Undefined Behavior: El uso de Herramientas Estáticas para su detección (Mayo 2020)

María Inés Quihillalt, Estudiante, Instituto Tecnológico de Buenos Aires.

Abstracto- Lenguajes de programación como el C y C++ otorgan amplias libertades al compilador ante comportamiento indefinido, resultando en comportamiento indefinido o undefined behavior. Este comportamiento puede resultar en errores que comprometen la funcionalidad de todo un programa y pueden pasar desapercibidos al lector, dificultando su detección. Las herramientas de análisis estático resultan de gran utilidad para el programador para identificar estos problemas. En esta monografía analizaremos el funcionamiento de dos tipos de herramientas estáticas, una de tipo léxico (Flawfinder) y otra de tipo semántico (CPPCHECK). El fin de este trabajo es ofrecer un panorama del funcionamiento de las herramientas de análisis estático para luego determinar en qué casos convienen usar a las herramientas estáticas de tipo léxico y estático.

***Términos índice*—Análisis estático, software, vulnerabilidad, compiladores.**

NOMENCLATURA

HAE	Herramienta de Análisis Estático.
ASA	Árbol de Sintaxis Abstracto.
ASC	Árbol de Sintaxis Concreto.
CWE	Common Weakness Enumeration™.
AE	Análisis Estático.

I. Introducción

Los lenguajes de programación como C y C++ brindan a los compiladores la libertad de generar códigos eficientes para determinados conjuntos de instrucciones a los que se llama *undefined behavior*. Pero las reglas que determinan el comportamiento indefinido son muy sutiles y los programadores pueden tener errores que llevarían a vulnerabilidades de seguridad.[1]

La guía de FAQ de C describe un "comportamiento indefinido" de esta forma:

"Cualquier cosa puede suceder; la norma no impone requisitos. El programa puede fallar al compilar, o puede ejecutarse incorrectamente (bloquearse o generar silenciosamente resultados incorrectos), o puede hacer fortuitamente exactamente lo que el programador pretendía." [2]

Es evidente que una definición tan ambigua en programación puede traer problemas. Para comprender mejor este concepto vamos a separarlo en dos categorías:

1) **Errores de programación**, como *buffer overflow*, desreferencias a *null* pointers y usos de memoria después de librear.
2) **Operaciones no portables**, implementaciones de hardware con pequeñas diferencias, como división por cero, instrucciones de shift y más.

La primera categoría de *undefined behavior* es responsabilidad del programador y evitable si se conocen bien las reglas del lenguaje, aunque a veces su detección pueda ser una tarea complicada. Mientras que la segunda categoría no es tan sencilla de identificar, ya que se requiere una comprensión no solo de la plataforma en la que estamos trabajando y de las plataformas donde se podría llegar a usar nuestro código, sino que además hay que tener en cuenta las optimizaciones de los compiladores y sus actualizaciones.

A. Conceptos Útiles

1) Optimizadores en compiladores de C++

Cambio incremental (refactorización) de software para aumentar su rendimiento, acercándolo a óptima. Hoy en día, los compiladores realizan muchas optimizaciones, y éstas ya no se encuentran a cargo del programador [3].

2) Análisis Léxico

Primera fase de compilación, el compilador toma el código fuente modificado por un preprocesador y divide sintaxis en serie de tokens [4]. A un analizador estático se lo llama lexer, tokenizer o analizador léxico.

3) Análisis Sintáctico (*parsing*)

Segunda fase de compilación, el compilador toma tokens generados en el análisis léxico y analiza conforme a las reglas de la gramática formal. Genera un concrete syntax tree como output.

4) Análisis Semántico

Tercera fase de compilación, el compilador verifica si el 'parse tree' generado en la segunda y analiza conforme a las reglas del lenguaje [5]. Utiliza una tabla de símbolos con nombres de las variables y sus tipos de datos correspondientes.

5) Lexeme

Un lexeme es una secuencia identificable de caracteres (por ej. char, class y while), literals (como números y strings), identificadores, operadores o caracteres de puntuación (como (y }).

6) Token

Un token es un objeto que describe un lexeme. Este tiene un tipo (operador, función, número, etc.) y un valor (los caracteres descriptos por el lexeme). Además, puede tener otra información como posición en el código y otros datos relevantes para los compiladores [6].

7) Árbol de Sintaxis Concreto

Representaciones sintácticas en forma de árbol de la estructura de una cadena de caracteres que refleja la sintaxis del lenguaje.

8) Árbol de Sintaxis Abstracto

Versiones simplificadas de la representación del código en forma de árbol. [7]

II. Análisis

Dada la problemática recurrente del *undefined behavior*, durante décadas, desarrolladores de software han buscado la forma de minimizar el impacto de estos detectando su existencia a tiempo. Para

esto resulta extremadamente útil la información brindada por el análisis estático**,** un análisis del programa en tiempo de compilación que informa a los programadores sobre la existencia de defectos y vulnerabilidades en el mismo. Las HAE son programas que se encargan de identificar estos problemas y generalmente se especializan en ciertos tipos de vulnerabilidades dado que existen diferentes tipos u objetivos de AE como resumimos en la fig. 1. Un beneficio crucial del AE es que, a diferencia de un programador, analiza todo el programa de forma objetiva y detallada, sin focalizar la atención en sectores del código de mayor importancia, sino analizando todos con idéntica minuciosidad [8].

A. Métodos de Análisis Estático

Las HAE más simples funcionan buscando patrones de caracteres predefinidos en el código fuente. Otras herramientas más complejas usan modelos para representar el código para luego analizarlo de una forma similar a como lo haría un compilador, pero con el añadido de focalizaciones específicas como las mencionadas en la fig. 1.

B. Enfoques de HAE

Es posible agrupar las herramientas de AE en tres categorías principales:

1) Herramientas con Léxicas (*lexical tools*)

Con este enfoque, las herramientas dividen el programa en una secuencia de tokens y buscan un conjunto predefinido de vulnerabilidad, funciones o patrones.

2) Herramientas Sintácticas (*syntatic tools*)

Las herramientas sintácticas funcionan de una manera análoga a las herramientas léxicas con la diferencia de que pueden funcionar con el token regular devuelto por el análisis léxico o crear un árbol de sintaxis abstracto a partir de dicha secuencia de tokens. Si se construye un ASA, la herramienta puede atravesarlo mientras realiza comprobaciones específicas en los nodos del árbol, reportando si encuentra una coincidencia de un patrón en el código.

3) Herramientas Semánticas (*semantic tools*)

Las herramientas semánticas se enfocan en la comprensión del código analizado. Esto se logra creando una representación abstracta de los estados del programa y realizando una simulación estática de su ejecución [9]. El análisis es tan profundo que la herramienta tiene comprensión completa del contexto y flujo entre procesos.

Es interesante notar cómo las herramientas semánticas construidas sobre herramientas sintácticas pueden obtener una profunda interpretación del código, con comprensión del contexto y el flujo entre procesos. Otros analizadores pueden obtener conocimiento incluso del lenguaje que se está analizando, teniendo en cuenta características propias del mismo. Por ejemplo, veamos el operador new de C++, éste nunca puede devolver un puntero a *null* debido a que siempre arroja una excepción en caso de que falle la asignación de memoria. Por lo tanto, una HAS puede tomar esta información y a partir de allí saber que un puntero recientemente asignado nunca puede ser nulo ni tampoco nombrado.

C. Herramientas de Análisis Estático para C++

C++ es un lenguaje de programación desarrollado originalmente por Bjarne Stroustrup a principios de la década de 1980. El lenguaje admite programación orientada a objetos, programación genérica de plantillas y paradigmas de programación funcional. Dado a que el lenguaje admite la aritmética de puntero arbitrario y una manipulación de flujo de control muy laxa utilizando instrucciones como goto, evita que las herramientas de análisis hagan representaciones precisas de la semántica [10]. Por lo tanto, debido a la naturaleza compleja de C ++, no es un objetivo fácil para el AE.

Actualmente, las herramientas C ++ existentes utilizan principalmente sus propios analizadores, lo que no es ideal ya que estos analizadores a menudo carecen de los medios para manejar algunas partes del lenguaje estándar. Esto puede ser un problema cuando se utilizan bibliotecas basadas en plantillas como BOOST y la propia biblioteca de plantillas estándar (STL). Los problemas de análisis a menudo pueden significar que la herramienta malinterpreta la sintaxis y la semántica de un programa.(Eli, B. 2009)

A continuación, analizaremos cómo herramientas logran sobrellevar estas complicaciones y detectan comportamiento indefinido (entre otros errores) en el AE. Nos concentraremos en dos HAE de código abierto, una herramienta de análisis léxico (FLAWFINDER) y una herramienta más completa como CPPCHECK que realiza análisis tanto léxico como semántico, centrándose en lo semántico.

III. FLAWFINDER

Flawfinder es un programa que mediante un análisis léxico de código en C y C++ puede detectar potenciales fallas de seguridad. A pesar de su simpleza es muy útil para cierto tipo de errores conocidos. La simpleza de Flawfinder se debe a que no entiende la semántica del programa, por ejemplo, no realiza análisis de flujo de datos (ver referencia a análisis de flujo de dato en CPPCHECK). El programa funciona haciendo un análisis del código y comparándolo con patrones de errores conocidos en una base de datos de errores integrada al programa. A diferencia de otras herramientas más sofisticadas, es muy eficiente sobre aquellas vulnerabilidades del código o errores que detecta [11].

El programa produce una salida de texto con una lista de descubrimientos, una lista de vulnerabilidades en el programa clasificados por riesgo con un puntaje del 0 (muy poco riesgo) hasta 5 (máximo riesgo). Estos puntajes son asignados por Flawfinder que trabaja usando una base de datos de funciones en C/C++ con errores o problemas típicos.

Algunos de los posibles problemas son: 1) riesgos de buffer overflow; 2) condiciones de carrera (*race condition*); 3) improper Input Validation; 4) buffer over-read; 5) integer overflow; 6) archivo temporario inseguro; 6) uso de posibles funciones peligrosas.

1. Type Checking: *Verificación del uso correcto de tipos de datos. (Ej. Desreferenciar puntero NULL)*

2. Bug Finding: *Búsqueda de patrones que lleven a comportamiento no intencionado.*

3. Security Vulnerabilities: *Detección de accesos a memoria sin permiso (Ej. Buffer overflow)*

4. Style Checking: *Verificación de cumplimiento de ciertas reglas de código y convenciones. (Ej. HIC++)*

5. Entendimiento: *obtención de información para visualizar formas en las que un programa puede funcionar.*

Figura 1. Mención de algunos de los usos posibles del AE. Los usos 1,2 y 3 son los más importantes a la hora de detectar comportamiento indefinido.

A. Diseño

Para correr Flawfinder se requiere Python 2.7 o Python 3 y funciona sobre sistemas de tipo Unix[1] y en Windows usando Cygwin[2].

1) Opciones

Para controlar la documentación, los datos de entrada, los errores encontrados, el formato de salida y la gestión de lista de resultados Flawfinder tiene un set de opciones que se activan según los 'Estándares GNU para interfaces de línea de comando'.

2) Lista de fallas de Software y Hardware

Common Weakness Enumeration™ es una lista de errores posibles desarrollada por una comunidad de programadores utilizada por Flawfinder. CWE incluye un total de 839 errores o debilidades posibles en programación [12]. Las descripciones de las fallas generalmente van incluidas con identificadores CWE.

3) Bugs (errores en el software)

El diseño de Flawfinder para buscar errores está enfocado en la búsqueda de errores basándose en la coincidencia de patrones de texto, por lo tanto está limitado a encontrar errores de manera léxica y esto es muy difícil de cambiar. Cuando una coincidencia entre el conjunto de reglas y el código fuente es encontrado, el nombre del error es agregado a un arreglo llamado **hitlist** que luego será evaluado elemento a elemento para comunicar al usuario sobre los hits encontrados.

La clase **Hit** contiene los siguientes miembros:

Hook: Función a la que se llama cuando el nombre del error fue encontrado.
Level: Nivel de riesgo de 0-5.
Warning: Texto describiendo el problema.
Suggestion: Texto sugiriendo posible corrección.
Category: Hay 4 categorías posibles (Buffer, race, tempfile, format).
Url: Fragmento URL de referencia.
Other: Un diccionario con otros parámetros de ajuste (nombre de función, parámetros, input, comienzo, final, nombre del archivo, línea y columna en el código, etc.)

B. Método

El primer paso que realiza la herramienta es el análisis sintáctico. Flawfinder utiliza un *parser* propio muy simple. La herramienta usa el escaneo léxico para encontrar tokens (como nombres de funciones) que sugieren vulnerabilidades probables. Es decir que durante el escaneo léxico es capaz de identificar errores y estimar el nivel de riesgo llamando a las funciones pertenecientes a la clase de cada *hit* (error) y guardando el resultado en un string llamado **patch_file.**

Flawfinder hace uso del módulo **re** de Python para encontrar la coincidencia de algún patrón de error con el código. **re** es un módulo construido en Python que provee funciones útiles para el manejo de expresiones [13]. El uso del método re.compile() hace simple encontrar la coincidencia de errores, tanto los propuestos por la lista CWE como otros propios de Flawfinder.

El extracto del archivo "flawfinder" (Fig. 2) facilita comprender la forma en que re.compile busca un patrón de scanf de bajo riesgo, utilizando caracteres especiales de Python, que no serán desarrollados en este trabajo (Para conocer tipo de caracteres consultar https://docs.python.org/3/library/re.html).

En primer lugar c_scanf(hit) es una función que recibe un objeto hit que será añadido a la hitlist llamando a otra función add_warning().

```
p_low_risk_scanf_format = re.compile(r'%[0-9]+s')

def c_scanf(hit):
  …
  if p_low_risk_scanf_format.search(source):
    # This is often okay, but sometimes extremely serious.
       hit.level = 1
       hit.warning = ("It's unclear if the %s limit in the format string is small
       enough (CWE-120)")
       hit.suggestion = ("Check that the limit is sufficiently small, or use a
       different input function")

   else:
        # No risky scanf request.
        # We'll pass it on, just in case it's needed, but at level 0 risk.
        hit.level = 0
        hit.note = "No risky scanf format detected."

    else:
        # Format isn't a constant.
        hit.note = ("If the scanf format is influenceable by an attacker, it's exploitable.")

 add_warning(hit)
```

Fig. 2. Extracto de programa Flawfinder (flawfinder). [14]

Podemos distinguir cómo la función c_scanf recibe el resultado de re.compile para luego editar el contenido del hit instanciado a partir del error y edita sus parámetros para ser añadido al patch_file.

C. Conclusiones de Flawfinder

Este primer análisis de la HAE Flawfinder sirve como introducción a herramientas semánticas y su funcionamiento. Debido a las limitaciones con las que cuenta Flawfinder, la utilización más común de esta herramienta es examinar los archivos fuente de alto riesgo de un proyecto. Se recomienda utilizar esta herramienta en conjunto con otras para la detección de errores, aun así Flawfinder es muy útil a la hora de determinar comportamiento *undefined behavior* en la lista de errores. Además, debido a su enfoque de comparación, es relativamente simple la adición de nuevos patrones de comportamiento indefinido que se descubren.

III. CPPCHECK

CPPCHECK es una herramienta de AE para código C / C ++ creada por Daniel Marjamäki. CPPCHECK proporciona un análisis de código único para detectar errores y se enfoca en detectar comportamientos indefinidos y construcciones de codificación peligrosas, y que además detecta otros errores como de estilo y errores de software. Esta herramienta está diseñada para poder analizar su código C / C ++ incluso si tiene una sintaxis no estándar (común en proyectos integrados). Este programa se puede utilizar en forma gráfica y en versión por línea de comando.
CPPCHECK ha sido utilizado para la verificación del código en dispositivos de lectura de detectores de partículas de alta energía, [15] software de monitoreo del sistema para radiotelescopios [16], así como en el análisis de errores de grandes proyectos.

Esta herramienta crea su propio ASA simplificado, llevando a cabo un análisis léxico propio, aunque tiene la desventaja de requerir constantes actualizaciones del estándar de C++ para llevarlo a cabo correctamente. CPPCHECK sigue siendo mejorada al mes de Mayo 2020.

[1] Fue probado en GNU/Linux. (https://dwheeler.com/flawfinder/)

[2] https://www.cygwin.com/

El analizador viene con diferentes verificaciones que son categorizadas según su gravedad. Las opciones del programa están disponibles para habilitar o deshabilitar los controles según su gravedad.

Las posibles verificaciones son: 1) error para problemas graves como errores de sintaxis; 2) advertencia sobre posibles problemas; 3) estilo para código muerto y otros problemas estilísticos; 4) sugerencias relacionadas con el rendimiento; 5) problemas de portabilidad de plataforma; 6) mensajes informativos sobre problemas con el análisis; 7) otros. [17]

A. Diseño

La data interna utilizada por las verificaciones (*checks),* consiste en la lista de *Tokens*, el ASA y el *Symbol Database*. En un comienzo los archivos fuente son pre-procesados y los verificadores no tienen acceso al código directo ni al código proveniente directamente del preprocesador. Los verificadores acceden a una lista de tokens y pueden hacer ciclos en esta lista. Estos tokens son usados como nodos del ASA creado. El analizador construye este árbol usando su propio parser y análisis léxico, lo cual puede llevar a no ajustarse al estándar de C++ más reciente en caso de faltas de actualizaciones.
CPPCHECK implementa un análisis del flujo de datos para poder realizar verificaciones elaboradas. Este análisis es un método utilizado por compiladores para encontrar código muerto y realizar optimizaciones, además puede ser utilizado por verificadores individuales.

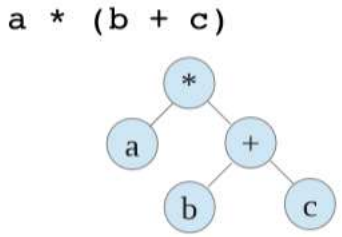


Fig. 3. Ejemplo de ASC con tokens obtenidos de operación a * (b + c). [18]

1) Clase Token:
La lista de tokens es una lista enlazada de la clase Token llamada TokenList. Los tokens se guardan como strings, la razón por la que no se usa la clase String de C++ se debe a funciones extras de la clase Token que necesitan los tokens como la ubicación donde están guardados, funciones para clasificarlos y funciones para su manejo.
Las expresiones regulares son usadas para encontrar defectos comparando los tokens con arreglos de patrones. Por ejemplo, en la función **simpleMatch** (token.h [19]):

static bool simpleMatch(const Token *tok, const char pattern[]);

La función simpleMatch compara un token, o lista de tokens, con una lista de patrones. Los patrones se pueden combinar para comparar múltiples tokens a la vez, por ejemplo, un patrón posible es “ **) void {**“. La función match al encontrar al token ‘ **)** ’ devolverá ***true*** si el token siguiente en la lista es ‘ **void** ’ y el siguiente a este es ‘ **{** ’. En la figura 4 mostramos un ejemplo del uso de las funciones **tokens**() y **next**(), siendo la primera una función que devuelve un puntero al token a analizar y next devuelve un puntero al próximo token.
Dentro de la clase token se utilizan otras clases para manejar la información de los tokens, por ejemplo, Function, Variable y Type como más relevantes, que pertenecientes a la base de datos de los símbolos (symboldatabase.h [20]).

```
// Loop through all tokens
for (const Token *tok = _tokenizer-> tokens();  tok ; tok = tok->next())
  {
     ...
  }
```

Fig. 4. Ejemplo de ciclo para analizar tokens del ASA obtenido en el análisis sintáctico.

2) Symbol database:
La base de datos contiene información sobre tipos, funciones, variables del código objeto y se logra mediante clases. Las verificaciones pueden usar estas clases cuando se identifican funciones, tipos, variables en la lista de tokens y obtener información de ellos.
Las clases definidas en la base de datos se encargan de guardar toda la información relevante de aquellos tokens o conjunto de tokens que representan funciones, variables y tipos (argumentos, alcance, tipo retornado, accesos, argumentos, si es virtual o no, entre, nombre, acceso, tipo, alcance, entre otros).
La fig. 5 muestra como un verificador puede detectar si una función es estática y la fig. 6 muestra cómo puede investigar una variable:

```
// Hay una llamada a función aquí?
if (Token::Match(token, “%type% (“)) {
      //Se busca información de la función llamada
      Const Function (function = token ->function();

      // Verificar si la función es estática..
      if (function && function->isStatic()) {
            …es una función estática…
      }
}
```

Fig. 5. Verificador analiza función desde token.

```
//Adición de variable
if (Token::Match(token, “%var% +”)) {
      //Se busca informacion de la variable
      const Variable *var = token ->variable();

      //Es la variable un puntero?
      if ( var && var->isPointer()) {
            …es un puntero…
      }
```

3) ValueFlow:
Antes de llevar a cabo las verificaciones que buscan errores en el código se realiza un análisis sensible al contexto del código y se rastrean todos los posibles valores de variables en múltiples direcciones. A este análisis lo llamamos análisis de flujo y los resultados se almacenan en la estructura de cada *token* y cada uno de estos contendrá todos los posibles valores que puede poseer. Por ejemplo, una función **foo** presente en el código fuente (Fig. 7), es llamada dos veces con los valores 2 y 4.

```
// Extracto de Código Fuente
  void foo(int x) {
  int a = 1 + x * x; }
  foo(2);
  foo(4);

En los tokens de la función foo el ValueFlow guardará lo siguiente:
   1: {1}
  +: {5,17}
  x: {2,4}
  *: {4,16}     //Notemos que los resultados se guardan en los tokens de operadores.
  x: {2,4}
```

Fig. 7. Ejemplo de funcionamiento de ValueFlow con función foo.

El *ValueFlow* se encarga de guardar esos valores y los posibles resultados en los tokens de la función foo.
Siguiendo esta lógica podemos entender cómo la 'división por cero' puede ser atrapada por un *check (Fig. 8.)*

```
// Tenemos un token, es el operador division?
  if (token->str() == "/")) {

    // Tomamos el operando del lado derecho (OLD) del operador división
    const Token *rhs = token->astOperand2();

    // Tomamos el valor "0" del OLD si existe
   const ValueFlow::Value *val0 = rhs->getValue(0);

    // Es el valor de rhs "0"?
   if (val0 != NULL) {
      … si hay un "0"...      }    }
```

Fig. 8. Ejemplo de uso de tokens para atrapar división por cero [18].

B. Método

1. Preparación de Código Fuente para el análisis

Al igual que los compiladores, los analizadores estáticos complejos primero llevan a cabo la fase del análisis léxico para luego generar un *CST*. Luego las HAE normalizan esta información creando un *ASA*, omitiendo nodos del árbol que representan marcas de puntuación y diseñándolo para facilitar la búsqueda de los errores que se desean encontrar. Luego se avanza a la fase de análisis sintáctico y las HAS utilizan la tabla de símbolos resultante junto con el ASA para analizar el flujo del programa y el flujo de datos (Penttila, 2014).

2. Verificaciones (checks)

Las verificaciones son la parte vital de la herramienta estática y en el caso de CPPCHECK tenemos una extensa lista de verificaciones implementadas tanto por la línea de comando como por la GUI disponible. La lista de verificaciones de CPPCHECK completa es dada en el apéndice 4, entre dichas verificaciones las más relevantes para el comportamiento indefinido son:

64.-bit portability: Asignación de dirección con *int´s y long´s*.
Casteo de direcciones de *int.*
Bounds checking: Problemas de límites que llevan a comportamiento indefinido de tipo error de programación.
IO using format string: Se verifica el formato de operaciones de I/O
Condiciones: Pruebas inválidas para *overflow* (por ejemplo, 'ptr + u < ptr'). Una condición es siempre falsa a menos que haya *overflow*, llevando a *undefined behavior.*
División por cero
De referenciación de puntero null
Pasar puntero NULL a una función con argumentos variables.

Hay dos formas en las que CPPCHECK puede crear verificaciones, una es a través de un documento XML con expresiones regulares que se usan de forma similar a las herramientas léxicas, comparando las expresiones con los tokens. El otro método es mediante el uso de clases que acceden como visitantes a los nodos del ASA.
La clase base **Cppcheck**[21] es usada por otras clases para realizar el AE de C++ y encontrar posibles errores o lugares donde el código puede ser mejorado. La función que empieza evaluando todas las verificaciones es **check**:

unsigned int Cppcheck::check (const std::string &path)

Esta función recibe un camino al archivo a analizar y devuelve la cantidad de errores o un 0 si no se encontró ninguno.
Para comunicarse con las diferentes verificaciones se usa la clase interfase **Check [22]**. La clase padre **Check** es heredada por todas las verificaciones y posee funciones virtuales que son sobreescritas por ellas. El diagrama de herencia para Check está dado en el apéndice 1.
La función virtual pura más importante de **Check** es:
virtual void runChecks(const Tokenizer*,const Settings*, ErrorLogger *) = 0;

Tokenizer es una clase cuyo propósito es tokenizar el código objeto (*AL*) y además contiene funciones que simplifican la lista de tokens. **Settings** es una clase contenedor para ajustes. **errorLogger** es una interfase donde diferentes clases responsables pueden cargar errores, es implementada por la clase base Cppcheck.
La función **runChecks** es sobreescrita por las clases de errores y utilizan otras funciones para realizar las verificaciones.

C. Conclusiones de CPPCHECK

En esta visión general de la manera en la que acciona CPPCHECK para analizar un código podemos ver cómo su enfoque con clases para errores se convierte en una herramienta muy versátil que permite encontrar errores que compiladores no identifican.

Esta extensa HAE es muy completa, pero de todas formas no se puede confiar solamente en CPPCHECK ya que su análisis está limitado a las clases de verificaciones existentes. Aun así existen errores que sólo CPPCHECK puede detectar comparado a otros compiladores tradicionales, por lo que es una excelente herramienta extra para mejorar la calidad de software. La actualización regular de CPPCHECK permite que con cada actualización se llegue a menos falsos positivos. De todas formas, el código fuente de CPPCHECK no está bien diseñado para las reglas del preprocesador [23]. Es argumentable que el uso de métodos propios para la creación de un ASA es costoso en cuanto a trabajo y siendo posible usar otros *parsers* más elaborados ya existentes para la creación de tokens.

En cuanto al comportamiento indefinido, la mayor ventaja de CPPCHECKER comparado a otros analizadores estáticos, es que la constante interacción entre usuarios y programadores facilitan que cada año se reporten más casos de comportamiento indefinido y se prosiga a editar la herramienta para detectarlos.[24][25].

IV. Resultados

Para su análisis, las herramientas fueron probadas en tres proyectos de la materia Algoritmos y Estructuras de Datos y en la librería Dear ImGui. De los proyectos A, B y C se probaron dos versiones de cada uno.

1. *Proyectos*

A: Simulación de blobs, trabajo introductorio a clases.
B: Programa orientado a eventos que simula desplazamiento y salto de dos gusanos.
C: Network de servidor y cliente para el intercambio de archivos.

CPPCHECK encontró una mayor cantidad de errores en todos los proyectos, aunque en su mayoría de estilo. A diferencia de Flawfinder supo detectar errores de rendimiento en los trabajos A1 y C1, y se recomendaron dos cosas: 1) pasar valores a funciones por referencia constante, 2) utilizar la función de la librería boost[3] start_with() para comparar dos secuencias de caracteres en vez de la función find() de la librería string que es más lenta. Las advertencias que fueron

[3] https://www.boost.org/

detectadas en los proyectos se debieron a la falta de inicialización de miembros de una clase en el constructor.

Flawfinder detecto en su mayoría vulnerabilidades respecto al memoria en los proyectos, categorizando estos errores con puntaje de 2 o 4. *Flawfinder* encontró más errores que CPPCHECK en los proyectos C1 y C2 relacionados al error CWE-78, una vulnerabilidad de seguridad del programa frente a un ataque, dado a que en ambos proyectos no se utiliza un mecanismo de neutralización para la verificación de la data recibida por el cliente. Flawfinder tuvo severos falsos positivos en los proyectos C, señalando errores de sincronización y condiciones de carrera (CWE-78). Estas advertencias fueron falsas dado a que el análisis léxico realizado por la herramienta limita su entendimiento de la librería utilizada para la sincronización de eventos en el programa.

Respecto al comportamiento indefinido CPPCHECK lleva una amplia ventaja en cuanto a cantidad, detectando principalmente errores relacionados a la falta de inicialización de miembros de clases. Pero Flawfinder encontró en los proyectos A1, C1 y C2 errores relacionados con el manejo de memoria y la falta de verificación de tamaños cuando se copia información (CWE-120/CWE-119). Estos errores de nivel cuatro y dos en la escala de Flawfinder de peligro no fueron detectados por CPPCHECK.

TABLA I
RESULTADOS PARA CPPCHECK

Tipos de error:	ESTILO	Advertencias	Rendimiento	Posible Undefined Behavior
A.1	21	12	1	12
A.2	9	1	0	1
B.1	4	1	0	1
B.2	2	0	0	0
C.1	3	0	4	0
C.2	3	3	0	3
Total	42	17	5	17

Tabla 1. Resultados obtenidos con CPPCHECK para los proyectos A1, A2, B1, B2, C1 y C2.

TABLA II
RESULTADOS PARA FLAWFINDER

Nivel de error:	0	1	2	3	4	5	Posible Undefined Behavior
A.1	0	0	0	1	2	0	2
A.2	0	0	0	1	0	0	1
B.1	0	0	0	1	2	0	2
B.2	0	0	0	0	0	0	0
C.1	0	0	2	0	6	0	8
C.2	0	0	4	0	6	0	9
Total	0	0	6	2	18	0	8

Tabla 2. Resultados obtenidos con Flawfinder para los proyectos A1, A2, B1, B2, C1 y C2.

Los resultados de las tablas I y II sugieren una amplia ventaja a CPPCHECK, pero analizando la relevancia de los errores detectados podemos concluir que estos resultados no pueden pensarse sólo en términos de cantidad. Notemos que las vulnerabilidades detectadas por Flawfinder relacionadas a el manejo de memoria son más graves, y CPPCHECK no pudo detectar esas vulnerabilidades.

2. *Librería ImGui*

Dear ImGui es una librería de código abierto que provee una interfaz gráfica para programas desarrollados en C++. Está diseñada para permitir iteraciones rápidas y crear herramientas de creación de contenido y de visualización. [26] Desarrollada independientemente por Omar Cornut[4] esta librería sigue estando en desarrollo y es modificada todos los años. Resulta interesante la cantidad de errores y posible comportamiento indefinido encontrado en esta librería por las herramientas estáticas que fueron utilizadas.

Por ejemplo, CPPCHECK encontró 4 errores de portabilidad, todos relacionados con el uso de la función **memset** en estructuras que contienen datos de tipo *float*. El error subyace en el hecho que *memset* convierte los valores pasados para la inicialización de un campo *float* de la estructura a *char*, por lo tanto, utiliza el byte menos significativo.

TABLA III
RESULTADOS PARA CPPCHECK

Tipos de error:	ESTILO	Adv.	Rendimiento	Portabilidad	Error	Posible Undefined Behavior
Dear ImGui	318	4	10	4	5	4
Errores totales	345					

Tabla 3. Resultados obtenidos en el análisis de Dear ImGui con CPPCHECK.

TABLA IV
RESULTADOS PARA FLAWFINDER

Nivel de error:	0	1	2	3	4	5	Posible Undefined Behavior
Dear ImGui	12	38	150	1	24	0	65
Errores totales	225						

Tabla 4. Resultados obtenidos en el análisis de la librería Dear ImGui con Flawfinder.

En el caso de Imgui no presenta problema grave dado a que se inicializa en 0:

```
ImGuiInputTextState() { memset(this, 0, sizeof(*this)); }
```
Extracto de "imgui_internal.h" [27]

De todas formas, esto puede llegar a generar problemas con futuras optimizaciones del compilador utilizado actualmente para la librería. Este error no fue detectado por Flawfinder.
Otras vulnerabilidades detectadas por CPPCHECK fueron 2 advertencias de acceso y 2 de efectos colaterales indeseados. Las primeras dos advertencias son correctas ya que en el código fuente el diseñador de ImGui aclara que dicho error es correcto pero que no genera problema significativo alguno. Los efectos colaterales señalados por las otras dos advertencias son detectados por CPPCHECK dado a la llamada a la función **IM_ASSERT**:

```
IM_ASSERT(my_str->begin() == data->Buf);
```
Extracto de "imgui_demo.cpp" [28].

CPPCHECK indica que el problema se debe a que *begin()* se llama dentro de la declaración de aserción, por lo que en versiones del código que utilizan optimizaciones, la función en la declaración no se ejecuta. El resto de las advertencias de CPPCHECK no son posibles fuentes de comportamiento indefinido, sino simples sugerencias para mejorar el diseño del código y su rapidez.
En la librería Flawfinder pudo encontrar 225 errores, los cuales en su mayoría son de nivel dos, relacionados a errores de buffer (CWE-120). Los errores relacionados a memoria, son debidos a falta de verificación de buffer *overflows* y el resto son debido a arreglos con tamaños definidos estáticamente, Flawfinder advierte que estos pueden ser restringidos incorrectamente por el programador, pero no los

[4] http://www.miracleworld.net/

consideraremos posibles fuentes de comportamiento indefinido. Por el otro lado, la falta de verificación del buffer sí compromete a la correcta funcionalidad del programa debido a la sutileza de las acciones que pueden desencadenarlos. También hay dos errores de nivel dos que pueden ser fuentes de *undefined behavior* ya que señalan operaciones que pueden resultar en *integer overflow*. (CWE 190) A estos dos errores los consideramos como fuentes de comportamiento indefinido. En total hay 65 errores de este tipo que consideramos como fuentes de posible comportamiento indefinido en la tabla IV.

Los errores de nivel cuatro detectados por Flawfinder son debidos al uso de la función *scanf* ya que advierte que sin una especificación de límite la función puede permitir *buffer overflow* y la entrada de información que no es la esperada por la aplicación. (CWE 20). El error de nivel tres encontrado se debe al uso de la función *srand* ya que Flawfinder advierte que no es suficientemente aleatoria para funciones relacionadas con la seguridad. Como en este programa la función *srand* no se utiliza para ese tipo de funcionalidades, no consideraremos a ese error relevante, a pesar de ser información útil de una función de C/C++. Por más que sean graves los errores, no son fuente de *undefined behavior*.

3. *Discusión*

Es evidente que CPPCHECK es una herramienta completa en cuanto a la variedad de tipos de errores que es capaz de detectar, aun así, no es capaz de detectar algunas vulnerabilidades que son señaladas en Flawfinder.

1) ErroresCWE

Flawfinder es una herramienta oficialmente compatible con CWE y enfocada en estos errores, pudiendo tener así una precisión mayor que CPPCHECK en estos errores. Varios estudios comparativos han confirmado esta conclusión con extensos análisis donde Flawfinder resultó poseer un mayor índice de detección de errores. En un estudio para la compañía de seguridad informática Bitdefender[5] de 2016 [29] CPPCHECK mostró reportar alguno de los 25 errores más peligrosos en 32 de 652 oportunidades. Otro análisis comparativo *del International Journal of Computer Applications* [30] agrupó los errores de CWE en 16 categorías de vulnerabilidades y demostró que Flawfinder encontró vulnerabilidades en 9 de ellas mientras que CPPCHECK en 7. Tomando en cuenta los tipos de errores que las dos herramientas detectaron, Flawfinder encontró, en su mayoría, vulnerabilidades relacionadas con el manejo de memoria y *buffer overflow*, una importante fuente de comportamiento indefinido. Mientras que CPPCHECK detectó en su mayoría errores de estilo y no supo detectar la mayoría de las vulnerabilidades en manejo de memoria detectadas por la otra herramienta.

Podemos confirmar que en cuanto a la cantidad de reportes de errores CWE Flawfinder le lleva una ventaja a CPPCHECK. Es importante considerar que ambas herramientas tienen significativos falsos positivos, pero en este caso de errores CWE es preferible la notificación de un falso positivo antes que pasarlo inadvertido.

2) Undefined behavior

Es compleja la comparación en este caso, ya que mientras Flawfinder detectó vulnerabilidades de manejo de memoria, CPPCHECK detectó errores de portabilidad severos en la librería ImGui. A pesar de que el comportamiento indefinido es una prioridad para la herramienta CPPCHECK [31], la simpleza del análisis léxico de Flawfinder logra enfocarse en ciertos errores, aunque resultando también en muchos falsos positivos. En lo que respecta a comportamiento indefinido podemos concluir que ambas herramientas tienen sus ventajas y debilidades por lo que no hay una herramienta superior a la otra en este criterio.

3) Uso

La interfaz gráfica de CPPCHECK es simple de usar y a diferencia de Flawfinder muestra el código fuente con los errores encontrados. Además, CPPCHECK provee comentarios con recomendaciones y explicaciones útiles para cada error reportado, como en los ejemplos dados en el apéndice 5. Una ventaja de la interfaz es que permite seleccionar múltiples proyectos a la vez y de diferentes plataformas. Flawfinder funciona por línea de comando y no es muy descriptivo en las vulnerabilidades detectadas, pero al mostrar los códigos CWE de los errores, el usuario tiene información detallada de ellos disponible. Una ventaja de Flawfinder en comparación a CPPCHECK es la escala de gravedad de los errores reportados.

4) Velocidad y falsos positivos

Los proyectos analizados presentaron falsos positivos solo frente al análisis de la herramienta Flawfinder. Mientras que en CPPCHECK todas las advertencias fueron válidas. Este resultado era esperable ya debido a la velocidad del análisis léxico realizado por Flawfinder. El análisis realizado por la herramientas léxicas es más veloz que las herramientas estáticas (Shrestha, 2013), con la desventaja de que produce falsos positivos dado a que ignora el flujo de data del programa y no detecta si una función fue usada de manera segura o peligrosa.

La librería ImGui presentó numerosas vulnerabilidades, de las cuales resulta complejo el análisis de falsos positivos debido al extenso y complejo diseño de la librería.

5) Otros enfoques

Las dos HAE analizadas representan dos modelos de herramientas populares y nos son útiles para mostrar el funcionamiento típico del análisis estático. De todas formas, en los últimos años un nuevo modelo para la detección de comportamiento indefinido ha estado progresando. Desde el Laboratorio de Ciencias de la Computación e Inteligencia Artificial del MIT[6] se propuso un nuevo modelo enfocado en detectar código inestable en términos de optimizaciones de compilador que puedan llevar a comportamiento indefinido [32]. Este modelo sigue en proceso de desarrollo y ya están disponibles herramientas como STACK [33], que basándose en optimizadores para encontrar código inestable detecta posibles errores que herramientas estáticas no llegan a comprender.

V. Conclusión

El objetivo primordial de esta monografía fue desarrollar el uso de herramientas estáticas para la detección de comportamiento indefinido y comparar dos modelos diferentes de herramientas, exponiendo sus ventajas y sus desventajas. Para lograr comprender las ventajas de cada uno fue vital comprender el funcionamiento interno de las herramientas para comprender cómo se obtienen las vulnerabilidades presentes.

En el caso de Flawfinder, entender el funcionamiento de esta herramienta léxica nos demuestra lo simple que puede llegar a ser una HAE y además lo útil que es en encontrar patrones ya establecidos y conocidos de errores en los programas que un programador puede no identificar. Pero, debido a su enfoque de comparación de patrones, esta herramienta es significativamente limitada en cuanto a los errores que pueda detectar y suele reportar falsos positivos.

CPPCHECK es una herramienta competa y su diseño permite comprender cómo funciona el análisis estático y el manejo de errores con estructuras del C++. La mayor ventaja de la herramienta es que al tener clases propias para diferentes tipos de vulnerabilidad se puedan editar clases existentes o incluso crear nuevas clases con errores que

[5] https://www.bitdefender.com/

[6] https://www.csail.mit.edu/

no sean detectados correctamente todavía. CPPCHECK realiza un análisis sensible al flujo del programa, a diferencia de otras herramientas que utilizan un análisis sensible a los caminos basándose en la interpretación abstracta. Este enfoque, aunque no sea el mejor en la teoría, permite que dicha herramienta pueda detectar errores que otras no y lo consideramos como una ventaja.

La comparación teórica entre las dos herramientas, al ser modelos diferentes, no resulta muy relevante. Pero el análisis práctico desarrollado en este ensayo arrojó como resultado que ninguna de las dos herramientas funciona como reemplazo de la otra. Además, como mencionamos en el análisis, no se descarta la existencia de otros modelos de herramientas que pueden ser diferentes o mejores a las dos que mencionamos. De esta forma concluimos que la mejor manera de detectar vulnerabilidades en programas es utilizando una combinación de modelos diferentes de herramientas, para así sacar el mejor provecho de cada una. Ya que es la superposición de resultados lo que permite un análisis minucioso de errores posibles y su corrección.

De todas formas, podríamos sugerir que para programas o partes de código donde el correcto manejo de memoria es fundamental, se utilice Flawfinder ya que es el que mejor detecta errores de este tipo. Por el otro lado, CPPCHECK podría ser usado para el análisis de proyectos donde el rendimiento (en términos de memoria utilizada y tiempo de ejecución) sea un factor importante, ya que la herramienta puede sugerir cambios significativos para mejorar el desempeño del programa.

Respecto al comportamiento indefinido, ninguna HAE es perfecta, los errores que lleven a *undefined behavior*, aunque detectables en la mayoría de los casos, no siempre son evitables. Siempre será responsabilidad del programador desarrollar software a conciencia siguiendo buenas prácticas de programación y luego utilizar las herramientas de análisis que detecten problemas que no son fáciles de detectar. Las sugerencias de las herramientas estáticas son una buena forma de aprender sobre vulnerabilidades en los programas y en particular aquellos errores que pueden llevar a comportamiento indefinido.

VI. Futuras investigaciones

Las herramientas de análisis estático son mecanismos con continuas actualizaciones, por lo que la evaluación de estas también debe ser puesto al día según los avances. Por lo tanto, el análisis de las herramientas es también un área susceptible a modificaciones futuras, que demandará la investigación de los programadores antes de seleccionar una o más herramientas para sus proyectos.

Referencias

[1] JuliaSoft Srl. (2018). Understanding Semantic Static Analysis: The Scientific Method behind Julia and Why It Makes a Difference. Recuperado de: https://juliasoft.com

[2] Summit, S. (1996). C Programming FAQs. Recuperado de htt.p://c-faq.com/ansi/undef.html

[3] Godbolt, M., “Optimizations in C++ Compilera,” vol. 17, issue 5, Nov. 2019.

[4] Compiler Design – Lexical Analysis. Telagana, India.: Tutorials Point. Recuperado de: https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm

[5] Eli, B. (2009, Febrero 16). Abstract vs. Concrete Syntax Trees. Recuperado de: https://eli.thegreenplace.net/2009/02/16/abstract-vs-concrete-syntax-trees/

[6] Tchirou, F. (2017, Marzo 5). Hackernoon. Recuperado de Lexical Analysis: https://hackernoon.com/lexical-analysis-861b8bfe4cb0

[7] Lithmee. (13 de Agosto de 2019). What is the Difference Between Syntax Analysis and Semantic Analysis. Recuperado de Pediaa: https://pediaa.com/what-is-the-difference-between-syntax-analysis-and-semantic-analysis/#Syntax%20Analysis

[8] E. Penttila, “Improving C++ Software Quality with Static Code Analysis” M.S. tesis, Dept. Comp. Sci. and Eng., Aalto University, Espoo, 2014.

[9] The Programming Reaserch Group. (2003). High-Integrity C++ Coding Standard Manual.

[10] J. Viega et al. “ITS4: a static vulnerability scanner for C and C++ code”. In: Proceedings of the 16th Annual Computer Security Applications Conference. 2000. doi: 10.1109/ACSAC.2000.898880.

[11] Wheeler, D. (n.d.). Flawfinder. Recuperado de: https://dwheeler.com/flawfinder/

[12] CWE Team. (2020, Febrero 20). Common Weakness Enumeration. Retrieved from https://cwe.mitre.org/index.html

[13] Python Software Foundation. (n.d.). Regular expression operations. Retrieved from https://docs.python.org/3/library/re.html

[14] Wheeler A. D., (2017), “flawfinder”, repositorio de GitHub, https://github.com/david-a-wheeler/flawfinder/blob/master/flawfinder.

[15] S. Müller-Klieser, “Entwurf und Implementierung eines adaptiven, strahlentoleranten eingebetteten Systems am Beispiel eines Read-Out-Controllers” M.S. tesis, Fak. Phy. und Ast., Universität Heidelberg, Heidelberg, 2009.

[16] Martin Ettl et al.: The Wettzell System Monitoring Concept and First Realizations, IVS 2010 General Meeting Proceedings, p.444–448. Recuperado de: http://ivscc.gsfc.nasa.gov/publications/gm2010/ettl.pdf

[17] Cppcheck team. (2014). Cppcheck manual.

[18] Cppcheck team. (2014). Cppcheck design.

[19] Marjmaki, D., (2020), "Cppcheck", repositorio de GitHub, https://github.com/danmar/Cppcheck/blob/master/lib/token.h

[20] Marjmaki, D., (2020), "Cppcheck", repositorio de GitHub, https://github.com/danmar/Cppcheck/blob/master/lib/symboldatabase.h

[22] Marjmaki, D., (2020), "Cppcheck", repositorio de GitHub, https://github.com/danmar/Cppcheck/blob/master/lib/Cppcheck.h

[22] Marjmaki, D., (2020), "Cppcheck", repositorio de GitHub, https://github.com/danmar/Cppcheck/blob/master/lib/check.h

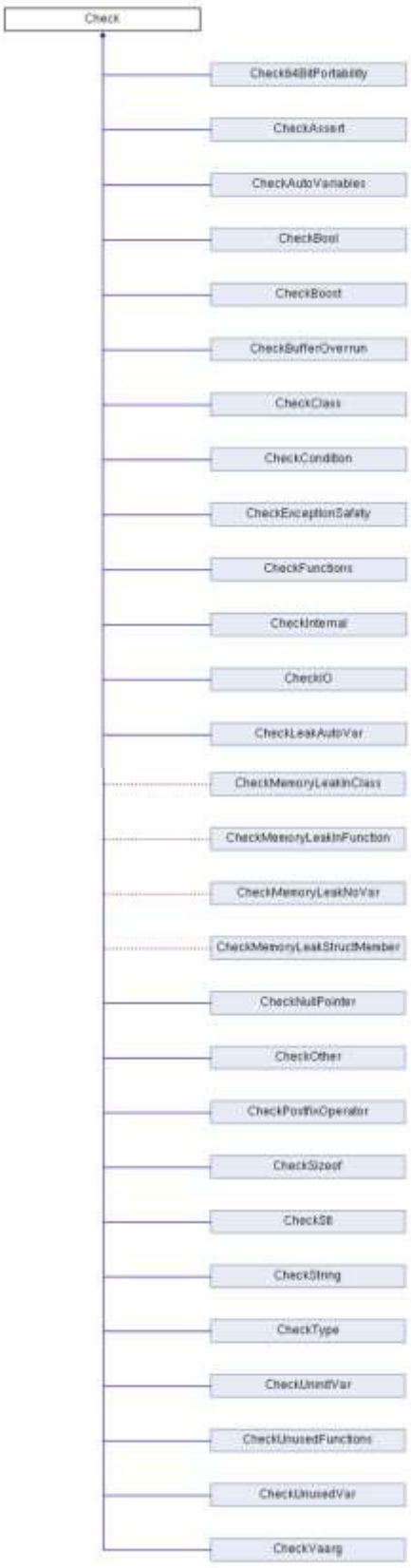
[23] Marjamaki, D. (19 de Mayo de 2016). Sourceforce. Obtenido de Cppcheck Discussion: https://sourceforge.net/p/Cppcheck/discussion/general/thread/165eb859/?limit=25#1266

[24] Marjamaki, D. (17 de Septiembre de 2017). Sourfeforge. Obtenido de Cppcheck discussion: https://sourceforge.net/p/Cppcheck/discussion/development/thread/901deffe/

[25] Marjamaki, D. (22 de Mayo de 2017). Sourceforge. Obtenido de Cppcheck Discussion: https://sourceforge.net/p/Cppcheck/discussion/general/thread/b3fd2484/?limit=25#fd7f
[26] Cornut, O. (2020),"ImGui" repositorio de GitHub, https://github.com/ocornut/imgui.
[27] Cornut, O. (2020),"ImGui" repositorio de GitHub, https://github.com/ocornut/imgui/blob/master/imgui_internal.h
[28] Cornut, O. (2020),"ImGui" repositorio de GitHub, https://github.com/ocornut/imgui/blob/master/imgui_demo.cpp
[29] Arusoaie A.,Ciobaca S., Craciun V., Gavrilut D. y Lucanu D., "A Comparison of Static Analysis Tools for Vulnerability Detection in C/C++ Code", Fac. Comp. Sci., Alexandru Ioan Cuza University, 2017.
[30] Hanmeet K B. y Puneet J. K., "Comparing Detection Ratio of Three Static Analysis Tools," International Journal of Computer Applications, vol. 124, no. 13, Agosto 2015.
[31] Cppcheck Team. (n.d.). Cppcheck. Retrieved from http://Cppcheck.sourceforge.net/
[32] Wang X., Zeldovich N., Kaashoek M. F. y Solar-Lezama A., "Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior", CSAIL, Massachussetts Institute of Technology, 2013.
[33] Wang X., (2013), "stack", repositorio de GitHub, https://github.com/xiw/stack/.

APÉNDICE

APÉNDICE 1. DIAGRAMA DE REFERENCIA PARA CLASE CHECK



APÉNDICE 2. ERRORES DE COMPORTAMIENTO INDEIFNIDO DETECTADOS POR FLAWFINDER

CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer (a parent of CWE-120, so this is shown as CWE-119!/CWE-120)
CWE-120: Buffer Copy without Checking Size of Input (‘‘Classic Buffer Overflow’’)
CWE-134: Uncontrolled Format String
CWE-190: Integer Overflow or Wraparound
CWE-190: Integer Overflow or Wraparound
CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization (‘‘Race Condition’’)
CWE-377: Insecure Temporary File
CWE-785: Use of Path Manipulation Function without Maximum-sized Buffer (child of CWE-120, so this is shown as CWE-120/CWE-785)

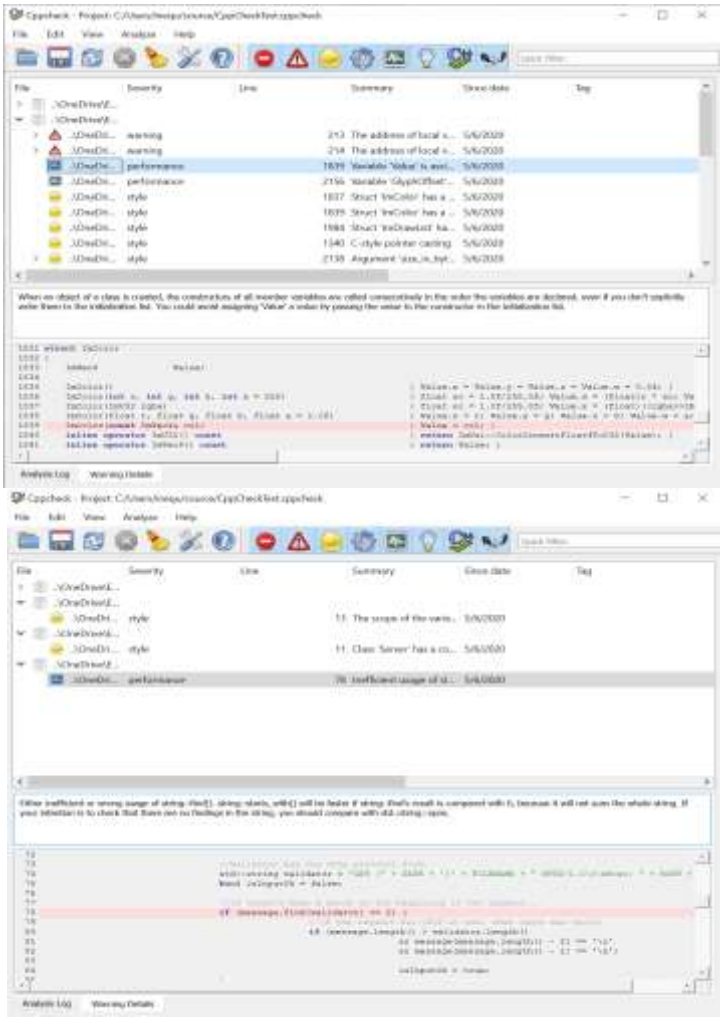
APÉNDICE 3. LISTA DE CWE DE LOS 25 ERRORES MÁS PELIGROSOS (2019).
Recuperado de: https://cwe.mitre.org/top25/archive/2019/

Ranking	Identificación CWE	Descripción
[1]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer
[2]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[3]	CWE-20	Improper Input Validation
[4]	CWE-200	Information Exposure
[5]	CWE-125	Out-of-bounds Read
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[7]	CWE-416	Use After Free
[8]	CWE-190	Integer Overflow or Wraparound
[9]	CWE-352	Cross-Site Request Forgery (CSRF)
[10]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[11]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[12]	CWE-787	Out-of-bounds Write
[13]	CWE-287	Improper Authentication
[14]	CWE-476	NULL Pointer Dereference
[15]	CWE-732	Incorrect Permission Assignment for Critical Resource
[16]	CWE-434	Unrestricted Upload of File with Dangerous Type
[17]	CWE-611	Improper Restriction of XML External Entity Reference
[18]	CWE-94	Improper Control of Generation of Code ('Code Injection')
[19]	CWE-798	Use of Hard-coded Credentials
[20]	CWE-400	Uncontrolled Resource Consumption
[21]	CWE-772	Missing Release of Resource after Effective Lifetime
[22]	CWE-426	Untrusted Search Path
[23]	CWE-502	Deserialization of Untrusted Data
[24]	CWE-269	Improper Privilege Management
[25]	CWE-295	Improper Certificate Validation

APÉNDICE 4. ERRORES DE COMPORTAMIENTO INDEIFNIDO DETECTADOS POR CPPCHECK

64-bit portability
Assert: Warn if there are side effects in assert statements (since this cause different behaviour in debug/release builds).
Out of bounds checking:
Array index out of bounds
Pointer arithmetic overflow
Buffer overflow
Dangerous usage of strncat()
Using array index before checking it
Partial string write that leads to buffer that is not zero terminated.
Mutual exclusion over || always evaluating to true
IO using format string
Check format string input/output operations.
Leaks (auto variables)
Detect when a auto variable is allocated but not deallocated or deallocated twice.
Memory leaks
Null pointer
Null pointer dereferencing
Undefined null pointer arithmetic
Other
Division with zero
Scoped object destroyed immediately after construction
Assignment in an assert statement
Dead pointers
Division by zero
Integer overflows
Invalid bit shift operands
Invalid conversions
Invalid usage of STL
Memory management
Null pointer dereferences
Out of bounds checking
Uninitialized variables
Writing const data
STL usage: Check for invalid usage of STL.
Sizeof: using 'sizeof(void)'
String: Detect misusage of C-style strings
Type: Type checks

APÉNDICE 5. INTERFÁZ GRÁFICA DE CPPCHECK



APÉNDICE 6. RESULTADO DE FLAWFINDER POR LÍNEA DE COMANDO

Usado en Debian 10.