x;
  }

}
```

```
contract Derived is Base {
  uint256 public x;

  constructor() public {
    x = 20;
  }

  function derivedGetter() public view returns (uint256) {
```

```
9       return x;
10    }
11  }
```

Este comportamiento puede ser confuso tanto para principiantes como para usuarios avanzados, especialmente cuando se trata de la sustitución de los *getters* generados automáticamente.

Considere rechazar la reutilización de nombres de variables internas y públicas de un contrato base.

5.1.7.5. No hay ningún mecanismo para evitar que se invaliden las funciones.

Si bien la herencia es muy conveniente para diseñar funcionalidades de forma modular, la falta de un mecanismo para deshabilitar la anulación de una función puede causar problemas. Resulta difícil razonar acerca de un contrato como una entidad aislada, ya que sus funciones pueden haber sido modificadas por otros contratos en el árbol de herencia. Esto eleva el nivel para entender un contrato inteligente al simplemente leer su código, permite backdoors sutiles y evita que los desarrolladores demuestren sus intenciones. El issue **#501**[203] de OpenZeppelin es un ejemplo en el que la incapacidad de verificar si un contrato derivado modifica el comportamiento de la base causó discusiones y confusión.

Se recomendó considerar agregar una palabra clave `final` o `sealed` que deshabilite los reemplazos para funciones y modificadores, causando un error del compilador si se hace un intento de modificarlos.

Actualidad: el equipo de Solidity respondió: *"Probablemente sea mejor requerir una palabra clave como `virtual` si una función puede ser modificada por un contrato derivado y el valor predeterminado debe ser sealed. Esto se solucionará con la limpieza de herencia para v0.6.0"*[135]. Ver *issue* **#5424**[196].

5.1.7.6. Se permiten secuencias UTF-8 no válidas en los comentarios.

Al analizar los comentarios (*parsing*), todos los caracteres se omiten hasta que se encuentra un carácter de nueva línea. Esto significa que las secuencias UTF-8 no válidas dentro de los comentarios no serán detectadas, lo que potencialmente podría conducir a un código no esperado/autorizado (es decir, lo que parece un comentario puede contener código).

Esto generalmente no es un problema, ya que la mayoría de los editores ignorarán las secuencias no válidas y se volverán a sincronizar con la secuencia tan pronto como se encuentre un carácter válido. Sin embargo, dado que realizar revisiones de código en el código fuente de Solidity es una tarea tan crítica, sería preferible no cargar con esta responsabilidad a todos los editores.

Se recomendó considerar el escaneo de todo el código fuente y rechazar código que no satisface el estándar UTF-8 (*non-compliant*) antes de realizar cualquier análisis.

5.1.8. Problemas en torno al proceso de optimización

5.1.8.1. Las optimizaciones opcionales pueden no ser seguras.

Debido a que algunas optimizaciones son opcionales, no están tan probadas como otras optimizaciones comúnmente usadas.

Se recomendó considerar el aumento de la cobertura del código de optimización, agregar informes de código de Solidity optimizado probado y agregar un aviso / advertencia cuando las optimizaciones estén habilitadas.

Luego, considerar animar a los desarrolladores a contribuir con Solidity al habilitar estas optimizaciones en las compilaciones de sus proyectos.

Actualidad: el equipo de Solidity respondió: *"Todas las pruebas semánticas se ejecutan con el optimizador activado y el optimizador desactivado. Además, todos ellos se ejecutan en combinación con varias versiones de la EVM".*

5.1.8.2. El código de las optimizaciones en el assembler (libevmasm) es difícil de leer.

El código de optimizaciones[133] es muy difícil de leer en `libevmasm`. Esto hace que sea difícil para los revisores de código interpretar, para que los contribuyentes mejoren y para que los desarrolladores agreguen pruebas.

Se recomendó considerar la posibilidad de refactorizar estos archivos utilizando los Clean Code Principles[160] (principios de código limpio). En particular, evite tener bloques profundamente anidados, números mágicos, funciones largas y código complejo sin comentarios.

Actualidad: este código de optimizaciones se eliminará y solo se utilizarán las optimizaciones de *YUL*.

5.1.9. Problemas en el contexto de mensajes (de salida)

5.1.9.1. No hay mensaje de error en las referencias de almacenamiento no inicializadas.

Cualquier escritura en una referencia de un *uninitialized* `storage` *reference* (almacenamiento no inicializada) puede sobrescribir el estado, ya que las referencias siempre tendrán un valor predeterminado. Esto, combinado con el hecho de que los tipos de referencia apuntan al `storage` por defecto, hace que sea muy fácil escribir código que se vea bien para un no experto, pero que en realidad contenga un error grave o un *backdoor*.

En el siguiente contrato, aunque foo no accede directamente a "a", lo modifica, y será igual a 3 después de llamar a foo (porque la longitud de "b"lo sobrescribirá).

```
1  pragma solidity ^0.4.24;
2  contract Storage {
3   uint256 public a;
4   constructor () public {
5    a = 8;
6   }
7   function foo() public {
8    uint256[] b;
9    b.push(5);
10   b.push(6);
11   b.push(7);
12  }
13 }
```

Si bien se emite una advertencia para estos contratos sobre un *uninitialized storage pointer* (puntero de almacenamiento no inicializado, y esto se señala en la documentación[265]), aún se pueden compilar e implementar. El código que emite esta advertencia siempre será incorrecto, por lo que no tiene sentido permitir que se compile. Se recomendó considerar hacer de esto un error de falla.

Actualidad: esta advertencia ya se ha convertido en un error y el cambio se publicó[224] en la versión `v0.5.0`.

5.1.9.2. La falta de la declaración de un retorno en una función no emite un error.

Cuando una función se declara con un parámetro de retorno, pero el cuerpo de la función no tiene la declaración de `return`, el compilador no muestra un error, ni advierte al usuario sobre esto.

Para ilustrar, considere los siguientes ejemplos de código, en los que no solo las funciones que declaran un valor devuelto no lo devuelven, sino que otras funciones intentan usar los valores devueltos sin quejas del compilador.

Código de ejemplo:

```
pragma solidity ^0.4.24;
contract Empty {
 int variableName;
 constructor() public {
   variableName = 0;
 }
 function emptyReturn() public pure returns (int) { }
 function setWithEmptyReturn() public {
   variableName = emptyReturn();
 }
}
```

Se recomendó considerar el uso del análisis de flujo de control para advertir a los usuarios cuando una función declara un parámetro de retorno pero no devuelve nada.

Actualidad: este problema se encuentra en el issue **#4751**[194] y se etiquetó bajo el área de diseño del lenguaje *language design*.

5.1.9.3. Los modificadores pueden utilizar un retorno (`return`).

Si bien los modificadores requieren el carácter `'_'` para indicar dónde se ejecutará el cuerpo de la función, pueden retornar antes de dicha ejecución, deshabilitando toda la función a la que se aplica el modificador.

En el siguiente código, la asignación a la variable `'a'` nunca se ejecuta, pero se compila sin advertencias:

```
pragma solidity ^0.4.24;
contract Disabled {
 uint256 a;
 modifier disable() {
  return;
  _;
 }
 function foo() disable public {
  a = 42;
 }
}
```

Mientras que retornar en un modificador a veces tiene sentido, el retorno generalmente no lo es, especialmente considerando que las diferentes funciones tendrán diferentes *signatures* (firmas) y tipos de valor de retorno.

Se recomendó considerar el rechazo de retornos dentro de un modificador.

Actualidad: Se creó el issue **#2340**[199], y se vió solucionado en la versión `v0.5.3` con el pull request **#5765**[238].

5.1.9.4. No hay error al llamar de manera externa el código del constructor de un contrato.

El uso de la palabra `this` en el cuerpo del constructor de un contrato para llamar a otras funciones dentro del contrato significa que la llamada se realizará en un contexto externo (es decir,

no con un salto regular). Dado que el *bytecode* del contrato en tiempo de ejecución no existe en el momento en que se ejecuta el cuerpo del constructor, usar `this` para llamar a una función pública o externa dentro de un constructor siempre será inválido.

El compilador emite una advertencia para esto, pero considerando que un constructor con tal problema siempre revertirá (`revert`), el compilador debería generar un error en su lugar.

```
pragma solidity ^0.4.24;
contract ConstructorThis {
 constructor () public {
   this.foo();
 }
 function foo() external pure {}
}
```

```
Warning: ‘‘this" used in constructor. Note that external functions of a contract
    cannot be called while it is being constructed.
  this.foo();
  ^--^
```

Se recomendó considerar hacer de esta advertencia un error para indicar mejor que este uso del idioma es incorrecto.

Actualidad: el equipo de Solidity respondió: *"Hay que tener en cuenta que el uso de `'this.f'` sigue siendo válido en el constructor y tiene un caso de uso específico: enviar una función a otro contrato como un mecanismo de devolución de llamada".*

5.1.9.5. No hay advertencia para *dead code* (código muerto).

Las funciones pueden tener sentencias que nunca se alcanzarán (debido a un retorno anticipado), pero el compilador no emitirá advertencias para éstas, incluso si tienen efectos secundarios.
El siguiente código compila sin advertencias:

```
pragma solidity ^0.4.24;
contract DeadCode {
 uint256 a;
 function foo() public {
  return;
  a = 42;
 }
}
```

Los compiladores de otros lenguajes (como *clang*, *rustc*, entre otros) emitirán una advertencia en estas situaciones, se recomienda considerar la posibilidad de emular este comportamiento.

Actualidad: se discutió en el issue **#2340**[199], y se implementó en la versión `v0.5.3` con el pull request **#5765**[238].

5.1.10. Problemas en torno a técnicas blackbox.

Esta sección cubre un análisis sobre la configuración del mecanismo *fuzzing* del proyecto Solidity y los problemas encontrados aplicando técnicas complementarias.

Como alternativa a la creación de `solc`, hay una opción para compilar un binario llamado `solfuzzer`. Este archivo, `fuzzer.cpp` se construye como un punto de entrada para el *Fuzzy Lop* fuzzer estadounidense (AFL), utilizando su propio compilador que podrá instrumentarlo para su uso posterior con el resto de su frmework. Con la ayuda de un archivo *Python* en la carpeta de scripts llamada `isolate_tests.py`[166], que se utiliza para extraer el código de Solidity de los archivos existentes, AFL testea utilizando black-box a `solfuzzer` como binario de entrada.

5.1.10.1. La configuración de fuzzing está rota.

Aplicar fuzzing es una muy buena garantía que puede llevar a encontrar errores significativos. En el proyecto Solidity no se estuvo realizando correctamente durante más de un año.

El propósito principal de `solfuzzer` es compilar archivos de entrada y verificar si se detecta algún error. Si se detecta un error basándose en una lista proporcionada de mensajes de error esperados, termina la ejecución notificando que se encontró un error.

El mensaje de error cambió hace un año desde el momento en el que se inició la investigación y la lista de errores del archivo nunca se actualizó, por lo que las detecciones no funcionaron, afectando a todo el *fuzzing* y también a las pruebas en `cmdLineTests.sh`[126] que utilizaban la misma lista.

Se recomendó considere arreglar las pruebas de *fuzzing* y verificar periódicamente si funcionan correctamente.

Actualidad: Este *issue*[187] ha sido resuelto[222].

5.1.10.2. Fallo al intentar declarar una variable ya declarada con el mismo nombre.

El analizador no detecta una variable previamente declarada cuando declara una nueva, mientras usa `pragma experimental`.

Código de ejemplo:

```
1 pragma solidity ^0.4.24;
2 pragma experimental ‘‘v0.5.0";
3 contract CrashContract {
4   function f() view public {
5     int variableDefinedTwice;
6     address variableDefinedTwice;
7   }
8 }
```

Actualidad: Ver la descripción de este *issue*[181]. El team de Solidity respondió que el problema ha sido resuelto efectivamente.

5.1.10.3. Fallo al convertir la un racional con signo utilizando ABIEncoderV2

El compilador se bloquea en la generación del código de assembler cuando se utiliza `pragma experimental ABIEncoderv2` para codificar un racional con signo.

Código de ejemplo:

```
1 pragma solidity ^0.4.24;
2 pragma experimental ABIEncoderV2;
3 contract CrashContract {
4  function f1() public pure returns (bytes) {
5    return abi.encode(1,-2);
6   }
7 }
```

Actualidad: La descripción de este problema se encuentra en el issue **#4706**[179], y se ha solucionado en **PR#4720**[229].

5.1.10.4. fuzzer.cpp y solfuzzer tienen nombres poco intuitivos.

`fuzzer.cpp` y su resultante, `solfuzzer`, tienen nombres contraintuitivos: `solfuzzer` es un punto de entrada para AFL u otros *fuzzers* que funcionan con instrumentación, pero no es un *fuzzer* por sí solo. Esto puede hacer que los usuarios piensen que están aplicando *fuzzing* cuando en realidad no lo están.

La sección de documentación también se titula *"Ejecutando el Fuzzer a través de AFL"*[244]; pero AFL en sí es el *fuzzer*, no el binario compilado.

Se recomendó considerar el cambio de el nombre del punto de entrada a algo que transmita claramente su propósito.

Actualidad: `fuzzer.cpp` ha sido renombrado a `afl_fuzzer.cpp`, sólo para diferenciar con qué tipo de *fuzzer* realmente trabaja, pero el concepto del problema permanece vigente hasta el momento de esta redacción.

5.1.10.5. El ejemplo de la documentación respecto a AFL no funciona.

Las pruebas de *aislamiento*, como se muestra en la sección *"Contribuyendo sobre fuzzing"*[245], no funcionan. El script `isolate_tests.py`[166] funciona con directorios como se usa en los scripts de prueba, pero no acepta archivos individuales como forma de entrada tal como se muestra en los ejemplos.

Se recomendó considerar arreglar el script `isolate_tests.py` para que pueda aceptar archivos individuales.

Actualidad: ya se ha incorporado una solución propuesta[237] para solucionar este problema.

5.1.10.6. Planificación de pruebas de fuzzing y su visibilidad.

Las pruebas de *fuzzing* no se realizan periódicamente ni públicamente, tampoco los resultados se hacen visibles a la comunidad. Adicionalmente no está claro en qué parte del proceso de desarrollo o el conjunto de pruebas está integrado.

Además de proporcionar una guía para configurar el fuzzer, se recomendó considerar publicar los resultados y la información al público después de aplicar fuzzing antes de cada lanzamiento. Actualidad: Una propuesta para integrarse en un servicio público de fuzzing en el issue **#5212**[195] ha sido incorporada en las últimas versiones utilizando *oss-fuzz*, un sistema contínuo open source para realizar *fuzzing*.

5.1.10.7. Fallo cuando el tipo de dato solicitado no se encuentra presente.

El compilador interrumpe su ejecución repentinamente cuando hay variables de estado con el mismo nombre que una función.

Código de ejemplo:

```
pragma solidity ^0.4.24;
contract C {
  uint256 public f = 0;
  function f() public pure {}
}
```

Actualidad: una descripción más detallada se encuentra en el issue **#4417**[192]. El problema se solucionó en **PR#4508**[227].

5.1.10.8. Fallo cuando se accede al slot de una variable de nombre vacío.

El analizador no detecta el mal uso de la sintáxis `_slot` sobre una variable con nombre vacío.

Código de ejemplo:

```
pragma solidity ^0.4.24;
contract CrashContract {
   function () internal {
```

```
4         assembly {
5             _slot
6         }
7     }
8 }
```

Actualidad: una descripción más detallada se encuentra en el issue **#4707**[175]. El problema se solucionó en **PR#4724**[225].

5.1.10.9. Fallo cuando no se setea el tipo para el valor de retorno del parámetro.

El analizador no reconoce que la variable de retorno no tiene su tipo en la declaración de variable del tipo en el caso de una función sin nombre.

Código de ejemplo #1:

```
1 pragma solidity ^0.4.24;
2 contract CrashContract {
3   function () returns (variableNameWithoutType) variableName;
4 }
```

Código de ejemplo #2:

```
1 pragma solidity ^0.4.24;
2 contract CrashContract {
3   function() internal returns (zeppelin)[] x;
4 }
```

Actualidad: una descripción más detallada se encuentra en el issue **#4708**[183]. El problema se encuentra solucionado en las versiones siguientes.

5.1.10.10. Fallo cuando no se setea el tipo para un parámetro de una función.

El analizador no reconoce que a una variable le falta su tipo en los parámetros de la función; ocurre al tener un contexto en el que se define una función sin nombre que retorna una matriz.

Código de ejemplo:

```
1 pragma solidity ^0.4.24;
2 contract CrashContract {
3   function(parameterWithoutType) internal returns (uint)[] y;
4 }
```

Actualidad: una descripción más detallada se encuentra en el issue **#4709**[182]. El problema se encuentra solucionado en las versiones siguientes.

5.1.10.11. Fallo al acceder al slot de una función en un bloque assembler.

El método visit falla al acceder al `_slot` de una función dentro de un bloque *assembler* de esa misma función.

Código de ejemplo:

```
1 pragma solidity ^0.4.24;
2 contract CrashContract {
3   function f() pure public {
4     assembly {
5       function g() -> x { x := f_slot }
6     }
7   }
8 }
```

Actualidad: una descripción más detallada se encuentra en el *issue* **#4710**[174]. El problema se solucionó en **PR#4729**[232].

5.1.10.12. Fallo al llamar a un tipo no llamable en una asignación doble de tipo no primitivo.

El compilador falla cuando se usa un tipo no llamable (`int, uint, struct`, etc.) fuera de una doble asignación que involucra `structs`.

Código de ejemplo:

```
pragma solidity ^0.4.24;
contract CrashContract {
  struct S { }
  S x;
  function f() public {
    (x, x) = 1(x, x);
  }
}
```

Actualidad: una descripción más detallada se encuentra en el *issue* **#4711**[177]. El problema se solucionó en **PR#4736**[233].

5.1.10.13. Fallo al utilizar instrucciones assembler de salto dentro de un constructor o función con el mismo nombre que el contrato.

La generación de código falla cuando hay una instrucción `jump` dentro de un bloque assembler que está dentro de una función con el mismo nombre que el contrato o un constructor.

Código de ejemplo #1:

```
pragma solidity ^0.4.24;
contract f {
  function zeppelin() {}
  function f() {
    assembly {
      jump(zeppelin)
    }
  }
}
```

Código de ejemplo #2:

```
contract CrashContract {
  function zeppelin() {}
  constructor() {
    assembly {
      jump(zeppelin)
    }
  }
}
```

Actualidad: una descripción más detallada se encuentra en el *issue* **#4712**[184]. El equipo de Solidity respondió que la instrucción de salto fue eliminada.

5.1.10.14. Detenimiento abrupto en diversas situaciones al utilizar `pragma experimental ABIEncoderV2`

Un arreglo de `structs` que está compuesto de uno o más arreglos es utilizado como parámetro en una función externa de una biblioteca. **Código de ejemplo:**

```
pragma experimental ABIEncoderV2;
pragma solidity ^0.4.24;
library Test {
  struct Nested { int[] a; }
  function Y(Nested[]) external {}
}
```

Actualidad: una descripción más detallada se encuentra en el *issue* **#4713**[180]. El problema se solucionó en **PR#4738**[228].

Al utilizar struct como un parámetro de una función externa. **Código de ejemplo:**

```
pragma experimental ABIEncoderV2;
pragma solidity ^0.4.24;
library Test {
  struct Nested { }
  function Y(Nested a) external {}
}
```

Actualidad: una descripción más detallada se encuentra en el issue **#4714**[185]. El problema se solucionó en **PR#4738**[228].

En la etapa de generación de *bytecode* al utilizar el método `encode()` de la ABI para un tipo de punto flotante. **Código de ejemplo:**

```
pragma solidity ^0.4.24;
pragma experimental ABIEncoderV2;
contract C {
  function f1() public pure returns (bytes) {
    return abi.encode(0.1, 1);
  }
}
```

Actualidad: se creó el issue **#4715**[178], y finalmente una solución **PR#5807**[235] se incorporó en la versión `v0.5.3`.

5.1.10.15. Fallo cuando el índice de un arreglo es sumamente largo.

El compilador se bloquea si un tipo racional de más de 78 dígitos está presente como un índice de un arreglo.

Código de ejemplo:

```
pragma solidity ^0.4.24;
contract CrashContract {
  function f() returns (string) {
        return (['zeppelin"
            ][12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678])
            ;
  }
}
```

Actualidad: una descripción más detallada se encuentra en el *issue* **#4716**[176]. El problema se solucionó en **PR#4872**[223].

5.1.10.16. Gran uso de ciclos del CPU al convertir literales numéricos.

El uso de valores numéricos literales aumenta el tiempo de compilación y el uso de la CPU, con un mayor retraso cuando es más grande, siendo `0e2147399999` el número más alto posible. La compilación puede tardar varios días y mucho más.

Código de ejemplo:

```
pragma solidity ^0.4.24;
contract VerySlowContract {
  function test() public returns (uint256) {
    return 0e2147399999;
  }
}
```

Actualidad: una descripción más detallada se encuentra en el *issue***#4717**[188]. El problema se solucionó en **PR#4765**[230].

5.1.10.17. Gran uso de ciclos del CPU al utilizar nombres de variables largos.

El tiempo de compilación se ve incrementado al utilizar nombres de variables grandes, con una latencia aparentemente mayor cuando las variables son más grandes y similares. La compilación

puede llevar varios días o más.

Código de ejemplo: *Nota: el código a continuación está truncado, el código completo se puede encontrar en internet[131].*

```
pragma solidity ^0.4.24;
contract VerySlowContract {
  function f() public {
    int YYYYYYY...YYYYYYYYY = YYYYYYYYY....YYYYYYYYY;
  }
}
```

Actualidad: una descripción más detallada se encuentra en el issue **#4718**[189]. El problema se solucionó en **PR#4797**[216].

5.1.11. Observaciones relacionadas a requerimientos no funcionales

Se recomendó considerar las siguientes recomendaciones para mejorar la calidad del sistema. Un *issue* **#5168**[201] se abrió para llevar registro del proceso. Herramientas como `clang-tidy` proporcionan una útil perspectiva después de analizar el código fuente del proyecto.

Modernización

- Utilizar `auto` a la hora de declarar iteradores e inicializarlos con un casteo para evitar duplicados.
- Utilizar `nullptr` en vez de `NULL`. Update: Corregido en **PR#5180**[236].
- Utilizar `emplace_back` en lugar de `push_back`.
- Evitar repetir el tipo de retorno en declaración; en su lugar, utilizar una lista de inicialización mediante llaves.
- Utilizar `bool` literal en lugar de representaciones a enteros.
- Strings literales escapados deberían escribirse como strings literales en *raw* (crudo).
- Utilizar `cctype` en lugar del *header* deprecado C++ `ctype.h`. Corregido en **PR#5180**[236]
- Utilizar `cstdio` en lugar del *header* deprecado C++ `stdio.h`. Update: Corregido en **PR#5180**[236].
- Utilizar el *keyword* de C++11 `override` sobre `virtual` para una anulación de una clase derivada.
- Utilizar `'= default'` para definir un constructor trivial por defecto.
- Utilizar `std::make_unique` en lugar de of `std::unique_ptr`. Actualmente se realizaron cambios en torno a esto: **PR#6712**[219], **PR#5694**[218].
- Cuando una función es declarada con `override` remover `virtual` por redundancia.

Readability

- Reemplazar `boost::lexical_cast<std::string>` con `std::to_string` para tipos fundamentales. Corregido en **PR#4753**[234].
- Reducir conversiones implícitas mediante *built-in types* (tipos incorporados) y booleanos.
- Utilizar los mismos nombres para todos los parámetros, en la declaración tanto como en la implementación.
- Acceder a miembros estáticos directamente, y no a través de instancias.
- Utilizar `empty()` en lugar de `size` a la hora de chequear por vacío en un contenedor. Corregido en **PR#5180**[236].

- Normalizar nombres de parámetros en declaraciones (*.h) para que no difieran con sus implementaciones (*.cpp).
- Evitar utilizar `static` dentro de *namespaces* anónimos, ya que `namespace` limita la visibilidad de las definiciones a una unidad de traducción individual.

Performance

- Utilizar búsquedas de caracteres individuales literales cuando sea posible (por ejemplo `'
n'` en lugar de `"
n"`). Corregido en **PR#5180**[236].
- Utilizar referencias con `const` para variables utilizadas en iteraciones que son copiadas pero sólo usadas como referencias constantes.
- Si una variable es construida mediante copia (`auto`) de una referencia `const`, utilizar `const &`.
- Utilizar `append()` en lugar de `operator=+` a la hora de concatenar `strings`.
- Quitar std::move en tipos que usen `const` o que trivialmente sean tipos copiables.

Se referencia a más información sobre este tipo de pautas en el artículo *C++ Core Guidelines*[121] y en la *lista de chequeos de Clang-Tidy*[124].

Capítulo 6

Trabajo futuro

Este trabajo brinda la posibilidad al lector de replantearse la posición que se toma o tomará respecto a la seguridad en cada proyecto, pasado, presente y futuro.

Viéndolo desde distintos ángulos se pueden separar los siguientes vectores a desarrollar:

1. utilizar herramientas de mayor nivel de abstracción para la auditoría de software.
2. aplicar metodologías que hagan énfasis en el aspecto de seguridad en lenguajes/compiladores.
3. aplicar procesos/metodologías conocidos y probados del campo de la auditoría de software.

6.0.1. (1) Herramientas abstractas para la auditoría de software

Una de las herramientas que existen en el mercado que más han llamado la atención, es la desarrollada por la empresa *Semmle*, que posee un lenguaje de consultas llamado *QL*.

QL permite realizar rápidamente análisis de variantes para encontrar vulnerabilidades de seguridad previamente desconocidas. QL trata el código como datos que le permiten escribir consultas personalizadas para explorar su código.

QL se entrega con amplias bibliotecas para realizar control y análisis de flujo de datos, seguimiento de contaminación y explorar modelos de amenazas conocidos sin tener que preocuparse por conceptos de lenguaje de bajo nivel y detalles del compilador. Los lenguajes compatibles incluyen C / C ++, C#, Java, Javascript, Python y más.

El tipo de consultas que se pueden hacer es casi tan simple como buscar "variables no inicializadas dentro de un ciclo".

Este tipo de herramientas parecen ser de lo más adecuadas para trabajar los problemas que aparecen al tener un nivel de abstracción superior, dado que las consultas semánticas se pueden reutilizar a través de distintos proyectos.

6.0.2. (2) Seguridad en lenguajes/compiladores

La seguridad como parte de los compiladores es un campo activo y reciente. Se encuentran alternativas en discusión, como proveer de características a LLVM[134][247] para que sea más seguro. Aunque posee dificultades para detectar a tan bajo nivel potenciales problemas de seguridad sin un contexto dado.

Otra opción es proveer un lenguaje seguro por defecto como es el caso de Rust. Su inconveniente es que si debe interaccionar con software que no está hecho en su mismo lenguaje, su seguridad se vuelve tan segura como la del software con el que interactúa. Es un problema recurrente en las grandes empresas que lo están incorporando, y rehacer todo el código en un nuevo lenguaje no suele ser una opción viable.

6.0.3. (3) Procesos/metodologías de auditoría de software

Desde esta perspectiva hace falta hacer más concientización para que lo que se incorporen no sean más herramientas aisladas sino procesos/metodologías que las contengan. Estas metodologías han mostrado ser efectivas durante muchos años en el proceso del desarrollo de cualquier sistema de software y no existen motivos para no aplicarlas en el desarrollo de un compilador.

Capítulo 7

Conclusiones

Mientras más avanzamos en el desarrollo de nuevas tecnologías, más difícil se hace la búsqueda de la panacea que permita resolver el problema al todo. Es inevitable el *tradeoff* de querer producir una solución lo suficientemente abstracta aplicable sin perder efectividad, y no parece haber otra opción más asertiva que entender el contexto de cada problema y aplicar su propia solución.

7.1. Conclusiones del trabajo realizado

En esta tesis se ha mostrado cómo fue posible realizar una auditoría de un software, como es caso del compilador del lenguaje de *smart contracts* más utilizado en el momento, *Solidity*, partiendo de un conjunto de conceptos teóricos, utilizando metodologías y tecnologías relacionadas con con área de lenguajes y compiladores, sistemas operativos, seguridad informática, arquitecturas de hardware, programación, ingeniería de software en sus aspectos de metodologías de desarrollo, calidad, testing y documentación entre otras.

Todos los conocimientos de las áreas mencionadas de la carrera de Ingeniería de Sistemas se han aplicado en forma conjunta y sinérgica para lograr los objetivos planteados.

Además, este trabajo sirvió para comprobar cuántas áreas de la disciplina se encuentran tratadas adecuadamente en un producto de software y los motivos pragmáticos por lo que eso no sucede en ciertos casos.

7.2. Conclusiones de la auditoría en seguridad en general

7.2.1. Automatización como respuesta

La mejor herramienta que permitió tener un acercamiento efectivo al realizar una auditoría de software, más particularmente a un compilador, DeepSmith (3.2), fue mediante el uso de inteligencia artificial. Parece ser la más avanzada del estado del arte, no obstante, no es fácil de configurar, no es fácil de entrenar, y no es trivial de adaptar a cualquier proyecto.

Es inevitable hacerse preguntas como *¿por qué siempre se introduce al fuzzing como una herramienta crucial en auditorías?*, o *¿es realmente la única opción como herramienta?*

Las respuestas parecen ser obvias cuando no se posee perspectiva de seguridad desde el comienzo de un proyecto. Es tentador recaer a herramientas del estilo cuando se observan increíbles ventajas a la hora de testear aplicando seguridad. Particularmente si siendo desarrollador se comparan estrategias de *fuzzing* con *unit testing*. En la tabla 7.2.1 se resume la evolución de las soluciones de testeo para algunos aspectos de la auditoría de un proyecto como el de esta tesis.

Atributo	**Unit testing**	**Primeros fuzzers**	**Fuzzers actuales**
Testear pequeñas partes de código	√	X	√
Se puede automatizar	√	√	√
Test de regresión	√	√/X	√
Fácil de escribir	√	X	√
Buscar nuevos bugs	√/X	√√√	√√√√√√
Buscar vulnerabilidades	X	√	√

7.2.1.1. La seguridad en el SDLC

Sin ir más lejos, por más que exista la herramienta perfecta, el factor humano siempre es necesario. Mientras humanos sean quienes desarrollan los proyectos que requieran de una auditoría, se necesitarán metodologías y procesos que permitan tener respuestas más humanas.

La falla está en tratar a la seguridad como un adicional, como un feature que se puede incorporar después. La clave está en integrar lo antes posible. Sin ir más lejos, uno de los libros más acertados de la época **Accelerate: Building and scaling high performing technology organizations**[91], basándose en estudios a miles de empresas y un seguimiento a través de los años, lo presenta claramente en uno de sus capítulos, *Shifting left to security* (desplazando la seguridad hacia la izquierda).

Descubrieron que cuando los equipos *desplazan a la izquierda* a la seguridad informática, es decir, cuando lo integran en el SDLC en lugar de convertirlo en una fase separada que ocurre más adelante del proceso de desarrollo, esto impacta positivamente en el *delivery performance* (rendimiento de la entrega).

¿Qué implica "desplazar a la izquierda¿ Primero, se realizan revisiones de seguridad para todas las funciones principales, y este proceso de revisión se realiza de tal manera que no ralentiza el proceso de desarrollo. ¿Cómo se puede garantizar que prestar atención a la seguridad no reduzca el rendimiento del desarrollo? Este es el enfoque del segundo aspecto de esta capacidad: la seguridad de la información debe integrarse en todo el ciclo de vida de entrega del software desde el desarrollo hasta las operaciones. Esto significa que los expertos de seguridad deben contribuir al proceso de diseño de aplicaciones, asistir y proporcionar comentarios sobre las demostraciones del software, y garantizar que las características de seguridad se prueben como parte del conjunto de pruebas automatizadas. Finalmente, hay que facilitar a que los desarrolladores hagan lo correcto en este aspecto. Esto se puede lograr asegurando que haya bibliotecas, paquetes, cadenas de herramientas y procesos preprogramados y fáciles de consumir disponibles para los desarrolladores y las operaciones de IT.

Lo que se observa aquí es un cambio de los equipos de seguridad que realizan las revisiones de seguridad ellos mismos, a proveer a los desarrolladores los medios para construir su propia seguridad. Esto refleja dos realidades: Primero, es mucho más fácil asegurarse de que las personas que construyen el software están haciendo lo correcto que inspeccionar los sistemas y características casi completos para encontrar problemas y defectos arquitectónicos significativos que impliquen una revisión sustancial. En segundo lugar, los equipos de seguridad simplemente no tienen la capacidad de realizar revisiones de seguridad cuando las implementaciones son frecuentes. En muchas organizaciones, la seguridad y el cumplimiento es un cuello de botella significativo para llevar los sistemas de "desarrollo completo.ª la vida. Involucrar a profesionales de Infosec en todo el proceso de desarrollo también tiene el efecto de mejorar la comunicación y el flujo de información, un objetivo fundamental de DevOps.

Cuando la creación de seguridad en el software forma parte del trabajo diario de los desarrolladores, y cuando los equipos de infosec proporcionan herramientas, capacitación y soporte para facilitar a los desarrolladores hacer lo correcto, el rendimiento de la entrega mejora. Además, esto tiene un impacto positivo en la seguridad. Descubrieron que las empresas de alto rendimiento gastaban un 50 % menos de tiempo en solucionar problemas de seguridad que las de bajo rendimiento. En otras palabras, en lugar de preocuparse al final, al incorporar la seguridad en su trabajo diario, terminaron dedicando muchísimo menos tiempo a abordar esos problemas.

7.2.2. Caso de estudio

Hay una diferencia crucial entre aplicar mecanismos de seguridad y estar seguro.

El costo en recursos de tiempo, capacitación y dinero que se deben invertir para contratar una empresa tercera para realizar tareas de este estilo, es altísimo. Más cuando se posee la posibilidad de incorporar a profesionales con perspectivas de seguridad al equipo desde las primeras iteraciones del proyecto, o incluso capacitar a los desarrolladores principales para ir aplicando técnicas de seguridad incrementalmente.

Aún así, es difícil adquirir este pensamiento lateral. Un ejemplo con el caso de estudio puede observarse en el siguiente comentario[132] luego de que un miembro activo de la comunidad provee la posibilidad de integrar un nuevo setup de fuzzing, la respuesta de unos de los desarrolladores de Solidity fue:

'Estaría encantado de ver tal integración, pero hay que tener en cuenta que fuzzear el compilador no va a encontrar ningún problema crítico. [..] En el mejor de los casos, puede encontrar problemas de memoria como desreferencia de punteros nulos.'

En este ejemplo puntual están minimizando un problema de manejo de memoria porque su impacto *"no es más que la detención abrupta del compilador"*.

Abstrayendo esto mismo, y mirando al compilador como un servicio, no les parece menor que su servicio se detenga inesperadamente en medio de su ejecución. Lo mismo se podría decir para la discusión respecto al problema reportado de la manipulación de las variables de entorno `($EDITOR)`:

'El código respectivo nunca se ejecutará en escenarios de producción y solo se ejecutará de forma interactiva; no tiene sentido usarlo automáticamente o sin la interacción del usuario.'

No trabajar un potencial problema de seguridad, que además es trivial de solucionar, con la excusa de que nunca ocurrirá el caso en el que pueda ser abusado, es dejar código propenso a que sea explotado en el futuro.

Esto valida que incluso los equipos que desean aplicar seguridad, por más que trabajen en conjunto con expertos en el área, no van a modificar su perspectiva de un día para el otro.

Es inminente la necesidad de concientizar y entender a la seguridad como algo que debería pertenecer al ciclo del desarrollo del software.

Apéndice A

Coinspect Audit Recheck Detalles

A.SOL-001 —SOLUCIONADO

Amplio incremento, descripto a razón O(n^2) en la salida de mensajes del compilador por advertencias forzadas / errores.

Se reportó que el compilador emitía sun sinnúmero de mensajes.

Recomendaciones previas

- No imprimir la línea en cuestión en caso de que esta sea larga a la hora de mostrar una advertencia o error.
- Setear un número máximo de advertencias o errores a reportar.

Estado actual

- La línea ya no se muestra completa.

```
    sol_001.sol:7:758: Warning: Use of unary + is deprecated.
    ... +x ...
```

- Cantidad de mensajes mostrados truncados a 256.

```
        Warning: There are more than 256 warnings. Ignoring the rest.
```

Código utilizado para probar nuevamente este caso

```
contract TryMe {
    function warns() {
    uint x;
    +x;+x;+x;+x;...+x;+x;
    }
```

A.SOL-002 —SOLUCIONADO

Amplio incremento, descripto a razón O(n^2) en la salida de mensajes del compilador mediante duplicados en los nombres de funciones.

Similarmente al anterior, se reportó que el compilador emitía sun sinnúmero de mensajes.

Recomendaciones previas

- Reportar un sólo error para todos los duplicados de una misma función.

Estado actual

- Duplicados truncados a las primeras 32 ocurrencias.

```
sol_002.sol:4:2: Error: Function with same name and arguments defined
    twice. Truncated from 4998 to the first 32 occurrences.
 ... function asdyrtuiwekjasdsagfkhjsada ...
    dasdasdyrtuiwekjasdsagfkhjsadasd(); ...
    ^-------------------------------------------------------------------------^

sol_011.sol:4:2394: Other declaration is here:
 ... function asdyrtuiwekjasdsagfkhjsada ...
    dasdasdyrtuiwekjasdsagfkhjsadasd(); ...
    ^-------------------------------------------------------------------------^

sol_002.sol:4:4786: Other declaration is here:
 ... function asdyrtuiwekjasdsagfkhjsada ...
    dasdasdyrtuiwekjasdsagfkhjsadasd(); ...
    ^-------------------------------------------------------------------------^
```

Código utilizado para probar nuevamente este caso

```
contract e {
    function e();function e();...function e();
}
```

A.SOL-003 —SOLUCIONADO

Incremento en el uso de RAM por ciclos entre constantes.

Se reportó que el compilador consume una gran cantidad de RAM detectando ciclos por referencias cruzadas.

Recomendaciones previas

- Reescribir el algoritmo de búsqueda de ciclos para evitar copiar estados mientras se recorre, o establecer un límite en la profundidad de las referencias constantes.

Estado actual

- El tiempo de compilación de ahora es el esperado.
- No hay notables incrementos en el consumo de RAM.

```
sol_003.sol:4:448: Warning: This declaration shadows an existing declaration.
... a constant a=b ...
    ^------------^
sol_003.sol:3:2: The shadowed declaration is here:
        contract a {}
        ^-----------^

sol_003.sol:4:23221: Warning: This declaration shadows an existing declaration.
 ... a constant XX=XY ...
    ^--------------^
sol_003.sol:4:2: The shadowed declaration is here:
        contract XX { a constant A=B; a con ... t ZZY=ZZZ; a constant ZZZ=a(0x00
            );}
```

Código utilizado para probar nuevamente este caso

```
contract a {}
contract XX { a constant A=B; a constant B=C; a constant C=D; a constant D=E;
   a constant E=F; a constant F=G; a constant G=H; a constant H=I; a constant I
       =J;
   a constant J=K; a constant K=L; a constant L=M; a constant M=N; a constant N
       =O;
   a constant O=P; a constant P=Q; a constant Q=R; a constant R=S; a constant S
       =T;
```

```
6         a constant T=U; a constant U=V;...a constant ZZW=ZZX; a constant ZZX=ZZY;
7         a constant ZZY=ZZZ;
8          a constant ZZZ=a(0x00);
9     }
```

A.SOL-004 —SOLUCIONADO

Incremento en el uso de RAM debido a pasos exponenciales en búsquedas de ciclos entre constantes.

Similar al anterior, explotando el mismo problema con una leve variación.

Recomendaciones previas

- Reescribir el algoritmo de búsqueda de ciclos para evitar copiar estados mientras se recorre, o establecer un límite en la profundidad de las referencias constantes.

Estado actual

- Tiempo de compilación actual reducido: archivo de 64k con un patrón como este toma 15 minutos en finalizar.
- No se ha encontrado un incremento en el uso de RAM.

Código utilizado para probar nuevamente este caso

```
1     contract XX {
2         int constant v0a=v1a+v1b; int constant v0b=v1a+v1b; int constant v1a=v2a+
              v2b;int constant v1b=v2a+v2b;int constant v2a=v3a+v3b;int constant v2b=
              v3a+v3b...int constant v9999b=v10000a+v10000b;
3
4         int constant v10000a = 0;
5         int constant v10000b = 0;
6         }
```

A.SOL-005 —SIN SOLUCIÓN

No se halla un límite en el costo del gas cuando se borran arreglos dinámicos.

La operación de borrado en arreglos dinámicos genera bytecode para borrar los elementos uno por uno. Esto puede producir excepciones del estilo .ºut of gas".

Recomendaciones previas

- Advertir al usuario de las implicancias de borrar arreglos de tamaño dinámico.

Estado actual

- Reportado en el *issue* #3324[200].
- No se produce ningún mensaje de advertencia.

Update: Este problema será solucionado con un cambio fundamental en la versión `v0.6.0`.

Teniendo un contrato con un arreglo dinámico de solamente 200 items, y una función que intenta borrarlo, los costos de llamar a esa misma función están cercanos al millón de gas para ambas transacción y ejecución. Excepciones .ºut of gas"son inminentes para casos superiores.

Código utilizado para probar nuevamente este caso

```
    contract ArrayTest {
        uint[] public dynArr;
        constructor () public {
            dynArr.length = 200;
            dynArr[0]=0;
            dynArr[1]=1;
            dynArr[2]=2;
            dynArr[3]=3;
            dynArr[198]=198;
            dynArr[199]=199;
        }
        function delArr() public {
            delete dynArr;
        }
    }
```

A.SOL-006 —SOLUCIONADO

No se reportan las llamadas duplicadas al constructor de la clase de la que se extendió el contrato (*super-constructor*).

Solidity provee dos maneras diferentes de usar llamadas al constructor de una clase de la que se extiende (`super constructor`). El compilador permite utilizar ambas al mismo tiempo, ignorando la primera opción sin producir error o emitir una advertencia.

Recomendaciones previas

- Quitarle al usuario la posibilidad de definir dos llamadas de este estilo, o advertir al usuario si una de ellas hace que se ignore la otra.

Estado actual

- Una advertencia se muestra cuando el constructor base es utilizado más de una vez.

```
sol_006.sol:8:25: Warning: Base constructor arguments given twice.
function P2(uint v) P1(40) public {
                    ^----^
sol_006.sol:7:16: Second constructor call is here:
contract P2 is P1(20) {
               ^----^
```

Código utilizado para probar nuevamente este caso

```
    contract P1 {
        function P1(uint v) public {}
     }
     contract P2 is P1(20) {
        function P2(uint v) P1(40) public {}
     }
```

A.SOL-007 —SIN SOLUCIÓN

Asignación múltiple con valores a izquierda (LValues) vacíos propenso a errores.

Solidity permite la asignación de múltiples valores al mismo tiempo, y algunos de los valores del lado izquierdo pueden ser omitidos.

Ejemplo de un LValue posicional vacío: `var (,y,) = (V0,V1,V2);` Ejemplo de un LValue al final vacío: `var (g1,) = (return1(),return2());` Ejemplo de un LValue al principio vacío: `var (,d1, d2) = (V0,V1,V2,V3,V4,V5,V6);`

Hay un caso que es propenso a errores, cuando la cantidad de LValues y RValues (valores del lado derecho) difieren. No está claro como se está realizando la asignación.

Ejemplo: var (, ,e2, e3) = (V0,V1,V2,V3,V4);. e2 = v3 y e3 = v4, incluso si esto supone e2=v2 y e3=v3.

Recomendaciones previas

- En caso de que sean LValues vacíos principio/fin, no se debería poder especificar ningún otro valor vacío para otro LValue. Para valores vacíos de principio/fin, marcarlos con tres puntos para diferenciarlos de los valores posicionales, como en el siguiente ejemplo: `var ( ... , e3, e4) = (V0,V1,V2,V3,V4);`

Estado actual

- Reportado en el *issue* #3314[193].
- Este tipo de asignaciones todavía es realizable.
- Se muestra una advertencia mostrando una cantidad diferente de componentes.
- No hay manera alternativa de expresar valores a izquierda posicionales con el método recomendado `"..."`.

Update: este *issue*[249] ha sido solucionado, y en la versión v0.5.0 esta situación será retornada como un error.

```
./sol_007.sol:19:9: Warning: Different number of components on the left hand
    side (4) than on the right hand side (5).
var ( , , e2, e3) = (v0, v1, v2, v3, v4);
        ^-------------------------------------^
```

Código utilizado para probar nuevamente este caso

```
contract Values {
    uint v0;
    uint v1;
    uint v2;
    uint v3;
    uint v4;
    uint f2;
    uint f3;
    constructor () public {
        v0 = 0;
        v1 = 1;
        v2 = 2;
        v3 = 3;
        v4 = 4;
        var (,, e2, e3) = (v0, v1, v2, v3, v4);
     f2 = e2; //3
     f3 = e3; //4
    }
 }
```

A.SOL-008 —SOLUCIONADO

Incremento en el uso de ciclos de procesamiento utilizando grandes números literales del tipo `bignum`.

El procesamiento de literales numéricos de precisión arbitraria consume grandes cantides de uso de CPU.

Recomendaciones previas

- Limitar el tamaño de literales numéricos.

Estado actual

- Litetales numéricos de precisión arbitraria se encuentran limitados.

```
sol_008.sol:11:13: Error: Type int_const 1000...(71 digits omitted)...0000 is
    not implicitly convertible to expected type uint256.
c = 1e78;
^--^
```

Código utilizado para probar nuevamente este caso

```
contract BIGNUMTEST {
    constructor() public {
        uint256 c;
        uint256 d;
        c = 1e77; //OK
        d = 2e76; //OK
        c = c ** d; //OK
        c = 1e78; //ERR
    }
}
```

A.SOL-009 —SOLUCIONADO

Amplio incremento en la salida de mensajes al usar grandes literales numéricos del tipo `bignum`.

Los errores presentados correspondientes a números con precisión arbitraria muestran todos sus dígitos.

Recomendaciones previas

- Achicar constantes literales reemplazando dígitos intermedios por `"..."` a la hora de imprimir errores en la salida estándar de errores (`stderr`, por ejemplo 1000...000).
- Reducir el número de advertencias/errores escritos a `stderr`.

Estado actual

- La salida ahora reemplaza dígitos intermedios con `"..."`.

```
        sol_009.sol:11:13: Error: Type int_const 1000...(71 digits omitted)
            ...0000 is not implicitly convertible to expected type uint256.
        c = 1e78;
            ^--^
```

- La cantidad de advertencias/errores se encuentra truncada a 256 mensajes.

Código utilizado para probar nuevamente este caso

```
contract BIGNUMTEST {
    constructor() public {
        uint256 c;
        uint256 d;
        c = 1e77; //OK
        d = 2e76; //OK
        c = c ** d; //OK
        c = 1e78; //ERR
    }
}
```

A.SOL-010 —SIN SOLUCIÓN

Es muy fácil confundir la manera en que funcionan las sobre-escrituras (`overrides`).

Hacer una sobre-escritura de una función sólo funciona cuando las firmas (signatures) de cada función son exactamente iguales. Escenarios reales son propensos a errores a código malintencionado.

Recomendaciones previas

- Modificar el lenguaje de Solidity para que se requiera la palabra reservada `override` como un modificador para la definición de funciones en casos como estos. El compilador debería generar un error al intentar compilar una función con este modificador si no existe una función padre a la que sobre-escribir.

Estado actual

- Reportado en el `issue` #2563[198].
- No se emiten advertencias.
- No existe la palabra reservada `override`.

Update: hay una solución planificada para esta situación en la versión `v0.6.0`.

Código utilizado para probar nuevamente este caso

```
 library String {
     function equals(string memory _a, string memory _b) internal pure returns (
         bool) {
         bytes memory a = bytes(_a);
         bytes memory b = bytes(_b);
     if (a.length != b.length)
             return false;
     for (uint i = 0; i < a.length; i++)
             if (a[i] != b[i])
                     return false;
             return true;
           }
     }
     contract Override {
         using String for string;
         constructor () public {}

         function overrideMe(int i) public pure {
             i = i + 1;
         }
     function overrideMeToo(string s) public pure {
             s = "zeppelin";
         }
     }
     contract TryOverride is Override {
     constructor () public {}
     function overrideMe(uint u) public pure {
               u = 1337;
           }
     function overrideMeToo(String s) public pure {
               String ss;
               String override;
               s = ss;
               override = s;
           }
         }
```

Bibliografía

[1] J.A.N. Lee. *The Anatomy of a Compiler*. Computer Science Series - Van Nostrand Reinhold Company. Van Nostrand Reinhold Company, 1974. ISBN: 9780442247331. URL: https://books.google.com.ar/books?id=jCYuAAAAIAAJ.

[2] J.P. Tremblay y P.G. Sorenson. *The theory and practice of compiler writing*. McGraw-Hill Computer Sciences Series. McGraw-Hill Higher Education, 1985. ISBN: 9780070651616. URL: https://books.google.com.ar/books?id=MacmAAAAMAAJ.

[3] A.V. Aho y col. *Compiladores: principios, técnicas y herramientas*. Compiladores: principios, técnicas y herramientas. Pearson Educación, 1990. ISBN: 9789684443334. URL: https://books.google.com.ar/books?id=yG6qJBAnE9UC.

[4] Barton P. Miller, Louis Fredriksen y Bryan So. «An Empirical Study of the Reliability of UNIX Utilities». En: *Commun. ACM* 33.12 (dic. de 1990), págs. 32-44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: http://doi.acm.org/10.1145/96267.96279.

[5] K. Slonneger y B.L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. NATO Asi Series A. Life Sciences; 282. Addison-Wesley Publishing Company, 1995. ISBN: 9780201656978. URL: https://books.google.com.ar/books?id=HIRQAAAAMAAJ.

[6] S. McConnell. *Rapid Development: Taming Wild Software Schedules*. Best Practices Series. Microsoft Press, 1996. ISBN: 9781556159008. URL: https://books.google.com.ar/books?id=qM4Yzf8K9hwC.

[7] Abdulazeez S. Boujarwah y Kassem Saleh. «Compiler test case generation methods: a survey and assessment». En: *Information & Software Technology* 39 (1997), págs. 617-625.

[8] P. Terry. *Compilers and Compiler Generators: An Introduction with C++*. ITCP Computer Science Series. International Thomson Computer Press, 1997. ISBN: 9781850322986. URL: https://books.google.com.ar/books?id=p%5C_rFQgAACAAJ.

[9] William M. McKeeman. «Differential Testing for Software». En: *DIGITAL TECHNICAL JOURNAL* 10.1 (1998), págs. 100-107.

[10] R. Morgan. *Building an optimizing compiler*. Butterworth-Heinemann, 1998. URL: https://books.google.com.ar/books?id=v1maoAEACAAJ.

[11] Jim Highsmith y Alistair Cockburn. «Agile Software Development: The Business of Innovation». En: *IEEE Computer* 34 (2001), págs. 120-122.

[12] Y.N. Srikant y P. Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2002. ISBN: 9781420040579. URL: https://books.google.com.ar/books?id=0K%5C_jIsgyNpoC.

[13] A.W. Appel y M. Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, 2004. ISBN: 9780521607650. URL: https://books.google.com.ar/books?id=A3yqQuLW5RsC.

[14] Greg Hoglund y Gary McGraw. *Exploiting Software: How to Break Code*. Pearson Higher Education, 2004. ISBN: 0201786958.

[15] S. McConnell. *Code Complete*. Developer Best Practices Series. Microsoft Press, 2004. ISBN: 9780735619678. URL: https://books.google.com.ar/books?id=QnghAQAAIAAJ.

[16] Gary McGraw. «Software Security». En: *Security & Privacy, IEEE* 2 (abr. de 2004), págs. 80-83. DOI: 10.1109/MSECP.2004.1281254.

[17] Brad Appleton, Steve Berczuk y Robert. Cowham. *The Agile Difference for SCM*. 2005. URL: https://www.cmcrossroads.com/article/agile-difference-scm.

[18] B. Arkin, S. Stender y G. McGraw. «Software penetration testing». En: *Security & Privacy, IEEE* 3 (feb. de 2005), págs. 84-87. DOI: 10.1109/MSP.2005.23.

[19] Alexander Kossatchev y Mikhail Posypkin. «Survey of Compiler Testing Methods». En: *Programming and Computer Software* 31 (ene. de 2005), págs. 10-19. DOI: 10.1007/s11086-005-0008-6.

[20] Scott W. Ambler. *Why Agile Software Development Techniques Work: Improved Feedback*. 2006. URL: http://www.ambysoft.com/essays/whyAgileWorksFeedback.html.

[21] Mark Dowd, John McDonald y Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 2006. ISBN: 0321444426.

[22] Michael Howard y Steve Lipner. *The Security Development Lifecycle*. Vol. 34. Jun. de 2006. ISBN: 0735622140. DOI: 10.1007/s11623-010-0021-7.

[23] S. McConnell. *Software Estimation: Demystifying the Black Art*. Developer Best Practices. Pearson Education, 2006. ISBN: 9780735637030. URL: https://books.google.com.ar/books?id=U5VCAwAAQBAJ.

[24] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006. ISBN: 0321356705.

[25] L.M. Pinho y M.G. Harbour. *Reliable Software Technologies – Ada-Europe 2006: 11th Ada-Europe International Conference on Reliable Software Technologies, Porto, Portugal, June 5-9, 2006, Proceedings*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006. ISBN: 9783540346647. URL: https://books.google.com.ar/books?id=ci0GCAAAQBAJ.

[26] Chris Wysopal y col. *The Art of Software Security Testing: Identifying Software Security Flaws (Symantec Press)*. Addison-Wesley Professional, 2006. ISBN: 0321304861.

[27] A. Meduna. *Elements of Compiler Design*. Computer science. Computer engineering. Computing. Taylor & Francis, 2007. ISBN: 9781420063233. URL: https://books.google.com.ar/books?id=kPIJaNyk404C.

[28] T. Reps, M. Sagiv y J. Bauer. *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007. ISBN: 9783540713227. URL: https://books.google.com.ar/books?id=xIOIs%5C_7GCB8C.

[29] Flash Sheridan. «Practical testing of a C99 compiler using output comparison». En: *Softw., Pract. Exper.* 37 (nov. de 2007), págs. 1475-1488. DOI: 10.1002/spe.812.

[30] B. Turner. *Random Program Generator*. 2007. URL: http://sites.google.com/site/brturn2/randomcprogramgenerator.

[31] Julia H. Allen y col. *Software Security Engineering: A Guide for Project Managers (The SEI Series in Software Engineering)*. 1.ª ed. Addison-Wesley Professional, 2008. ISBN: 032150917X, 9780321509178.

[32] Eric Eide y John Regehr. «Volatiles Are Miscompiled, and What to Do About It». En: *Proceedings of the 8th ACM International Conference on Embedded Software*. EMSOFT '08. Atlanta, GA, USA: ACM, 2008, págs. 255-264. ISBN: 978-1-60558-468-3. DOI: 10.1145/1450058.1450093. URL: http://doi.acm.org/10.1145/1450058.1450093.

[33] C. Jones. *Applied Software Measurement: Global Analysis of Productivity and Quality*. McGraw Hill professional. McGraw-Hill Education, 2008. ISBN: 9780071643863. URL: https://books.google.com.ar/books?id=mj7yQiQEusUC.

[34] Bojan. *A new fascinating Linux kernel vulnerability*. 2009. URL: https://isc.sans.edu/forums/diary/A+new+fascinating+Linux+kernel+vulnerability/6820/.

[35] R. Gupta. *Compiler Construction: 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Advanced research in computing and software science. Springer, 2010. ISBN: 9783642119699. URL: https://books.google.com.ar/books?id=0gtXj16MtssC.

[36] Michael Howard, David LeBlanc y John Viega. *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. 1.ª ed. New York, NY, USA: McGraw-Hill, Inc., 2010. ISBN: 0071626751, 9780071626750.

[37] K. Cooper y L. Torczon. *Engineering a Compiler*. Elsevier Science, 2011. ISBN: 9780080916613. URL: https://books.google.com.ar/books?id=%5C_tgh4bgQ6PAC.

[38] B. Dasnois. *HaXe 2 Beginner's Guide*. Community experience distilled. Packt Publishing, 2011. ISBN: 9781849512572. URL: https://books.google.com.ar/books?id=sX-lOUrMJZ4C.

[39] Tobias Klein. *A Bug Hunter's Diary: A Guided Tour Through the Wilds of Software Security*. 1st. San Francisco, CA, USA: No Starch Press, 2011. ISBN: 1593273851, 9781593273859.

[40] R. Mak. *Writing Compilers and Interpreters: A Software Engineering Approach*. Wiley, 2011. ISBN: 9781118079737. URL: https://books.google.com.ar/books?id=EUs94gRZHEUC.

[41] Y. Su y S.Y. Yan. *Principles of Compilers: A New Approach to Compilers Including the Algebraic Method*. Springer Berlin Heidelberg, 2011. ISBN: 9783642208355. URL: https://books.google.com.ar/books?id=u5wkJ4R03d4C.

[42] Xuejun Yang y col. «Finding and Understanding Bugs in C Compilers». En: *SIGPLAN Not.* 46.6 (jun. de 2011), págs. 283-294. ISSN: 0362-1340. DOI: 10.1145/1993316.1993532. URL: http://doi.acm.org/10.1145/1993316.1993532.

[43] L. Bolc. *The Design of Interpreters, Compilers, and Editors for Augmented Transition Networks*. Symbolic Computation. Springer Berlin Heidelberg, 2012. ISBN: 9783642821226. URL: https://books.google.com.ar/books?id=xdX6CAAAQBAJ.

[44] D. Grune y col. *Modern Compiler Design*. Springer New York, 2012. ISBN: 9781461446996. URL: https://books.google.com.ar/books?id=zkpFTBtK7a4C.

[45] Christian Holler, Kim Herzig y Andreas Zeller. «Fuzzing with Code Fragments». En: ago. de 2012.

[46] S.P. Midkiff. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Synthesis digital library of engineering and computer science. Morgan & Claypool Publishers, 2012. ISBN: 9781608458417. URL: https://books.google.com.ar/books?id=Y7Ocm4IC6W8C.

[47] E. Ries y J.S. Julián. *El método Lean Startup: Cómo crear empresas de éxito utilizando la innovación continua*. Biblioteca empresarial Deusto. Deusto, 2012. ISBN: 9788423409495. URL: https://books.google.com.ar/books?id=v3%5C_C4yd-wR4C.

[48] Robert W. Sebesta. *Concepts of Programming Languages*. 10th. Pearson, 2012. ISBN: 0273769103, 9780273769101.

[49] Not So Secure. *What to/not to expect from pentest*. 2012. URL: https://www.notsosecure.com/what-tonot-to-expect-from-pentest/.

[50] H. Seidl, R. Wilhelm y S. Hack. *Compiler Design: Analysis and Transformation*. SpringerLink : Bücher. Springer Berlin Heidelberg, 2012. ISBN: 9783642175480. URL: https://books.google.com.ar/books?id=zfKYi6O%5C_9WEC.

[51] M. Summerfield. *Programming in Go: Creating Applications for the 21st Century*. Developer's Library. Pearson Education, 2012. ISBN: 9780132764094. URL: https://books.google.com.ar/books?id=HW-5SZ1HKusC.

[52] Jason Creasey. *Will Vulnerability Assessments & Penetration Testing Find the Security Weaknesses in Your Systems?* 2013. URL: https://crest-approved.org/wp-content/uploads/CREST-Penetration-Testing-Guide.pdf.

[53] KRYPSYS. *Will Vulnerability Assessments & Penetration Testing Find the Security Weaknesses in Your Systems?* 2013. URL: https://www.krypsys.com/news/will-vulnerability-assessments-and-penetration-testing-find-all-the-security-vulnerabilities-in-your-systems.

[54] James Ransome y Anmol Misra. *Core Software Security: Security at the Source*. Boston, MA, USA: Auerbach Publications, 2013. ISBN: 1466560959, 9781466560956.

[55] Vu Le, Mehrdad Afshari y Zhendong Su. «Compiler Validation via Equivalence Modulo Inputs». En: *ACM SIGPLAN Notices* 49 (jun. de 2014). DOI: 10.1145/2594291.2594334.

[56] Eriko Nagai, Atsushi Hashimoto y Nagisa Ishiura. «Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions». En: *IPSJ Transactions on System LSI Design Methodology* 7 (ene. de 2014), págs. 91-100. DOI: 10.2197/ipsjtsldm.7.91.

[57] Adam Shostack. *Threat Modeling: Designing for Security*. 1st. Wiley Publishing, 2014. ISBN: 1118809998, 9781118809990.

[58] A.A.A. Donovan y B.W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional Computing Series. Pearson Education, 2015. ISBN: 9780134190563. URL: https://books.google.com.ar/books?id=SJHvCgAAQBAJ.

[59] An Lam y col. «Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N)». En: nov. de 2015, págs. 476-481. DOI: 10.1109/ASE.2015.73.

[60] Janet Leon. *The True Cost of a Software Bug: Part One*. 2015. URL: blog.celerity.com/the-true-cost-of-a-software-bug.

[61] J. McCurdy. *Haxe Game Development Essentials*. Packt Publishing, 2015. ISBN: 9781785286919. URL: https://books.google.com.ar/books?id=mfaoCwAAQBAJ.

[62] Gary McGraw. «Software Security and the Building Security in Maturity Model (BSIMM)». En: *J. Comput. Sci. Coll.* 30.3 (ene. de 2015), págs. 7-8. ISSN: 1937-4771. URL: http://dl.acm.org/citation.cfm?id=2675327.2675329.

[63] Junjie Chen y col. «An empirical comparison of compiler testing techniques». En: mayo de 2016, págs. 180-190. DOI: 10.1145/2884781.2884878.

[64] Michael Felderer y col. «Chapter One - Security Testing: A Survey». En: ed. por Atif Memon. Vol. 101. Advances in Computers. Elsevier, 2016, págs. 1-51. DOI: https://doi.org/10.1016/bs.adcom.2015.11.003. URL: http://www.sciencedirect.com/science/article/pii/S0065245815000649.

[65] Michael Felderer y col. «Security Testing: A Survey». En: mar. de 2016, págs. 1-51. DOI: 10.1016/bs.adcom.2015.11.003.

[66] A. Narayanan y col. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016. ISBN: 9780691171692. URL: https://books.google.com.ar/books?id=fW-YDwAAQBAJ.

[67] David Siegel. *Understanding The DAO Attack*. 2016. URL: https://www.coindesk.com/understanding-dao-hack-journalists.

[68] Chengnian Sun, Vu Le y Zhendong Su. «Finding compiler bugs via live code mutation». En: oct. de 2016, págs. 849-863. DOI: 10.1145/2983990.2984038.

[69] Peter Vessenes. *More Ethereum attacks: race-to-empty is the real deal*. 2016. URL: http://vessenes.com/more-ethereum-attacks-race-to-empty-is-the-real-deal/.

[70] Collis Aventinus. *Billetera Parity Multisig hackeada, o ¿cómo es?* 2017. URL: https://es.cointelegraph.com/news/parity-multisig-wallet-hacked-or-how-come.

[71] Osbert Bastani y col. «Synthesizing Program Input Grammars». En: *ACM SIGPLAN Notices* 52 (jun. de 2017), págs. 95-110. DOI: 10.1145/3140587.3062349.

[72] Jeremy Bennett. *Security Enhanced Compilers*. 2017. URL: https://www.embecosm.com/2017/02/20/security-enhanced-compilers/.

[73] Junjie Chen y col. «Learning to Prioritize Test Programs for Compiler Testing». En: mayo de 2017. DOI: 10.1109/ICSE.2017.70.

[74] Richard Chirgwin. *30 million below Parity: Ethereum wallet bug fingered in mass heist*. 2017. URL: https://www.theregister.co.uk/2017/07/20/us30_million_below_parity_ethereum_bug_leads_to_big_coin_heist/.

[75] Min-je Choi y col. «End-to-End Prediction of Buffer Overruns from Raw Source Code via Neural Memory Networks». En: ago. de 2017, págs. 1546-1553. DOI: 10.24963/ijcai.2017/214.

[76] Patrice Godefroid, Hila Peleg y Rishabh Singh. «Learn and Fuzz: Machine learning for input fuzzing». En: oct. de 2017, págs. 50-59. DOI: 10.1109/ASE.2017.8115618.

[77] Kihong Heo, Hakjoo Oh y Kwangkeun Yi. «Machine-Learning-Guided Selectively Unsound Static Analysis». En: mayo de 2017, págs. 519-529. DOI: 10.1109/ICSE.2017.54.

[78] Alyssa Hertig. *Ethereum client bug freezes user funds as fallout remains uncertain*. 2017. URL: https://www.coindesk.com/ethereum-client-bug-freezes-user-funds-fallout-remains-uncertain/.

[79] Xuan Huo y Ming Li. «Enhancing the Unified Features to Locate Buggy Files by Exploiting the Sequential Nature of Source Code». En: ago. de 2017, págs. 1909-1915. DOI: 10.24963/ijcai.2017/265.

[80] Ugur Koc y col. «Learning a classifier for false positive error reports emitted by static code analysis tools». En: jun. de 2017, págs. 35-42. DOI: 10.1145/3088525.3088675.

[81] Manos Koukoutos y col. «On Repair with Probabilistic Attribute Grammars». En: (jul. de 2017).

[82] Jim Manicode. *Secure Software Development Life Cycle*. 2017. URL: https://www.youtube.com/watch?v=M7qMP3C5bkU/.

[83] Alex McPeak. *What's the True Cost of a Software Bug?* 2017. URL: crossbrowsertesting.com/blog/development/software-bug-cost.

[84] T.Æ. Mogensen. *Introduction to Compiler Design*. Undergraduate Topics in Computer Science. Springer International Publishing, 2017. ISBN: 9783319669663. URL: https://books.google.com.ar/books?id=puo7DwAAQBAJ.

[85] Emmanuel Okon. «The New Trends in Compiler Analysis and Optimizations». En: *International Journal of computer trends and technology* 46 (mayo de 2017).

[86] Jordan Pearson. *A hacker allegedly stole $32 million in Ethereum*. 2017. URL: https://motherboard.vice.com/en_us/article/zmvkke/this-is-not-a-drill-a-hacker-allegedly-stole-dollar32-million-in-ethereum.

[87] Junjie Wang y col. «Skyfire: Data-Driven Seed Generation for Fuzzing». En: mayo de 2017, págs. 579-594. DOI: 10.1109/SP.2017.23.

[88] Qirun Zhang, Chengnian Sun y Zhendong Su. «Skeletal program enumeration for rigorous compiler testing». En: jun. de 2017, págs. 347-361. DOI: 10.1145/3062341.3062379.

[89] I. Adelekan. *Kotlin Programming By Example: Build real-world Android and web applications the Kotlin way*. Packt Publishing, 2018. ISBN: 9781788479783. URL: https://books.google.com.ar/books?id=OrZTDwAAQBAJ.

[90] Chris Cummins y col. «Compiler Fuzzing Through Deep Learning». En: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: ACM, 2018, págs. 95-105. ISBN: 978-1-4503-5699-2. DOI: 10.1145/3213846.3213848. URL: http://doi.acm.org/10.1145/3213846.3213848.

[91] N. Forsgren, J. Humble y G. Kim. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution Press, 2018. ISBN: 9781942788355. URL: https://books.google.com.ar/books?id=Kax-DwAAQBAJ.

[92] Nicole Forsgren. «Accelerate: State of DevOps, Strategies for a New Economy». En: 2018, págs. 58-60.

[93] Nicole Forsgren, Jez Humble y Gene Kim. *Accelerate: The Science of Lean Software and DevOps Building and Scaling High Performing Technology Organizations*. 1st. IT Revolution Press, 2018. ISBN: 1942788339, 9781942788331.

[94] T. Guney. *Hands-On Go Programming: Explore Go by solving real-world challenges*. Packt Publishing, 2018. ISBN: 9781789534870. URL: https://books.google.com.ar/books?id=bh9sDwAAQBAJ.

[95] S. Klabnik y C. Nichols. *The Rust Programming Language*. No Starch Press, 2018. ISBN: 9781593278519. URL: https://books.google.com.ar/books?id=lrgrDwAAQBAJ.

[96] Steve Klabnik y Carol Nichols. *The Rust Programming Language*. San Francisco, CA, USA: No Starch Press, 2018. ISBN: 1593278284, 9781593278281.

[97] Kaspersky Lab. *The human factor in IT security*. 2018. URL: https://www.kaspersky.com/blog/the-human-factor-in-it-security/.

[98] McAfee y CSIS. *Cybercrime hurting businesses to tune of $600 billion*. 2018. URL: https://www.mcafee.com/enterprise/de-de/about/newsroom/press-releases/press-release.html?news_id=20180221005206.

[99] J. Skeen y D. Greenhalgh. *Kotlin Programming: The Big Nerd Ranch Guide*. Big Nerd Ranch Guides. Pearson Education, 2018. ISBN: 9780135162361. URL: https://books.google.com.ar/books?id=Ol9qDwAAQBAJ.

[100] Y.N. Srikant y P. Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*. CRC Press, 2018. ISBN: 9781420043839. URL: https://books.google.com.ar/books?id=1kqAv-uDEPEC.

[101] The Economist Intelligence Unit. *The cyber chasm: How the disconnect between the c-suite and security endangers the enterprise*. 2018. URL: https://eiuperspectives.economist.com/technology-innovation/cyber-chasm-how-disconnect-between-c-suite-and-security-endangers-enterprise-0/infographic/cyber-attacks.

[102] Kevin Beaver. *What to Expect during Your next Penetration Test*. 2019. URL: https://www.specopssoft.com/blog/what-to-expect-during-your-next-penetration-test.

[103] Ethereum Foundation. *Solidity: Read the docs*. 2019. URL: https://solidity.readthedocs.io/.

[104] Ledger. «How It All Began: A Brief History On Bitcoin & Cryptocurrencies». En: (2019). URL: https://www.ledger.com/how-it-all-began-a-brief-history-of-bitcoin-cryptocurrencies/.

[105] C. Matzinger. *Hands-On Data Structures and Algorithms with Rust: Learn programming techniques to build effective, maintainable, and readable code in Rust 2018*. Packt Publishing, 2019. ISBN: 9781788991490. URL: https://books.google.com.ar/books?id=gYKFDwAAQBAJ.

[106] OWASP. *Application Security Verification StandardSoftware Assurance Maturity Model*. 2019. URL: https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project.

[107] OWASP. *Software Assurance Maturity Model*. 2019. URL: https://www.opensamm.org/.

[108] R. Sharma y V. Kaihlavirta. *Mastering Rust: Learn about memory safety, type system, concurrency, and the new features of Rust 2018 edition, 2nd Edition*. Packt Publishing, 2019. ISBN: 9781789341188. URL: https://books.google.com.ar/books?id=lGSGDwAAQBAJ.

[109] Martin White y col. «Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities». En: feb. de 2019, págs. 479-490. DOI: 10.1109/SANER.2019.8668043.

[110] URL: https://github.com/ethereum/serpent/tree/ad53fa2a8a496448d58ef9137959b4a1e86b14d7.

[111] 101blockchains. *¿Qué es Ethereum?* URL: https://101blockchains.com/es/hyperledger-vs-corda-r3-vs-ethereum-la-guia/.

[112] 101blockchains. *Historia de la blockchain*. URL: https://101blockchains.com/es/historia-de-la-blockchain/.

[113] *A timeline of major crypto-exchange hacks*. URL: https://discover.ledger.com/hackstimeline/.

[114] *AFL*. URL: http://lcamtuf.coredump.cx/afl/.

[115] *American Fuzzy Lop*. URL: http://lcamtuf.coredump.cx/afl/.

[116] *Awesome Fuzzing*. URL: https://github.com/secfigo/Awesome-Fuzzing.

[117] *Awesome Static Analysis*. URL: https://github.com/mre/awesome-static-analysis#multiple-languages-1.

[118] *Binary Ninja*. URL: https://binary.ninja/.

[119] *Binnavi*. URL: https://github.com/google/binnavi.

[120] *Bus Factor, Wikipedia*. URL: https://en.wikipedia.org/wiki/Bus_factor.

[121] *C++ Core Guidelines - isocpp*. URL: https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md.

[122] *Capstone*. URL: https://github.com/aquynh/capstone.

[123] *Clang*. URL: http://clang.llvm.org/.

[124] *Clant-Tidy Checks - llvm.org*. URL: https://clang.llvm.org/extra/clang-tidy/checks/list.html.

[125] *CMake file for cpp-ethereum. Compiler settings*. URL: https://github.com/ethereum/solidity/blob/v0.4.24/cmake/EthCompilerSettings.cmake.

[126] *cmdlineTests.sh - v0.4.25 ethereum/solidity*. URL: https://github.com/ethereum/solidity/blob/v0.4.24/test/cmdlineTests.sh.

[127] *Code coverage for Ethereum Solidity, codecov.io*. URL: https://codecov.io/gh/ethereum/solidity/tree/9508406984c1e83e6ce571af4af593c33aed52e6.

[128] *Code coverage for Ethereum Solidity, codecov.io*. URL: https://codecov.io/gh/ethereum/solidity.

[129] *CodeCompass*. URL: https://github.com/Ericsson/CodeCompass.

[130] *CodeSonnar*. URL: https://www.grammatech.com/products/codesonar.

[131] *Código de ejemplo sin truncar - Mattaereal/Test.sol*. URL: https://gist.github.com/mattaereal/1f2e4a03b8d0cdecb0ed56974adefd0a.

[132] *Comment on oss-fuzz integration #5212*. URL: https://github.com/ethereum/solidity/issues/5212#issuecomment-429777270.

[133] *CommonSubexpressionEliminator.cpp at v0.4.24 · ethereum/solidity · GitHub*. URL: https://github.com/ethereum/solidity/blob/v0.4.24/libevmasm/CommonSubexpressionEliminator.cpp#L343.

[134] *Compilation for Security – Embecosm*. URL: https://www.embecosm.com/services/compilation-for-security/.

[135] *Consolidate inheritance rules*. URL: https://github.com/ethereum/solidity/projects/9.

[136] *Contrato malicioso verificado. Etherscan, Ropsten*. URL: https://ropsten.etherscan.io/address/0xb2a840de7569f4eee2e4677da65eae6890db4d6d#code.

[137] *Contributing - Running the compiler tests. Solidity Docs*. URL: https://github.com/ethereum/solidity/blob/v0.4.24/docs/contributing.rst#running-the-compiler-tests.

[138] *Contributors to ethereum/solidity March, GitHub*. URL: https://github.com/ethereum/solidity/graphs/contributors?from=2018-03-01&to=2019-07-17&type=c.

[139] *Contributors to ethereum/solidity, GitHub*. URL: https://github.com/ethereum/solidity/graphs/contributors.

[140] *Coverity*. URL: http://www.coverity.com/.

[141] *cppCheck*. URL: http://cppcheck.sourceforge.net/.

[142] *Cryptocurrency Market Capitalizations*. URL: https://coinmarketcap.com/.

[143] *Csmith's GitHub - Bugs reported*. URL: https://github.com/csmith-project/csmith/blob/master/BUGS_REPORTED.TXT.

[144] *DappRadar - Ranked list of blockchain dapps*. URL: https://dappradar.com/rankings/.

[145] *Decentralized computing network and app ecosystem*. URL: https://blockstack.org/.

[146] CVE details. *PHP Security Vulnerabilities*. URL: https://www.cvedetails.com/product/128/PHP-PHP.html?vendor_id=74.

[147] CVE details. *Python Security Vulnerabilities*. URL: https://www.cvedetails.com/product/18230/Python-Python.html?vendor_id=10210.

[148] CVE details. *Ruby lang Security Vulnerabilities*. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-7252/product_id-12215/Ruby-lang-Ruby.html.

[149] *ENV33-C. Do not call system(), Carnegie Mellon University*. URL: https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152177.

[150] *External dependencies - Installing Solidity. Read the docs.* URL: https://solidity.readthedocs.io/en/v0.4.24/installing-solidity.html#external-dependencies.

[151] Ethereum Foundation. *LLL Compiler Documentation.* URL: https://lll-docs.readthedocs.io/en/latest/lll_introduction.html.

[152] Ethereum Foundation. *Serpent's GitHub.* URL: https://github.com/ethereum/serpent.

[153] Ethereum Foundation. *Solidity Read the docs.* URL: https://solidity.readthedocs.io.

[154] *GCC Bug List Found by Random Testing.* URL: http://embed.cs.utah.edu/csmith/gcc-bugs.html.

[155] *gdb.* URL: https://www.gnu.org/s/gdb/.

[156] *Ghidra.* URL: https://ghidra-sre.org/.

[157] *Grammarinator.* URL: https://github.com/renatahodovan/grammarinator.

[158] *Honggfuzz.* URL: https://github.com/google/honggfuzz.

[159] *How Ethereum works.* URL: https://www.ethereum.org/learn/#how-ethereum-works.

[160] *How to write clean code? Lessons learnt from The Clean Code, Robert C. Martin.* URL: https://medium.com/mindorks/how-to-write-clean-code-lessons-learnt-from-the-clean-code-robert-c-martin-9ffc7aef870c.

[161] *Ida Pro.* URL: https://www.hex-rays.com/products/ida/.

[162] ISO & IEC. «Open STD c99». En: (). URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[163] *Infer.* URL: https://github.com/facebook/infer.

[164] *Install dependencies scruipt, Solidity GitHub.* URL: https://github.com/ethereum/solidity/blob/v0.4.24/scripts/install_deps.sh.

[165] *Introducing Clarity, a language for predictable smart contracts.* URL: https://blog.blockstack.org/introducing-clarity-the-language-for-predictable-smart-contracts/.

[166] *isolate_tests.py - v0.4.24 ethereum/solidity.* URL: https://github.com/ethereum/solidity/blob/v0.4.24/scripts/isolate%5C_tests.py.

[167] *Issue: Add a non-utf8 string type #5167.* URL: https://github.com/ethereum/solidity/issues/5167.

[168] *Issue: Add code coverage CI #2663.* URL: https://github.com/ethereum/solidity/issues/2663.

[169] *Issue: Add gas usage tests #5165.* URL: https://github.com/ethereum/solidity/issues/5165.

[170] *Issue: Add performance tests #5252.* URL: https://github.com/ethereum/solidity/issues/5252.

[171] *Issue: Add some guards around the editor execution in isoltest. #5159.* URL: https://github.com/ethereum/solidity/issues/5159.

[172] *Issue: Add workarounds for building against CVC4 on ArchLinux. #4767.* URL: https://github.com/ethereum/solidity/issues/4767.

[173] *Issue: Build failures with CVC4, e.g. on ArchLinux #4762.* URL: https://github.com/ethereum/solidity/issues/4762.

[174] *Issue: Crash when accessing a _slot of a function in assembly block #4710.* URL: https://github.com/ethereum/solidity/issues/4710.

[175] *Issue: Crash when accessing empty name variable slot #4707.* URL: https://github.com/ethereum/solidity/issues/4707.

[176] *Issue: Crash when array index value is too large #4716.* URL: https://github.com/ethereum/solidity/issues/4716.

[177] *Issue: Crash when calling a non callable type on a non primitive type double assignment #4711.* URL: https://github.com/ethereum/solidity/issues/4711.

[178] *Issue: Crash when converting fixed point type using ABIEncoderV2 #4715.* URL: https://github.com/ethereum/solidity/issues/4715.

[179] *Issue: Crash when converting signed rational using ABIEncoderV2 #4706.* URL: https://github.com/ethereum/solidity/issues/4706.

[180] *Issue: Crash when declaring external function with array of struct that possesses arrays #4713.* URL: https://github.com/ethereum/solidity/issues/4713.

[181] *Issue: Crash when trying to declare an already declared variable with the same name #4705.* URL: https://github.com/ethereum/solidity/issues/4705.

[182] *Issue: Crash when type not set for parameter function value #4709.* URL: https://github.com/ethereum/solidity/issues/4709.

[183] *Issue: Crash when type not set for parameter return value #4708.* URL: https://github.com/ethereum/solidity/issues/4708.

[184] *Issue: Crash when using assembly jump instruction inside a constructor or function with same name as contract #4712.* URL: https://github.com/ethereum/solidity/issues/4712.

[185] *Issue: Crash when using struct as external function parameter using ABIEncoderV2 #4714.* URL: https://github.com/ethereum/solidity/issues/4714.

[186] *Issue: Cropped link path in generated bytecode #4429.* URL: https://github.com/ethereum/solidity/issues/4429.

[187] *Issue: Fuzzer catches too many exceptions #4458.* URL: https://github.com/ethereum/solidity/issues/4458.

[188] *Issue: High CPU usage on conversion between numeric literal and others #4717.* URL: https://github.com/ethereum/solidity/issues/4717.

[189] *Issue: High CPU usage when using large variable names #4718.* URL: https://github.com/ethereum/solidity/issues/4718.

[190] *Issue: Improve running the tests"section #5166.* URL: https://github.com/ethereum/solidity/issues/5166.

[191] *Issue: Improving library names for the linker #579.* URL: https://github.com/ethereum/solidity/issues/579.

[192] *Issue: Internal compiler error on function name collision #4417.* URL: https://github.com/ethereum/solidity/issues/4417.

[193] *Issue: Less error prone tuple assignments #3314.* URL: https://github.com/ethereum/solidity/issues/3314.

[194] *Issue: Missing return statement on functions do not issue an error #4751.* URL: https://github.com/ethereum/solidity/issues/4751.

[195] *Issue: oss-fuzz integration #5212.* URL: https://github.com/ethereum/solidity/issues/5212.

[196] *Issue: Require 'override' and 'virtual' keyword #5424.* URL: https://github.com/ethereum/solidity/issues/5424.

[197] *Issue: Robot that posts CircleCI error messages as PR comments #5241.* URL: https://github.com/ethereum/solidity/issues/5241.

[198] *Issue: Shadowing of inherited state variables should be an error (override keyword) #2563.* URL: https://github.com/ethereum/solidity/issues/2563.

[199] *Issue: Should warn for unreachable code #2340.* URL: https://github.com/ethereum/solidity/issues/2340.

[200] *Issue: SOL-005 Unbounded gas cost when deleting dynamically sized arrays #3324.* URL: https://github.com/ethereum/solidity/issues/3324.

[201] *Issue: Some code cleanup #5168.* URL: https://github.com/ethereum/solidity/issues/5168.

[202] *Issue: Support Manjaro Linux distributions in dependencies script #4377.* URL: https://github.com/ethereum/solidity/issues/4377.

[203] *Issue: Tests to check if child classes don't modify superclass-behaviour in unwanted ways #501.* URL: https://github.com/OpenZeppelin/openzeppelin-solidity/issues/501.

[204] *Issue: Truncate linkerObject at the beginning, not end #3918.* URL: https://github.com/ethereum/solidity/pull/3918.

[205] *Issue: Warn about shadowing state variables #973.* URL: https://github.com/ethereum/solidity/issues/973.

[206] *IULIA Optimiser, Ethereum Solidity GitHub.* URL: https://github.com/ethereum/solidity/blob/v0.4.24/libjulia/optimiser/README.md.

[207] *lldb.* URL: https://lldb.llvm.org/.

[208] *LLVM Bug List Found by Random Testing.* URL: http://embed.cs.utah.edu/csmith/llvm-bugs.html.

[209] Microsoft. *Beware of compiler optimizations.* URL: https://wiki.sei.cmu.edu/confluence/display/c/MSC06-C.+Beware+of+compiler+optimizations.

[210] MiEthereum. *Vitalik Buterin.* URL: https://miethereum.com/vitalik-buterin/.

[211] *Mutan's GitHub.* URL: https://github.com/obscuren/mutan.

[212] *Ollydbg.* URL: http://www.ollydbg.de/.

[213] *OpenZeppelin GitHub: Improve RBAC role type #1090.* URL: https://github.com/OpenZeppelin/openzeppelin-contracts/issues/1090.

[214] *Override specifier (since C++11) - cppreference.com.* URL: https://en.cppreference.com/w/cpp/language/override.

[215] OWASP. *Insecure Compiler Optimization.* URL: https://www.owasp.org/index.php/Insecure_Compiler_Optimization.

[216] *Pull Request:* URL: https://github.com/ethereum/solidity/pull/4797.

[217] *Pull Request: [backport] Fix newline bugs #4937.* URL: https://github.com/ethereum/solidity/pull/4937.

[218] *Pull Request: [libyul] use unique_ptr in AST over shared_ptr #5694.* URL: https://github.com/ethereum/solidity/pull/5694.

[219] *Pull Request: [SMTChecker] Use unique_ptr instead of shared_ptr where applicable #6712.* URL: https://github.com/ethereum/solidity/pull/6712.

[220] *Pull Request: Add override keyword for ensuring function overrides #3737.* URL: https://github.com/ethereum/solidity/pull/3737.

[221] *Pull Request: Add workarounds for building against CVC4 on ArchLinux. #4767.* URL: https://github.com/ethereum/solidity/pull/4767.

[222] *Pull Request: Catch the proper exceptions in solfuzzer #4461.* URL: https://github.com/ethereum/solidity/pull/4461.

[223] *Pull Request: Crash when array index value is too large #4872.* URL: https://github.com/ethereum/solidity/pull/4872.

[224] *Pull Request: Disallow uninitialized storage pointers as experimental #3521.* URL: https://github.com/ethereum/solidity/pull/3521.

[225] *Pull Request: Do not crash on using _slot and _offset suffixes on their own #4724.* URL: https://github.com/ethereum/solidity/pull/4724.

[226] *Pull Request: Docs: Adding AFL's alternative configuration with clang. #4360.* URL: https://github.com/ethereum/solidity/pull/4360.

[227] *Pull Request: Don't exclude public state variables when looking for conflicting declarations. #4508.* URL: https://github.com/ethereum/solidity/pull/4508.

[228] *Pull Request: Enforce data location #4738.* URL: https://github.com/ethereum/solidity/pull/4738.

[229] *Pull Request: Fix type identifiers for RationalNumberType (on negative numbers) #4720.* URL: https://github.com/ethereum/solidity/pull/4720.

[230] *Pull Request: Fixes issue where computing storage size for a number would take too long (or even cause a crash) #4765*. URL: https://github.com/ethereum/solidity/pull/4765.

[231] *Pull Request: Hash linker #5145*. URL: https://github.com/ethereum/solidity/pull/5145.

[232] *Pull Request: Properly handle invalid references used together with _slot and _offset #4729*. URL: https://github.com/ethereum/solidity/pull/4729.

[233] *Pull Request: Remove remaining instances of fillRight #4736*. URL: https://github.com/ethereum/solidity/pull/4736.

[234] *Pull Request: Replace boost:lexical_cast¡std::string¿for std::to_string. #4753*. URL: https://github.com/ethereum/solidity/pull/4753.

[235] *Pull Request: Return TypeError is fixed point encoding is attempted #5807*. URL: https://github.com/ethereum/solidity/pull/5807.

[236] *Pull Request: Some C++ cleanup #5180*. URL: https://github.com/ethereum/solidity/pull/5180.

[237] *Pull Request: Update isolate_tests.py #4434*. URL: https://github.com/ethereum/solidity/pull/4434.

[238] *Pull Request: Warn about unreachable code #5765*. URL: https://github.com/ethereum/solidity/pull/5765.

[239] *Pull Request:CMake policies #4560*. URL: https://github.com/ethereum/solidity/pull/4560.

[240] *Radamsa*. URL: https://github.com/aoh/radamsa.

[241] *Radare2*. URL: https://rada.re/.

[242] Mozilla Research. *Guide to Rustc development*. URL: https://rust-lang.github.io/rustc-guide/.

[243] Mozilla Research. *Rust language*. URL: https://research.mozilla.org/rust/.

[244] *Runnin the fuzzer via AFL - ReadTheDocs Solidity*. URL: https://solidity.readthedocs.io/en/v0.4.24/contributing.html#running-the-fuzzer-via-afl.

[245] *Running the fuzzer via AFL - ReadTheDocs Solidity*. URL: https://solidity.readthedocs.io/en/v0.4.24/contributing.html#running-the-fuzzer-via-afl.

[246] *Security Development Lifecycle (SDL) Banned Function Calls, Microsoft*. URL: https://docs.microsoft.com/en-us/previous-versions/bb288454(v=msdn.10)?redirectedfrom=MSDN.

[247] *Security Enhanced Compilers – Embecosm*. URL: https://www.embecosm.com/2017/02/20/security-enhanced-compilers/.

[248] *Serpent Compiler Audit by Zeppelin*. URL: https://blog.zeppelin.solutions/serpent-compiler-audit-3095d1257929.

[249] *SOL-006 Duplicated super-constructor calls not reported #3325*. URL: https://github.com/ethereum/solidity/issues/3325.

[250] *Solidity Coding Style. GitHub*. URL: https://github.com/ethereum/solidity/blob/v0.5.13/CODING_STYLE.md.

[251] *Solidity Compiler Audit by Zeppelin*. URL: https://blog.openzeppelin.com/solidity-compiler-audit-8cfc0316a420/.

[252] *Solidity Compiler Audit Report by Coinspect*. URL: https://medium.com/@AugurProject/solidity-compiler-audit-report-1832cedb50a8.

[253] *Solidity fuzzing stats*. URL: https://solfuzz.ethdevops.io.

[254] *Solidity Read the Docs*. URL: https://solidity.readthedocs.io/.

[255] *Solidity, tag v0.4.24*. URL: https://github.com/ethereum/solidity/releases/tag/v0.4.24.

[256] *Solidity, the Contract-Oriented Programming Language*. URL: https://github.com/ethereum/solidity/.

[257] *SourceInsight*. URL: https://www.sourceinsight.com/.

[258] *SourceTrail*. URL: https://www.sourcetrail.com/.

[259] *Summary of findings, Solidity Report by Coinspect*. URL: https://github.com/AugurProject/augur-audits/blob/master/solidity-compiler/Coinspect%20-%20Solidity%20Compiler%20Audit%20v1.0.pdf.

[260] *Understand*. URL: https://scitools.com/.

[261] *Unicode Character 'GREEK QUESTION MARK' (U+037E)*. URL: https://www.fileformat.info/info/unicode/char/037e/index.htm.

[262] *Unicode Character 'ZERO WIDTH SPACE' (U+200B)*. URL: https://www.fileformat.info/info/unicode/char/200b/index.htm.

[263] *Unicode line breaking algorithm*. URL: https://www.unicode.org/reports/tr14/tr14-32.html.

[264] *Vyper Read the docs*. URL: https://vyper.readthedocs.io/.

[265] *What is the memory keyword? What does it do? - Solidity 0.4.24 documentation*. URL: https://solidity.readthedocs.io/en/v0.4.24/frequently-asked-questions.html#what-is-the-memory-keyword-what-does-it-do.

[266] *Windbg*. URL: http://www.windbg.org/.

[267] H.A. Yousef. *Kotlin Programming Language: JVM, Andrid, Browser and Server*. Oryx. URL: https://books.google.com.ar/books?id=yNtRDwAAQBAJ.

[268] *Yul Optimiser, Ethereum Solidity GitHub*. URL: https://github.com/ethereum/solidity/blob/v0.5.13/libyul/optimiser/README.md.