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

Estado del arte

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

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

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

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

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

1.1 CSMith (2009-2013)

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

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

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

Listing 1.1: Código que produjo bug en GCC

El código del recuadro1.1 pertenece a un bug hallado en una versión de GCC incluído con Ubuntu Linux 8.04.1 para x86. En todos los niveles de optimización compilaba esa función de forma que en su ejecución retorne el valor "1"; cuando el resultado correcto es "0". Lo que sucedió

fue que el compilador de Ubuntu había sido fuertemente modificado a través de lo que se conoce como "parche", ya que la versión base de GCC no poseía ese bug[9].

Csmith genera un programa en C; luego un código de prueba (conocido como "arnés de prueba") compila ese programa utilizando diversos compiladores, ejecuta los ejecutables, y compara las salidas. A pesar de que esta estrategia ya había sido utilizada previamente [8][3][6], las técnicas de generación de Csmith, para ese momento, avanzaron sustancialmente el estado del arte generando programas aleatorios que son expresivos—conteniendo código complejo utilizando muchas de sus características—mientras también aseguraban que cada programa generado tenga una sola interpretación. Para poder hacer esto, un programa no debe poder ejecutar ningún tipo de comportamientos indefinido, ni debe depender de ninguno de los 52 tipos de comportamiento no especificado que están descritos en el estándar C99[32].

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

Este proyecto comenzó como un **fork** (bifurcación) de **Randprog**[7], un proyecto ya existente que genera programas aleatorios en C de 1600 líneas de código. En sus trabajos tempranos extendieron y adaptaron Randprog para encontrar bugs en la parte de la traducción de accesos a objetos calificados de volátiles, lo que dió como resultado a un programa de 7000 líneas de código. Los autores convirtieron Randprog en Csmith, un programa C++ de 40,000 líneas para generar programas de C aleatorios. En comparación con Randprog, Csmith puede generar programas de C que utilizan una gama mucho más amplia de características de C, incluido el flujo de control complejo y estructuras de datos como punteros, matrices y estructuras.

El programa Csmith utiliza pruebas diferenciales aleatorias. Las pruebas aleatorias[5], también llamadas fuzzing[1], consisten en un método de test black box en el que las entradas de prueba se generan aleatoriamente. Las pruebas diferenciales aleatorias[3] tienen la ventaja de que no se necesita un oráculo (principio heurístico o mecanismo por el cual podremos reconocer un problema, y determinar si es correcto o no). para los resultados de las pruebas. Explota la idea de que si uno tiene implementaciones deterministas múltiples de la misma especificación, todas las implementaciones deben producir el mismo resultado de la misma entrada válida. Cuando dos implementaciones producen salidas diferentes, una de ellas debe ser defectuosa. Dadas tres o más implementaciones, un evaluador puede usar el voto para determinar heurísticamente qué implementaciones son incorrectas. La Figura 1.1 muestra cómo usaron estas ideas para encontrar errores de compilación.

1.1.1 Resultados obtenidos

Desde el año 2009 al 2013 encontraron 476 bugs en GCC[31] y LLVM[33].

La lista de bugs es accesible desde su repositorio[30] en github, también desde el sitio del departamento de Ciencias de la computación de UTAH.

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

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

1.2 DeepSmith (2018)

Los autores de la herramienta presentan **DeepSmith**[26], como un novedoso enfoque de aprendizaje automático para acelerar la validación del compilador a través de la inferencia de modelos generativos para las entradas del compilador. Su enfoque infiere un modelo, aprendido de la estructura de código real basado en un gran códigos open source. Luego, utiliza el modelo para generar

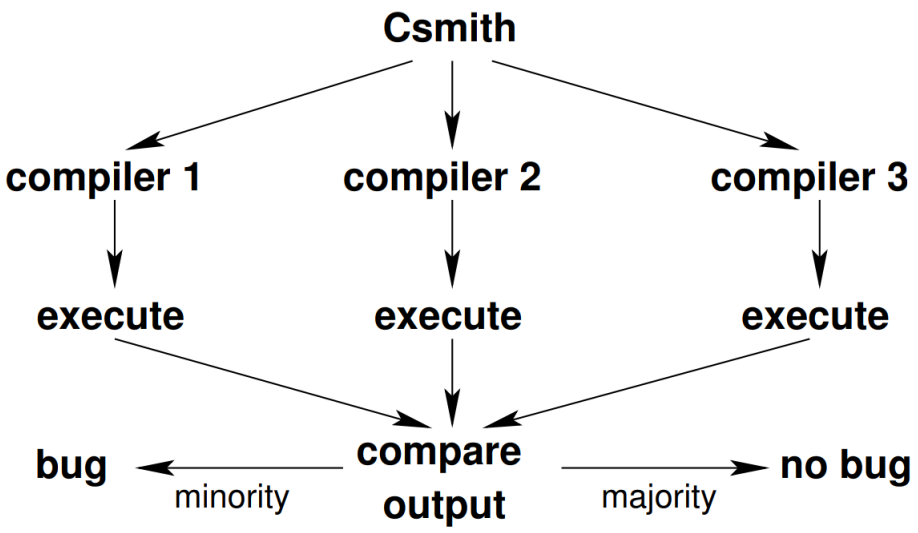

Figure 1.1: Ejecución en distintos compiladores

automáticamente decenas de miles de programas realistas. Finalmente, aplican metodologías de pruebas diferenciales establecidas para exponer errores en los compiladores. Se ha aplicado este enfoque al lenguaje de programación **OpenCL**, exponiendo automáticamente los errores en los compiladores de OpenCL con poco esfuerzo de su lado. En 1.000 horas de pruebas automatizadas de compiladores comerciales y de código abierto, descubrieron errores en todos ellos.

1.2.1 Comparación con CSmith

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

Por ejemplo, adaptar CSmith de C a **OpenCL**[2] - una tarea que parecería ser simple - les tomó 9 meses y 8k líneas adicionales de código. Dada la dificultad de definir una nueva gramática, generalmente solo se implementa un subconjunto del lenguaje.

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

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

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

1.2.2 Extensibilidad del modelado de lenguajes

Una gran parte de la arquitectura de DeepSmith es independiente del lenguaje;ya que solo requiere un corpus, un codificador y un arnés para cada nuevo lenguaje. Esto potencialmente reduce significativamente la barrera de entrada en comparación con los fuzzers basados en gramática

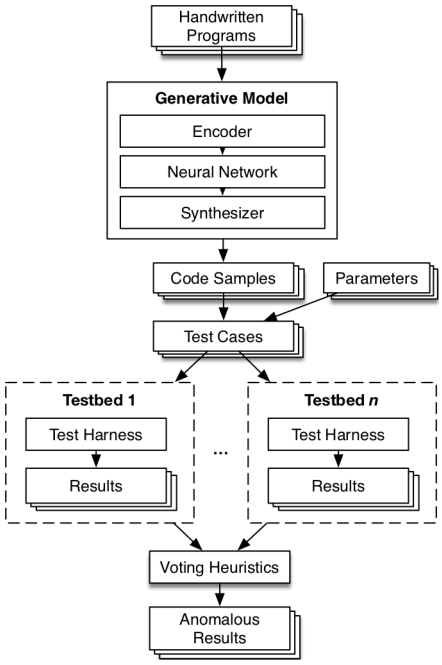

Figure 1.2: Arquitectura de DeepSmith

anteriores (la gran mayoría). Para explorar esto, deciden intentarlo con un lenguaje reciente, siendo este Solidity.

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

1.2.3 Resultados iniciales

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

Sus resultados se resumen en la tabla que muestra la Figura 1.3.

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

Figure 1.3: Resultados de testear solc y solc-js.

Los resultados de la investigación demostraron numerosos casos en los que el compilador se bloquea silenciosamente y dos aserciones distintas del compilador. El primero se debe a la falta de manejo de errores en las características del idioma (este problema es conocido por los desarrolladores). La fuente de la segunda afirmación es el tiempo de ejecución de JavaScript y se activa solo en la versión Emscripten, lo que sugiere un error en la traducción automática de LLVM a JavaScript.

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

indican la utilidad de su enfoque para simplificar la generación de casos de prueba.

1.2.4 Trabajos relacionados

La generación aleatoria de casos de prueba es un enfoque bien establecido para el problema de validación del compilador. Los enfoques anteriores se examinan en *Compiler test case generation methods: a survey and assessment*[2], *Survey of Compiler Testing Methods*[4] y se contrastan empíricamente en *An empirical comparison of compiler testing techniques*[14]. La principal pregunta de interés es cómo generar de manera eficiente los códigos que desencadenan errores.

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

1.2.5 Programa de generación

En el trabajo fundacional sobre pruebas diferenciales para compiladores, McKeeman et al. presentan generadores actuales capaces de enumerar programas de una variedad de calidades, desde secuencias ASCII aleatorias hasta programas conformes con el modelo C[3]. Los trabajos posteriores han presentado generadores cada vez más complejos que mejoran en alguna métrica de interés, generalmente expresividad o probabilidad de corrección. CSmith[9] es un generador ampliamente conocido y efectivo que enumera programas al vincular funciones de lenguaje combinadas con poca frecuencia. Al hacerlo, produce programas correctos con un comportamiento claramente definido pero una funcionalidad muy poco probable, lo que aumenta las posibilidades de desencadenar un error.

Lograr esto requirió un extenso trabajo de ingeniería, la mayoría no se puede transportar a otros idiomas, e ignorar algunas características del idioma. Los generadores subsiguientes influenciados por CSmith, como Orange3[12], se enfocan en características y tipos de errores más allá del alcance de CSmith, errores aritméticos en el caso de Orange3.

Glade[16] deriva una gramática de un corpus de programas de ejemplo. La gramática derivada se enumera para producir nuevos programas, aunque a diferencia de nuestro enfoque, no se aprende ninguna distribución sobre la gramática; la enumeración del programa es uniformemente aleatoria.

1.2.6 Programa de mutación

La prueba de entradas de módulo de equivalencia (EMI)[11][15] sigue un enfoque diferente para la generación de casos de prueba. Comenzando con el código existente, inserta o elimina instrucciones que no se ejecutarán, por lo que la funcionalidad debe seguir siendo la misma. Si se ve afectado, se debe a un error del compilador. Si bien es una técnica poderosa capaz de encontrar errores difíciles de detectar, se basa en tener una gran cantidad de programas para mutar. Como tal, todavía requiere un generador de código externo. De manera similar a CSmith, EMI favorece los programas de prueba muy largos.

LangFuzz[10] también usa la mutación, pero lo hace insertando segmentos de código que previamente han expuesto errores. Esto aumenta las posibilidades de descubrir vulnerabilidades en los motores de lenguaje de scripting.

La enumeración de programas esqueléticos[25] funciona nuevamente al transformar el código existente. Identifica patrones algorítmicos en piezas cortas de código y enumera todas las posibles permutaciones del uso variable. Comparado con todo esto, su enfoque de fuzzing es de bajo costo, fácil de desarrollar, portátil, capaz de detectar una amplia gama de errores y enfocado por diseño a los errores que es más probable encontrar en un escenario de producción.

1.2.7 Aprendizaje automático

Existe un creciente interés en aplicar el aprendizaje automático a las pruebas de software. Más parecido a su trabajo es Learn & fuzz[19], en el cual una red LSTM se entrena a través de un conjunto de archivos PDF para generar entradas de prueba para el renderizador Microsoft Edge, produciendo un error. A diferencia de las pruebas de compilación, los casos de prueba de PDF no requieren entradas ni procesamiento previo del cuerpo de entrenamiento.

Skyfire[24] aprende una gramática probabilística sensible al contexto sobre un corpus de programas para generar semillas de entrada para pruebas de mutación. Se muestra que las semillas

generadas mejoran la cobertura del código de AFL[29] cuando confunden los motores XSLT y XML, aunque las semillas no se usan directamente como casos de prueba. El aprendizaje automático también se ha aplicado a otras áreas, como la mejora de los analizadores estáticos de detección de errores [20][22] la reparación de programas[23][27], la priorización de los programas de prueba[17], la identificación de buffer overruns (sobrecargas de búfer)[18] y el procesamiento de informes de errores[21][13]. Según el conocimiento de los autores, ningún trabajo hasta el momento ha tenido éxito en la búsqueda de errores de compilación al explotar la sintaxis aprendida del código fuente extraído para la generación de casos de prueba. Aparentemente el trabajo de estos autores es el primero en hacerlo.

1.3 Serpent by Zeppelin (Jul 2017)

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

La empresa Augur (un servicio de apuestas descentralizado) los contrató para realizarle una auditoría a Serpent, un compilador de un lenguaje Python-style que compila a EVM. El interés de Augur en esta auditoría está dado por razones propias, tuvieron un inconveniente de seguridad por utilizar este lenguaje. El código del proyecto se puede ver aún en su repositorio de GitHub[28].

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

En su publicación[34], la introducción a los resultados del reporte también es utilizada como conclusión al final del mismo. El contenido de ella es una la lista con los problemas apuntando directamente al documento original que posee detalle técnico.

El reporte original menciona que su estrategia fue analizar el código C++ del compilador, sumado a revisar documentación, ejemplos, y herramientas recomendadas para trabajar con los contratos de Serpent. Realizaron diversos contratos de ejemplo minimales, para verificar y exponer los problemas que encontraron. Analizaron el código assembler tanto en LLL (Lisp Like Language, un lenguaje con un nivel de abstracción superior a la EVM, pero considerado assembler) como en la EVM, así como también su comportamiento en ejecución.

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

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

1.4 Solidity by Coinspect (Nov 2017)

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

Coinspect, es un equipo reconocido por sus trabajos específicamente en el ambiente de la cyber seguridad. Entendiendo esto, no es extraño encontrar en la introducción de la publicación[35] un poco más de detalle[39] sobre lo que esperan encontrar o qué piensan en buscar relacionado en términos de seguridad.

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

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

También buscaron vulnerabilidades comunes a vectores en aplicaciones de software:

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

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

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

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

1.5 Solidity by Ethereum Foundation

Para situarse en contexto, el compilador de Solidity junto con su lenguaje fueron impulsados por la Ethereum Foundation. Su código está disponible en GitHub[38] desde sus comienzos, donde realizan releases periódicamente, tienen su documentación en ReadTheDocs[37], y una amplia comunidad que colabora con el proyecto.

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

Varios de los bugs reportados por el equipo de Coinspect siguen sin estar corregidos, y no es posible distinguir en el sistema de reportes de tickets de github, qué bugs corresponden a su suite de testing, y de qué manera se han encontrado. Tal vez muchos errores no llegan a ser publicados mediante commits porque son testeados localmente, y esa información no se encuentra visible al público. El sitio solfuzz.ethdevops.io[36] solía dar estadísticas de los resultados encontrados por el fuzzer incorporado pero en el momento de este análisis se comprobó que ha dejado de funcionar hace tiempo.

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

1.6 Discusión sobre las soluciones presentadas

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

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

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

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

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

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

Bibliography

[1] Barton P. Miller, Louis Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: http://doi.acm.org/10.1145/96267.96279.

[2] Abdulazeez S. Boujarwah and Kassem Saleh. "Compiler test case generation methods: a survey and assessment". In: *Information & Software Technology* 39 (1997), pp. 617–625.

[3] William M. McKeeman. "Differential Testing for Software". In: *DIGITAL TECHNICAL JOURNAL* 10.1 (1998), pp. 100–107.

[4] Alexander Kossatchev and Mikhail Posypkin. "Survey of Compiler Testing Methods". In: *Programming and Computer Software* 31 (Jan. 2005), pp. 10–19. DOI: 10.1007/s11086-005-0008-6.

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

[6] Flash Sheridan. "Practical testing of a C99 compiler using output comparison". In: *Softw., Pract. Exper.* 37 (Nov. 2007), pp. 1475–1488. DOI: 10.1002/spe.812.

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

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

[9] Xuejun Yang et al. "Finding and Understanding Bugs in C Compilers". In: *SIGPLAN Not.* 46.6 (June 2011), pp. 283–294. ISSN: 0362-1340. DOI: 10.1145/1993316.1993532. URL: http://doi.acm.org/10.1145/1993316.1993532.

[10] Christian Holler, Kim Herzig, and Andreas Zeller. "Fuzzing with Code Fragments". In: Aug. 2012.

[11] Vu Le, Mehrdad Afshari, and Zhendong Su. "Compiler Validation via Equivalence Modulo Inputs". In: *ACM SIGPLAN Notices* 49 (June 2014). DOI: 10.1145/2594291.2594334.

[12] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. "Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions". In: *IPSJ Transactions on System LSI Design Methodology* 7 (Jan. 2014), pp. 91–100. DOI: 10.2197/ipsjtsldm.7.91.

[13] An Lam et al. "Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N)". In: Nov. 2015, pp. 476–481. DOI: 10.1109/ASE.2015.73.

[14] Junjie Chen et al. "An empirical comparison of compiler testing techniques". In: May 2016, pp. 180–190. DOI: 10.1145/2884781.2884878.

[15] Chengnian Sun, Vu Le, and Zhendong Su. "Finding compiler bugs via live code mutation". In: Oct. 2016, pp. 849–863. DOI: 10.1145/2983990.2984038.

[16] Osbert Bastani et al. "Synthesizing Program Input Grammars". In: *ACM SIGPLAN Notices* 52 (June 2017), pp. 95–110. DOI: 10.1145/3140587.3062349.

[17] Junjie Chen et al. "Learning to Prioritize Test Programs for Compiler Testing". In: May 2017. DOI: 10.1109/ICSE.2017.70.

[18] Min-je Choi et al. "End-to-End Prediction of Buffer Overruns from Raw Source Code via Neural Memory Networks". In: Aug. 2017, pp. 1546–1553. DOI: 10.24963/ijcai.2017/214.

[19] Patrice Godefroid, Hila Peleg, and Rishabh Singh. "Learn and Fuzz: Machine learning for input fuzzing". In: Oct. 2017, pp. 50–59. DOI: 10.1109/ASE.2017.8115618.

[20] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. "Machine-Learning-Guided Selectively Unsound Static Analysis". In: May 2017, pp. 519–529. DOI: 10.1109/ICSE.2017.54.

[21] Xuan Huo and Ming Li. "Enhancing the Unified Features to Locate Buggy Files by Exploiting the Sequential Nature of Source Code". In: Aug. 2017, pp. 1909–1915. DOI: 10.24963/ijcai.2017/265.

[22] Ugur Koc et al. "Learning a classifier for false positive error reports emitted by static code analysis tools". In: June 2017, pp. 35–42. DOI: 10.1145/3088525.3088675.

[23] Manos Koukoutos et al. "On Repair with Probabilistic Attribute Grammars". In: (July 2017).

[24] Junjie Wang et al. "Skyfire: Data-Driven Seed Generation for Fuzzing". In: May 2017, pp. 579–594. DOI: 10.1109/SP.2017.23.

[25] Qirun Zhang, Chengnian Sun, and Zhendong Su. "Skeletal program enumeration for rigorous compiler testing". In: June 2017, pp. 347–361. DOI: 10.1145/3062341.3062379.

[26] Chris Cummins et al. "Compiler Fuzzing Through Deep Learning". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: ACM, 2018, pp. 95–105. ISBN: 978-1-4503-5699-2. DOI: 10.1145/3213846.3213848. URL: http://doi.acm.org/10.1145/3213846.3213848.

[27] Martin White et al. "Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities". In: Feb. 2019, pp. 479–490. DOI: 10.1109/SANER.2019.8668043.

[28] URL: https://github.com/ethereum/serpent/tree/ad53fa2a8a496448d58ef9137959b4a1e86b14d7.

[29] *American Fuzzy Lop*. URL: http://lcamtuf.coredump.cx/afl/.

[30] *Csmith's GitHub - Bugs reported*. URL: https://github.com/csmith-project/csmith/blob/master/BUGS_REPORTED.TXT.

[31] *GCC Bug List Found by Random Testing*. URL: http://embed.cs.utah.edu/csmith/gcc-bugs.html.

[32] ISO & IEC. "Open STD c99". In: (). URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[33] *LLVM Bug List Found by Random Testing*. URL: http://embed.cs.utah.edu/csmith/llvm-bugs.html.

[34] *Serpent Compiler Audit by Zeppelin*. URL: https://blog.zeppelin.solutions/serpent-compiler-audit-3095d1257929.

[35] *Solidity Compiler Audit Report by Coinspect*. URL: https://medium.com/@AugurProject/solidity-compiler-audit-report-1832cedb50a8.

[36] *Solidity fuzzing stats*. URL: https://solfuzz.ethdevops.io.

[37] *Solidity Read the Docs*. URL: https://solidity.readthedocs.io/.

[38] *Solidity, the Contract-Oriented Programming Language*. URL: https://github.com/ethereum/solidity/.

[39] *Summary of findings, Solidity Report by Coinspect*. URL: https://github.com/AugurProject/augur-audits/blob/master/solidity-compiler/Coinspect%20-%20Solidity%20Compiler%20Audit%20v1.0.pdf.