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

Contents

Chapter 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.*[51]

> *"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[43][75][42][39]. 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*[51]. 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[65][38].

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[69][66].

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*[76].

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[58].
- 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 [63].
- 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[54].

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.*

Chapter 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[20][45][11][28][31][24][21][44], y se han aplicado estándares como **Building Security In Maturity Model**[49] (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**[79] (ASVS de OWASP) y su versión abierta del **Software Assurance Maturity Model** (Modelo de Madurez de la Seguridad del Software) llamado **OpenSAMM**[80].

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*[17] (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[8] (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[13][18][15][59]:

- **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[37], 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

Figure 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[47].

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 [60].

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.

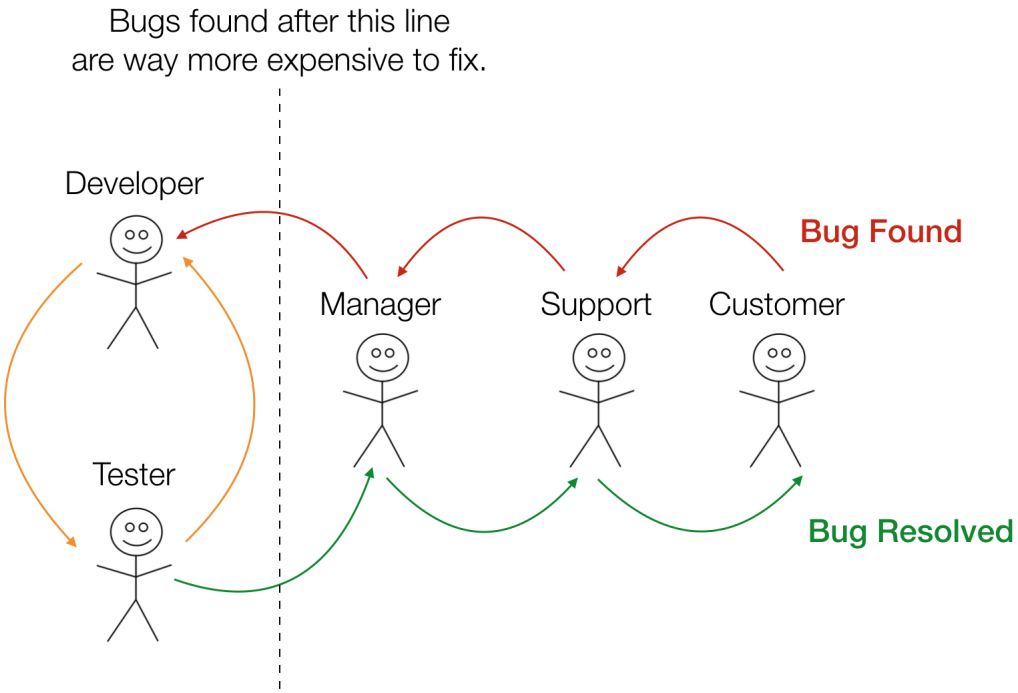

Figure 2.2: Cost of a software bug, Emanuil Slavov.

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[25] el costo para corregir un bug en las distintas etapas se ve descrito por el gráfico de la Figura 2.3

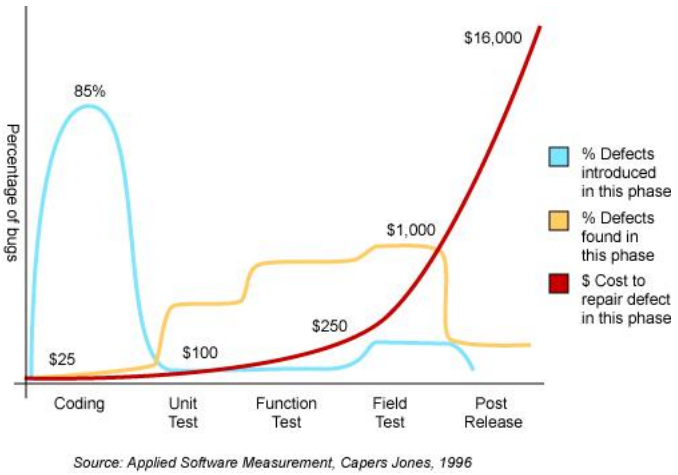

Figure 2.3: Applied Software Measurement, Caper Jones, 1996.

Steve McConnell autor de muchos libros de texto de ingeniería de software conocidos, como **Code Complete**[12], **Rapid Development**[5], y **Software Estimation**[19], 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[14] (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

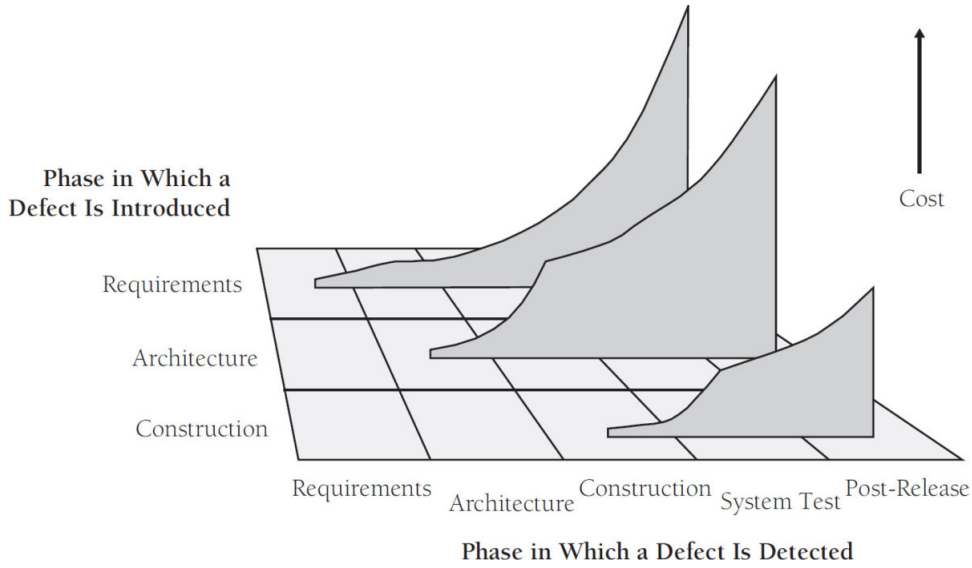

Figure 2.4: Costo de encontrar defectos por fase, Steve McConnell en Code Complete, 2004.

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**[16] (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[71]. El 46% de los incidentes de ciberseguridad del 2017 son debidos a personas internas a la organización[70]. El 30% de los profesionales de seguridad esperan un gran y efectivo ataque dentro de los próximos 90 días.[74]

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[50] 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[50].

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

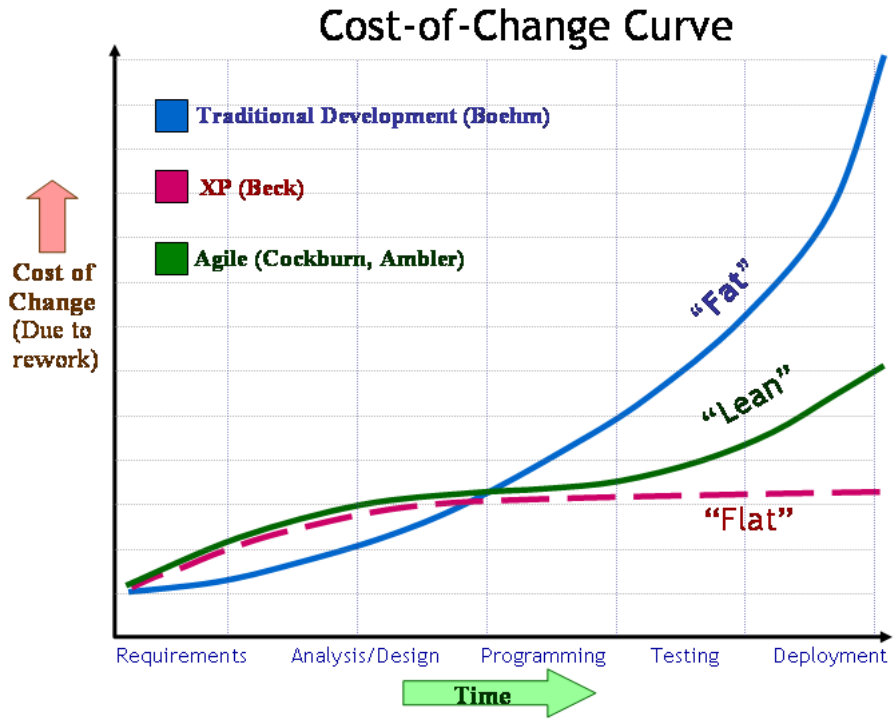

Figure 2.5: Curvas del costo de un cambio, CM Crossroads.

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,

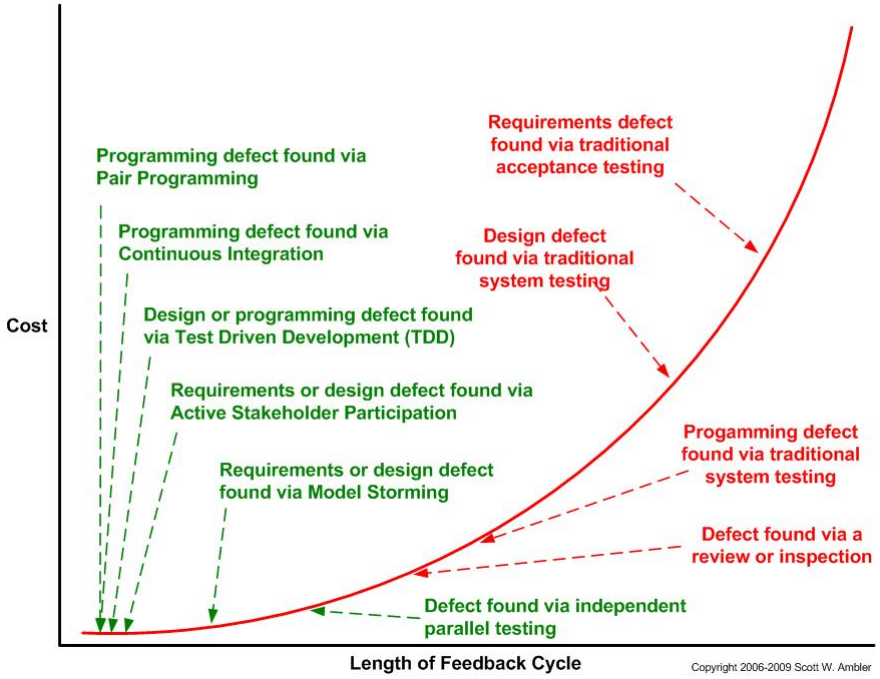

Figure 2.6: Longitud y costo de un defecto dentro del ciclo de desarrollo, Scott W. Ambler

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

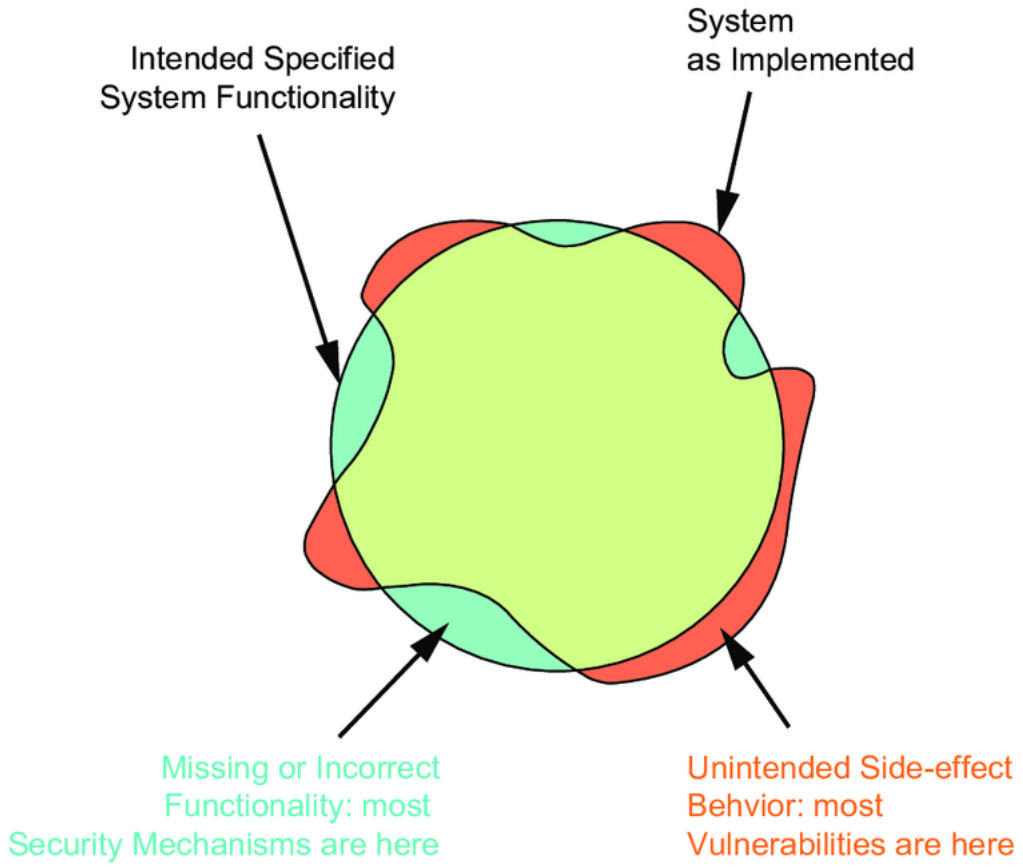

Figure 2.7: Security Testing: A Survey, Michael Felderer.

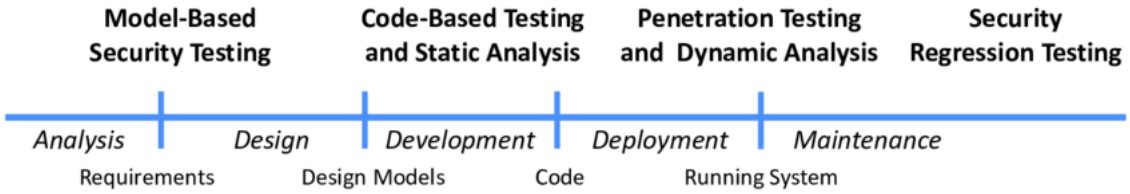

Figure 2.8: Técnicas de prueba de seguridad en el ciclo de vida de desarrollo de software seguro.

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[92][123][125][124], analizadores de código estático[91][94][95][93][111][87], analizadores del software en ejecución/debuggers [120][105][113][127][117], navegadores de binarios/disassemblers[89][90][110][88][106], fuzzers[85][119][107][108][86], 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][9][4][3][22][10][7][6][35][2][1][29][27][23][32][36][40][9][33][61] y con el surgimiento de nuevos lenguajes de programación[30][48][68][78][81][46][41][67][72][64][128] contínuamente está siendo puesta

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

a prueba sin que esto haya significado modificaciones esenciales en el esquema. El caso del lenguaje Rust, de la fundación Mozilla[122] cuyo compilador se encuentra documentado[121] 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.

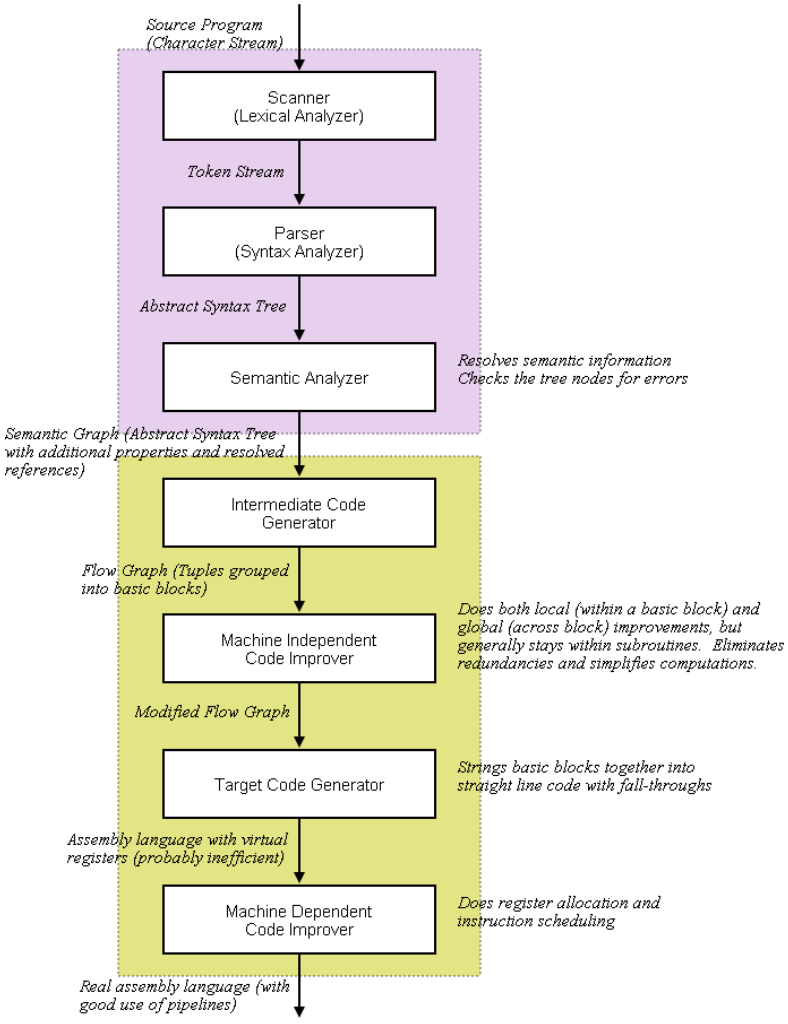

Figure 2.10: Componentes de un compilador. Ray Toal, Loyola Marymount University.

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.

```
#define ZERO 0
unsigned  gcd (
    unsigned int x,// Algoritmo Euclideo
        unsigned y) {
            while ( /* hello */ x> ZERO) {
                unsigned temp=x;
                x=y%x;y  = temp;
            }
        return y;
}
```

Listing 2.1: Código de ejemplo

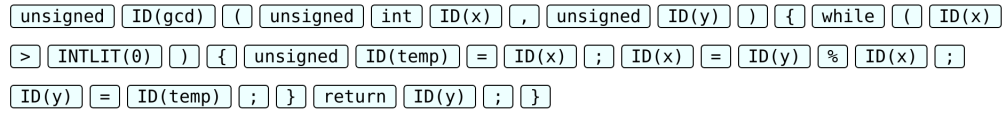

Figure 2.11: Conjunto de tokens. Ray Toal, Loyola Marymount University.

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 **??**

```
(fundecl unsigned gcd
  (params (param unsigned x) (param unsigned y))
  (block
    (while
      (> x 0)
      (block (vardecl unsigned temp y) (= x (% y x)) (= y temp)))
    (return y)))
```

Listing 2.2: AST

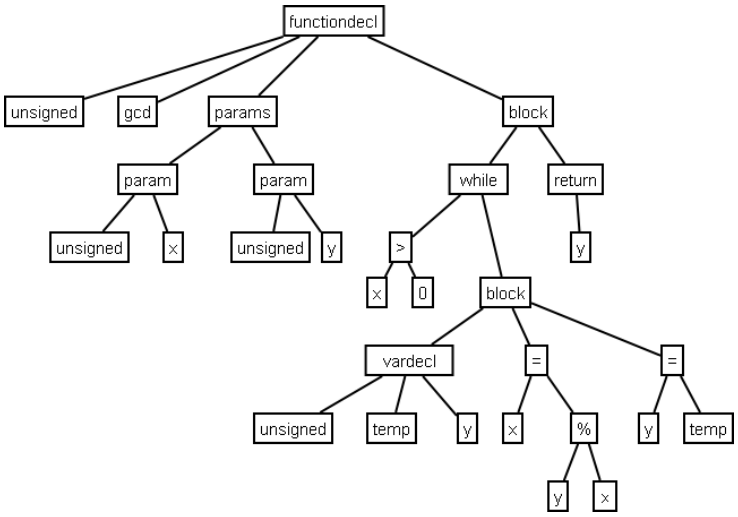

Figure 2.12: Árbol sintáctico abstracto. Ray Toal, Loyola Marymount University.

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;`

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.

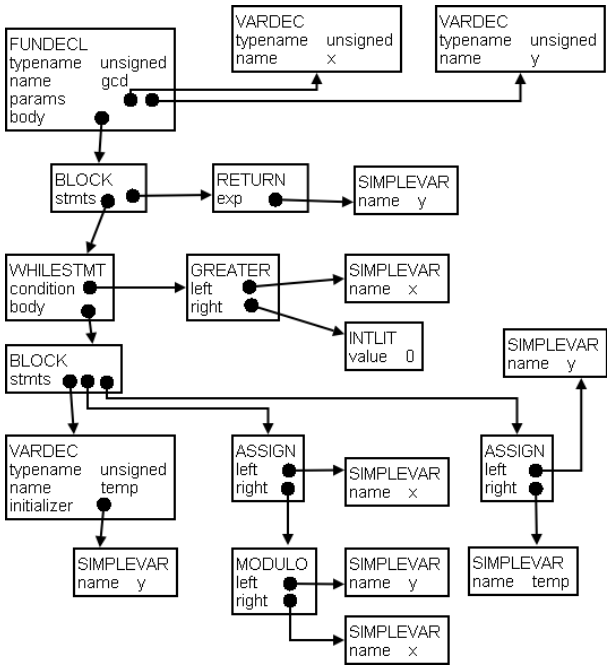

Figure 2.13: Vista de objetos del AST. Ray Toal, Loyola Marymount University.

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[73][62] (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.

```
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
```

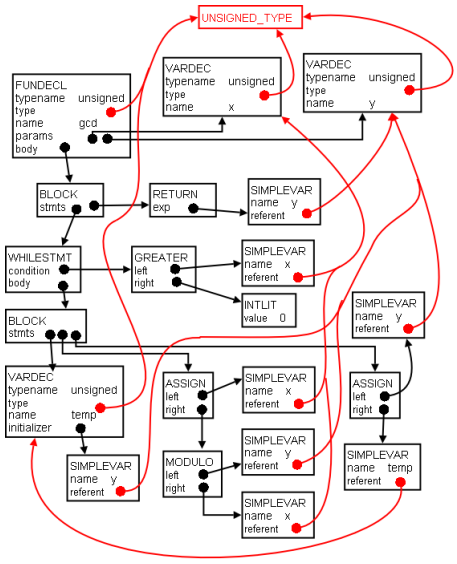

Figure 2.14: AST con análisis semántico. Ray Toal, Loyola Marymount University.

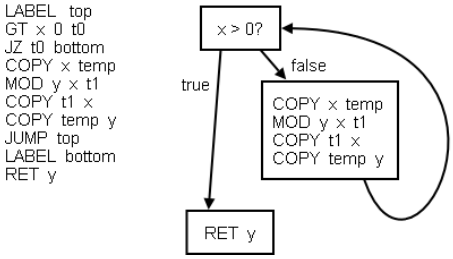

Figure 2.15: Gráfico de flujo intermedio. Ray Toal, Loyola Marymount University.

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
        divl    %edi
        movl    %edi, %eax
        testl   %edx, %edx
        jne     .L5
.L1:
        movl    %edi, %eax
        Ret
```

```
.L7:
        movl    %esi, %edi
        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[56], aunque la interpretación puede verse como una suerte de compilación en línea, compilación bajo demanda o compilación JIT (*Just in Time*)[34][32].

¿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.[118]

```
void GetData(char *MFAddr) {
    char pwd[64];
    if (GetPasswordFromUser(pwd, sizeof(pwd)))
     if (ConnectToMainframe(MFAddr, pwd)) {
            // Interaction with mainframe
        }
     }
     memset(pwd, 0, sizeof(pwd));
}
```

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.[114]

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:[26]

```
struct sock *sk = tun->sk;  // initialize sk with tun->sk
...
if (!tun)
    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[101], Python[100] y PHP[99] 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.[77]

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*[52].

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.[83]

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.[83]

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[115], 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[82] 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.

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.

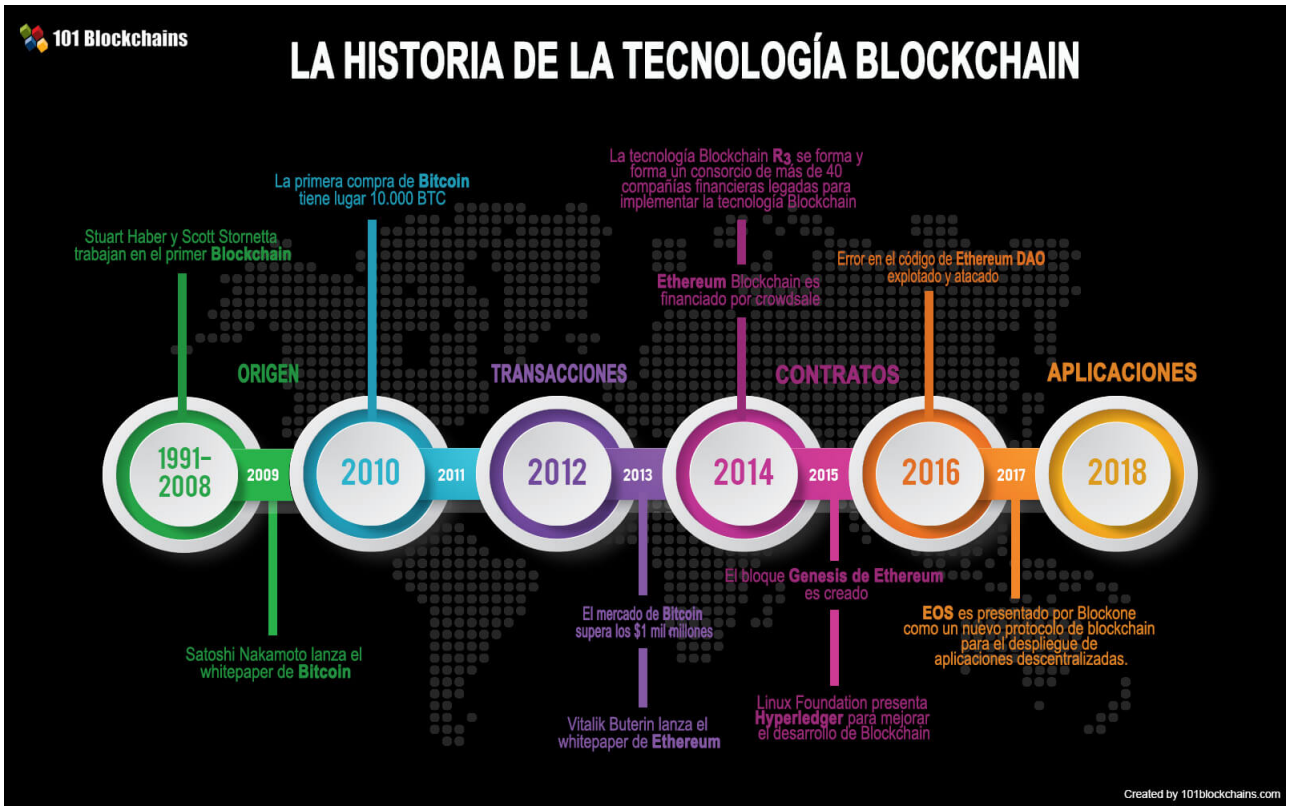

Figure 2.16: Cronología de tecnologías blockchain por 101blockchains.

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**[104] (un lenguaje de bibliotecas que posee similitudes con C y JavaScript), **Serpent**[103] (similar a Python, pero deprecado),

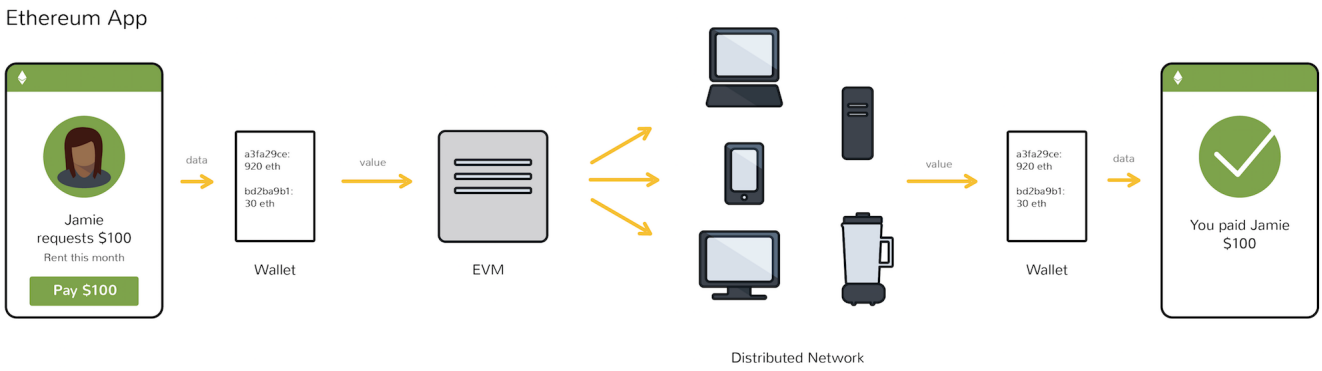

Figure 2.17: Diagrama de despliege, por Coindesk.

LLL[102] (un lenguaje bajo nivel similar a Lisp), **Mutan**[116] (basado en Go, pero también deprecado), **Vyper**[126] (orientado a la investigación, fuertemente tipado, basado en Python, recursivo), y recientemente la empresa **BlockStack**[98] presentó **Clarity**[112] (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**[109].

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.

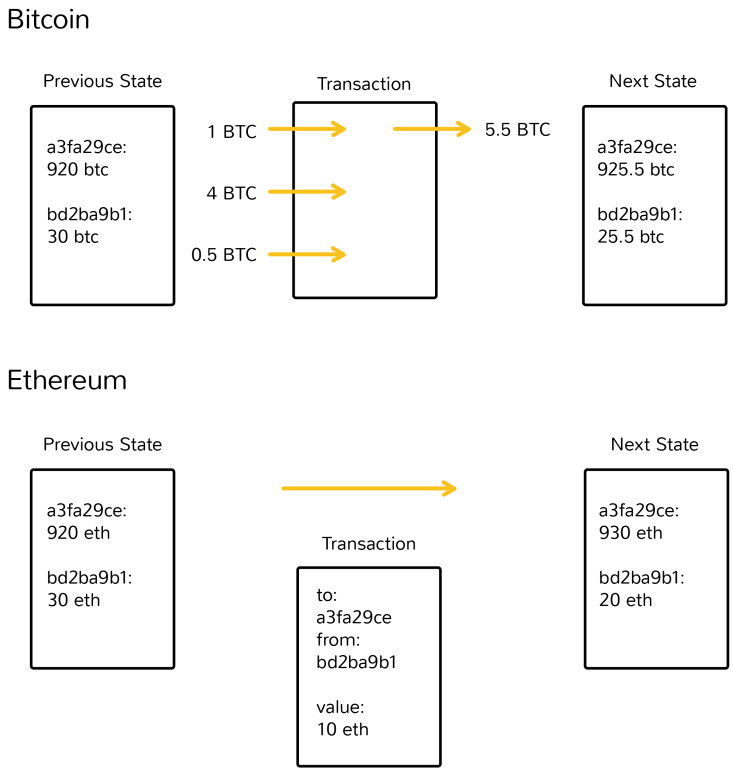

Figure 2.18: Diferencia del estado de las transacciones por Coindesk.

¿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[96] 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[97].

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[84] 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.[53]

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[57].

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[55].

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.*

Bibliography

[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 and 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 et al. *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] K. Slonneger and 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.

[5] 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.

[6] 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.

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

[8] Jim Highsmith and Alistair Cockburn. "Agile Software Development: The Business of Innovation". In: *IEEE Computer* 34 (2001), pp. 120–122.

[9] Y.N. Srikant and 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.

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

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

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

[13] Gary McGraw. "Software Security". In: *Security & Privacy, IEEE* 2 (Apr. 2004), pp. 80–83. DOI: 10.1109/MSECP.2004.1281254.

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

[15] B. Arkin, S. Stender, and G. McGraw. "Software penetration testing". In: *Security & Privacy, IEEE* 3 (Feb. 2005), pp. 84–87. DOI: 10.1109/MSP.2005.23.

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

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

[18] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Vol. 34. June 2006. ISBN: 0735622140. DOI: 10.1007/s11623-010-0021-7.

[19] 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.

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

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

[22] 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.

[23] T. Reps, M. Sagiv, and 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.

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

[25] 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.

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

[27] 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.

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

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

[30] B. Dasnois. *HaXe 2 Beginner's Guide*. Community experience distilled. Packt Publishing, 2011. ISBN: 9781849512572. URL: https://books.google.com.ar/books?id=sXlOUrMJZ4C.

[31] 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.

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

[33] Y. Su and 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.

[34] 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.

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

[36] 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.

[37] E. Ries and 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.

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

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

[40] H. Seidl, R. Wilhelm, and 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.

[41] 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.

[42] 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.

[43] 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.

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

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

[46] A.A.A. Donovan and 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.

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

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

[49] Gary McGraw. "Software Security and the Building Security in Maturity Model (BSIMM)". In: *J. Comput. Sci. Coll.* 30.3 (Jan. 2015), pp. 7–8. ISSN: 1937-4771. URL: http://dl.acm.org/citation.cfm?id=2675327.2675329.

[50] Michael Felderer et al. "Chapter One - Security Testing: A Survey". In: ed. by Atif Memon. Vol. 101. Advances in Computers. Elsevier, 2016, pp. 1–51. DOI: https://doi.org/10.1016/bs.adcom.2015.11.003. URL: http://www.sciencedirect.com/science/article/pii/S0065245815000649.

[51] Michael Felderer et al. "Security Testing: A Survey". In: Mar. 2016, pp. 1–51. DOI: 10.1016/bs.adcom.2015.11.003.

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

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

[54] 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/.

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

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

[57] 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/.

[58] 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/.

[59] Jim Manicode. *Secure Software Development Life Cycle*. 2017. URL: https://www.youtube.com/watch?v=M7qMP3C5bkU/.

[60] Alex McPeak. *What's the True Cost of a Software Bug?* 2017. URL: crossbrowsertesting.com/blog/development/software-bug-cost.

[61] 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.

[62] Emmanuel Okon. "The New Trends in Compiler Analysis and Optimizations". In: *International Journal of computer trends and technology* 46 (May 2017).

[63] 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.

[64] 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.

[65] Nicole Forsgren. "Accelerate: State of DevOps, Strategies for a New Economy". In: 2018, pp. 58–60.

[66] Nicole Forsgren, Jez Humble, and 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.

[67] 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.

[68] S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, 2018. ISBN: 9781593278519. URL: https://books.google.com.ar/books?id=lrgrDwAAQBAJ.

[69] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. San Francisco, CA, USA: No Starch Press, 2018. ISBN: 1593278284, 9781593278281.

[70] Kaspersky Lab. *The human factor in IT security*. 2018. URL: https://www.kaspersky.com/blog/the-human-factor-in-it-security/.

[71] McAfee and 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.

[72] J. Skeen and 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.

[73] Y.N. Srikant and 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.

[74] 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.

[75] 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.

[76] Ethereum Foundation. *Solidity: Read the docs*. 2019. URL: https://solidity.readthedocs.io/.

[77] Ledger. "How It All Began: A Brief History On Bitcoin & Cryptocurrencies". In: (2019). URL: https://www.ledger.com/how-it-all-began-a-brief-history-of-bitcoin-cryptocurrencies/.

[78] 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.

[79] OWASP. *Application Security Verification StandardSoftware Assurance Maturity Model*. 2019. URL: https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project.

[80] OWASP. *Software Assurance Maturity Model*. 2019. URL: https://www.opensamm.org/.

[81] R. Sharma and 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.

[82] 101blockchains. *¿Qué es Ethereum?* URL: https://101blockchains.com/es/hyperledger-vs-corda-r3-vs-ethereum-la-guia/.

[83] 101blockchains. *Historia de la blockchain*. URL: https://101blockchains.com/es/historia-de-la-blockchain/.

[84] *A timeline of major crypto-exchange hacks*. URL: https://discover.ledger.com/hackstimeline/.

[85] *AFL*. URL: http://lcamtuf.coredump.cx/afl/.

[86] *Awesome Fuzzing*. URL: https://github.com/secfigo/Awesome-Fuzzing.

[87] *Awesome Static Analysis*. URL: https://github.com/mre/awesome-static-analysis#multiple-languages-1.

[88] *Binary Ninja*. URL: https://binary.ninja/.

[89] *Binnavi*. URL: https://github.com/google/binnavi.

[90] *Capstone*. URL: https://github.com/aquynh/capstone.

[91] *Clang*. URL: http://clang.llvm.org/.

[92] *CodeCompass*. URL: https://github.com/Ericsson/CodeCompass.

[93] *CodeSonnar*. URL: https://www.grammatech.com/products/codesonar.

[94] *Coverity*. URL: http://www.coverity.com/.

[95] *cppCheck*. URL: http://cppcheck.sourceforge.net/.

[96] *Cryptocurrency Market Capitalizations*. URL: https://coinmarketcap.com/.

[97] *DappRadar - Ranked list of blockchain dapps*. URL: https://dappradar.com/rankings/.

[98] *Decentralized computing network and app ecosystem*. URL: https://blockstack.org/.

[99] CVE details. *PHP Security Vulnerabilities*. URL: https://www.cvedetails.com/product/128/PHP-PHP.html?vendor_id=74.

[100] CVE details. *Python Security Vulnerabilities*. URL: https://www.cvedetails.com/product/18230/Python-Python.html?vendor_id=10210.

[101] CVE details. *Ruby lang Security Vulnerabilities*. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-7252/product_id-12215/Ruby-lang-Ruby.html.

[102] Ethereum Foundation. *LLL Compiler Documentation*. URL: https://lll-docs.readthedocs.io/en/latest/lll_introduction.html.

[103] Ethereum Foundation. *Serpent's GitHub*. URL: https://github.com/ethereum/serpent.

[104] Ethereum Foundation. *Solidity Read the docs*. URL: https://solidity.readthedocs.io.

[105] *gdb*. URL: https://www.gnu.org/s/gdb/.

[106] *Ghidra*. URL: https://ghidra-sre.org/.

[107] *Grammarinator*. URL: https://github.com/renatahodovan/grammarinator.

[108] *Honggfuzz*. URL: https://github.com/google/honggfuzz.

[109] *How Ethereum works*. URL: https://www.ethereum.org/learn/#how-ethereum-works.

[110] *Ida Pro*. URL: https://www.hex-rays.com/products/ida/.

[111] *Infer*. URL: https://github.com/facebook/infer.

[112] *Introducing Clarity, a language for predictable smart contracts*. URL: https://blog.blockstack.org/introducing-clarity-the-language-for-predictable-smart-contracts/.

[113] *lldb*. URL: https://lldb.llvm.org/.

[114] Microsoft. *Beware of compiler optimizations*. URL: https://wiki.sei.cmu.edu/confluence/display/c/MSC06-C.+Beware+of+compiler+optimizations.

[115] MiEthereum. *Vitalik Buterin*. URL: https://miethereum.com/vitalik-buterin/.

[116] *Mutan's GitHub*. URL: https://github.com/obscuren/mutan.

[117] *Ollydbg*. URL: http://www.ollydbg.de/.

[118] OWASP. *Insecure Compiler Optimization*. URL: https://www.owasp.org/index.php/Insecure_Compiler_Optimization.

[119] *Radamsa*. URL: https://github.com/aoh/radamsa.

[120] *Radare2*. URL: https://rada.re/.

[121] Mozilla Research. *Guide to Rustc development*. URL: https://rust-lang.github.io/rustc-guide/.

[122] Mozilla Research. *Rust language*. URL: https://research.mozilla.org/rust/.

[123] *SourceInsight*. URL: https://www.sourceinsight.com/.

[124] *SourceTrail*. URL: https://www.sourcetrail.com/.

[125] *Understand*. URL: https://scitools.com/.

[126] *Vyper Read the docs*. URL: https://vyper.readthedocs.io/.

[127] *Windbg*. URL: http://www.windbg.org/.

[128] H.A. Yousef. *Kotlin Programming Language: JVM, Andrid, Browser and Server*. Oryx. URL: https://books.google.com.ar/books?id=yNtRDwAAQBAJ.