universidad de león

Departamento de Matemáticas

MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN CIBERSEGURIDAD

Trabajo de Fin de Máster

INFERENCIA DE TIPOS DE DATO EN FICHEROS EJECUTABLES SIN SÍMBOLOS

DATA TYPE INFERENCE IN STRIPPED EXECUTABLE FILES

Autor: Pablo Ruiz Encinas

Tutor académico: Camino Fernández Llamas

Cotutor: Ricardo Julio Rodríguez Fernández

(Septiembre, 2023)

UNIVERSIDAD DE LEÓN

Departamento de Matemáticas

MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN CIBERSEGURIDAD

Trabajo de Fin de Máster

ALUMNO: Pablo Ruiz Encinas

TUTOR ACADÉMICO: Camino Fernández Llamas

COTUTOR: Ricardo Julio Rodríguez Fernández

TÍTULO: Inferencia de tipos de dato en ficheros ejecutables sin símbolos

TITLE: Data type inference in stripped executable files

CONVOCATORIA: Septiembre, 2023

RESUMEN:

La inferencia de tipos de dato en ficheros ejecutables sin símbolos es un problema desafiante en el campo del análisis de aplicaciones y la ingeniería inversa. Estos archivos son ficheros ejecutables a los que se les han eliminado los símbolos de depuración, lo que dificulta la determinación de su estructura y funciones internas. Las técnicas de inferencia de tipos de dato pretenden recuperar la información tipográfica que falta en estos binarios, útil para diversos fines, como el análisis de vulnerabilidades y la detección de código dañino. A grandes rasgos, estas técnicas pueden clasificarse en estáticas y dinámicas. Las técnicas estáticas incluyen métodos como la reconstrucción del flujo de control, el análisis del flujo de datos y la ejecución simbólica. Si bien son eficaces en ciertos casos, también están limitadas por el nivel de optimización y otros factores presentes en el binario. Por otro lado, las técnicas dinámicas implican ejecutar el binario y observar su comportamiento durante la ejecución. Estos enfoques pueden ser más exhaustivos, pero también requieren una mayor cantidad de recursos computacionales y son propensos a falsos positivos. En este trabajo, se examina la literatura existente sobre la inferencia de tipos de dato en binarios sin símbolos, y se evalúan los métodos utilizados para medir su efectividad, discutiendo las fortalezas y debilidades de los enfoques existentes. Finalmente, se extiende uno de los sistemas actuales más prometedor, actualizando su código para que funcione tanto con las versiones más recientes de los componentes que utiliza, como para soportar la ejecución exclusivamente en CPU.

Palabras clave: inferencia de tipos, aprendizaje automático, ingeniería inversa

Firma del alumno:	**VºBº Tutor:**

Agradecimientos

A ti Constanza, mi compañera de vida, mi mayor soporte, por bancarte las dificultades, los cambios de humor, la falta de tiempo y todas las taras de este loco. Porque contigo formo el mejor equipo.

A mis padres, porque desde que estoy lejos se lo que es echar en falta y valorar el tiempo juntos. Vuestra felicidad es la mía.

Y a mi abuelo, porque parecerme a ti sería el mayor halago al que aspirar. Ojalá poder reconstruir tus recuerdos de la misma forma que este trabajo reconstruye información.

Por otro lado, gracias a mi tutor Ricardo J. por atender mis mensajes a casi cualquier hora, por exprimir este trabajo al límite y, por encima de todo, por haber hecho de esta titulación algo muy disfrutable con sus asignaturas. Sin lugar a dudas es lo mejor que me llevo.

A shout-out to you too, Kexin Pei. Thanks for guiding and supporting me with my stupid questions, without you, this learning path would have taken ages. Thanks.

Abstract

Data type inference in stripped binaries is a challenging problem in the field of reverse engineering. These are executable files that don't have their debugging symbols, making difficult to determine their internal structure and functions. Data type inference techniques aim to recover the missing typographical information in these binaries, useful for a variety of purposes, such as vulnerability analysis and malware detection. These techniques can be broadly classified into static and dynamic techniques. Static techniques include methods such as control flow reconstruction, data flow analysis, and symbolic execution. While these can be effective in certain cases, they may also be limited by the level of optimization and other factors present in the binary. On the other hand, dynamic techniques involve executing the binary and observing its behavior during execution. These approaches can be more comprehensive, but they also require a greater amount of computational resources and are prone to false positives. In this thesis, the existing literature on type inference on stripped binaries is reviewed and the methods used to measure their effectiveness are also evaluated, discussing the strengths and weaknesses of existing approaches. Finally, one of the most promising current systems is extended by updating its code to work both with the latest versions of the components it uses and to support CPU-only execution.

Índice general

Agradecimientos II

Abstract III

Índice de figuras VI

Índice de tablas VII

1. Introducción 1
- 1.1. Planteamiento del problema . . . 1
- 1.2. Objetivos . . . 4
- 1.3. Alcance . . . 5
- 1.4. Estructura del trabajo . . . 5

2. Conocimientos previos 6
- 2.1. Análisis de software . . . 6
  - 2.1.1. Análisis estático . . . 7
  - 2.1.2. Análisis dinámico . . . 9
- 2.2. Ejecución simbólica . . . 9
- 2.3. Ejecución concólica . . . 10
- 2.4. Fairseq . . . 11

3. Estado del arte 13
- 3.1. Sobre la inferencia de tipos . . . 13
  - 3.1.1. Técnicas para realizar inferencia de tipos . . . 13
  - 3.1.2. Efectividad de cada enfoque . . . 14
  - 3.1.3. Características clave . . . 16
  - 3.1.4. Restricciones específicas al lenguaje de programación . . . 17
- 3.2. Sistemas de inferencia de tipos más destacados en la actualidad . . . 17
  - 3.2.1. EKLAVYA . . . 17

3.2.2. DEBIN 18
3.2.3. TYPEMINER 18
3.2.4. CATI 19
3.2.5. STATEFORMER 19

4. Análisis y diseño del sistema desarrollado **23**
4.1. Metodología 23
4.2. Mejoras realizadas sobre la implementación original 24

5. Experimentos y evaluación **27**
5.1. Entorno de evaluación y métricas 27
5.2. Análisis de resultados 28
5.2.1. Limitaciones 33

6. Conclusiones y trabajo futuro **34**
6.1. Conclusiones 34
6.2. Trabajo futuro 35
6.3. Problemas encontrados 36

Bibliografía **37**

A. Horas invertidas **43**

B. Preentrenamiento de STATEFORMER con GSM **45**

C. Estructura física de STATEFORMER **47**

Índice de figuras

1.1. Fragmento de código del programa *base64.c*. Fuente [8]. 3
1.2. Asignación de diferentes tipos. Fuente [8]. 4

2.1. CFG demonstrando las relaciones entre bloques básicos. Fuente propia. 8

3.1. Arquitectura simplificada de STATEFORMER. Fuente propia. 22

4.1. Preparación de datos para alimentar a STATEFORMER. Fuente propia. 25

5.1. *F1-score*. Fuente propia. 32

A.1. Diagrama de Gantt. Fuente propia. 44

B.1. Modelo de entradas-salida de STATEFORMER cuando se entrena con GSM. Fuente [38]. 45

Índice de tablas

5.1. Resultados con modelo entrenado 50 épocas 29
5.2. Resultados con modelo entrenado 100 épocas 30
5.3. Resultados de experimento 1 con modelo entrenado 300 épocas 31

A.1. Desglose de hitos y horas invertidas 44

Capítulo 1

Introducción

Este capítulo presenta el contexto de realización del trabajo, los objetivos que se han perseguido y el alcance del mismo. Por último, se expone la estructura del presente documento.

1.1. Planteamiento del problema

La seguridad informática es un campo en constante evolución y una de las principales preocupaciones en el mundo de la tecnología de la información. Una de las aplicaciones más instintivamente intrínsecas a la misma es el análisis del propio código que hace funcionar toda esta tecnología digital. Esta actividad puede resultar rutinaria y fácilmente realizable cuando los revisores disponen del código fuente original, condición que no siempre se ve satisfecha.

Cuando el punto de partida no es más que el propio archivo que alberga toda la funcionalidad en código máquina, surge la necesidad de realizar la misma tarea, originando lo que se conoce como *análisis de binarios* o estudio de ficheros ejecutables. El análisis de binarios tiene diversas aplicaciones, como la ingeniería inversa, el perfilado de rendimiento, la depuración o la detección de código dañino y vulnerabilidades [2, 4].

En una realidad en la que impera la rapidez, la optimización, la reducción de cargas y el intercambio hipermasivo de datos, los compiladores no se han quedado atrás. Fruto de esto se originan los binarios sin símbolos, que son aquellos programas compilados en los que se elimina todo rastro de información sobre símbolos de depuración, disposición de funciones, y tipología específica de datos con los que opera. Esto da lugar a programas de menor tamaño, lo cual es óptimo desde el punto de

vista ingenieril. No obstante, una consecuencia directa de este proceso es la mayor dificultad para analizar y comprender el comportamiento de los binarios, aumentando así la incertidumbre y el abanico de casuísticas desde el punto de vista de la seguridad de la información.

Con el exponencial aumento de los ciberataques en la última década, resulta esencial disponer de herramientas, procedimientos y estrategias oportunas para comprender el funcionamiento de un programa a partir de su archivo ejecutable. Conocer los tipos de datos que maneja y con los que opera en todo momento es una característica fundamental que facilita este objetivo [3].

La *inferencia de tipos* se refiere a la deducción automática de los tipos de datos que maneja un programa a partir de su comportamiento [4]. Como ya se ha expuesto, la inferencia de tipos en el contexto de binarios sin símbolos puede ser utilizada para recuperar información sobre el código eliminado y mejorar la comprensión del comportamiento del binario, pero también es parte fundamental de otros procesos como el análisis de código dañino, la introspección de memoria o la modificación y comparación de código binario [8].

Sin embargo, hasta ahora ha faltado una investigación sistemática en este campo específico. La Figura 1.1 procede del programa de codificación y descodificación *base64.c* de la biblioteca de C. El programa utiliza una variable decode de tipo `booleano` para registrar las opciones del usuario. Sin embargo, tras la compilación, la variable decode se representa simplemente como un byte en la pila (almacenada en `[ebp-1]`) y se pierde el tipo `booleano` original. Debido al análisis excesivamente conservador que adoptan algunas de las herramientas hasta ahora existentes al aplicar inferencia de tipos, se infiere que la variable `[ebp-1]` es de tipo `char` o `byte_t`, lo que es incorrecto.

```c
int main()
{
    bool decode = false;
    int opt = getopt;
    switch(opt){
        case 'd':
            decode = true;
            break;
        default:
            break;
    }
    if(decode)
        do_decode;
}
```

(a) Código C

```
  mov    byte ptr[ebp-1], 0
  cmp    dword ptr[ebp-8], 64h
  jz     short loc_40101B
  jmp    short loc_40101F
loc_40101B:
  mov    byte ptr[ebp-1], 1
loc_40101F
  movzx  eax, byte ptr[ebp-1]
  test   eax, eax
  jz     short loc_401035
  call   do_decode
loc_401035
  ret
```

(b) Código ensamblador

Figura 1.1: Fragmento de código del programa *base64.c*. Fuente [8].

La inferencia de tipos en binarios sin símbolos presenta, por tanto, diversos desafíos:

1. En primer lugar, el proceso de eliminar ciertos componentes del binario ocasiona una falta de información útil para la inferencia de tipos. Por ejemplo, resulta complejo determinar los tipos de datos de las variables sin la información de declaración de tipos disponible en el código fuente.
2. La propia complejidad del código ensamblador puede dificultar el análisis de las relaciones entre las variables y sus tipos de datos.
3. La dificultad que presenta el manejo de tipos complejos, como estructuras de datos o clases.
4. La presencia de ofuscación o cifrado en el código.
5. Como añadido, cuando en el proceso de inferencia se utilizan técnicas de aprendizaje automático, es necesario disponer de un gran conjunto de ejemplos de código con tipos de datos conocidos y distintos para entrenar un modelo de aprendizaje automático eficaz, lo que resulta un desafío si no se dispone de suficientes ejemplos o de suficiente poder computacional.

Ejemplificando una de estas dificultades se encuentra la Figura 1.2, donde se consideran tres asignaciones simples para tres variables con tipos diferentes (por orden de aparición, puntero a entero, a real y a real con doble precisión). Algunas aplicaciones (como `SmartDec` [10]) recuperan el mismo tipo `int32_t` para estas tres

variables, mientras que otras (como Hex-Rays [14]) infieren el tipo `Dword*` para `i` y `f` y el tipo `Qword*` para `d`. De nuevo, algunos de estos resultados son incorrectos.

```
func1 (int* i){
    *i = 10;
}
func2 (float* f){
    *f = 10.0;
}
func3 (double* d){
    *d = 10.0;
}
```

(a) Pseudo código C

```
mov     [i], 0Ah
movss   xmm0, ds:XX
movss   [f], xmm0
movsd   xmm0, ds:XX
movsd   [d], xmm0
```

(b) Código ensamblador

Figura 1.2: Asignación de diferentes tipos. Fuente [8].

A pesar de estos desafíos, la inferencia de tipos en binarios sin símbolos es un campo emergente de investigación, con el potencial de mejorar significativamente la seguridad de la información.

1.2. Objetivos

El objetivo general de este trabajo es profundizar en el actual estado del arte de inferencia de tipos en análisis de código binario para culminar con una implementación a nivel de software que mejore los sistemas existentes.

Con objeto de satisfacer este objetivo general, se proponen los siguientes objetivos específicos:

1. Identificación de las técnicas y enfoques utilizados en materia de inferencia de tipos.
2. Análisis de los principales mecanismos existentes.
3. Comparativa entre los mismos y evaluación de su efectividad, reconociendo las fortalezas y debilidades de cada uno de ellos.
4. Ampliación del sistema más destacado para la realización de un análisis más exhaustivo.
5. Simplificación de conceptos y técnicas relacionados con el aprendizaje automático mediante análisis y documentación meticulosa.

1.3. Alcance

Existen multitud de diferentes arquitecturas de CPU en las que se puede compilar un programa [7]. De igual manera, un fichero binario o ejecutable se puede crear con mayor o menor número de medidas para dificultar su análisis, reforzar su seguridad o, simplemente hacerlo más eficiente (véase la ofuscación de código, la eliminación de símbolos de depuración o el empaquetado, entre otras técnicas [27]).

A esta ecuación se ha de añadir la ingente cantidad de herramientas que abordan el problema del análisis de código binario, algunas de código abierto como Ghidra [11] o Angr [12], y otras propias como lo son Binary Ninja [13] o IDA Pro [14].

Con esta realidad, se ha decidido mantener el alcance de este proyecto de investigación limitado a técnicas de inferencia de tipos de dato en binarios exclusivamente sin símbolos, utilizando únicamente software libre como elección para realizar la implementación.

1.4. Estructura del trabajo

Por último, se presenta la distribución restante del presente documento: en el Capítulo 2 se redactan los conocimientos previos necesarios para comprender correctamente el trabajo elaborado. Posteriormente, el Capítulo 3 resume el estado actual de la investigación en lo relativo a la inferencia de tipos en ficheros ejecutables. A continuación, en el Capítulo 4 se expone y desarrolla el sistema elegido y la implementación realizada. Una vez compartida la solución desarrollada, se describen las pruebas realizadas en el Capítulo 5. Finalmente, el Capítulo 6 presenta las conclusiones y el trabajo futuro extraídos de la investigación.

Adicionalmente y para resolver algunas cuestiones planteadas a lo largo del trabajo, se han añadido al final los siguientes anexos: el Anexo A muestra las horas invertidas en el desarrollo, el Anexo B muestra el funcionamiento del módulo de preentrenamiento del sistema elegido, y el Anexo C detalla qué archivos son relevantes en caso de querer trabajar y/o modificar el sistema elegido.

Capítulo 2

Conocimientos previos

En este capítulo se exponen los conocimientos requeridos de especial relevancia para una adecuada comprensión del trabajo realizado. En primer lugar, se realiza una descripción general sobre qué es el análisis de software para después concretar en los diferentes tipos de análisis que se practican en la actualidad. A continuación, se explica el concepto de ejecución simbólica de manera más detallada, incluyendo en qué consiste la ejecución concólica. Estos tipos de análisis se usan en el sistema de inferencia de tipos desarrollado en este trabajo. Para concluir, se presenta Fairseq, el principal entorno utilizado para trabajar con modelos de aprendizaje automático.

2.1. Análisis de software

Tradicionalmente, se entiende por análisis de software la validación de un sistema contra los requisitos del mismo para verificar que el sistema se comporte de la forma esperada. Durante la fase de pruebas, generalmente se realiza tanto una verificación funcional (esto es, *qué hace*) como una lógica (esto es, *cómo lo hace*) del sistema. El objetivo de este proceso no es más que el identificar las propiedades, comportamiento e interacciones entre los distintos componentes del software para asegurar su calidad, robustez y seguridad.

Cuando se habla de análisis de software en el contexto de la ciberseguridad, el principal objetivo de confirmación es que no existen vulnerabilidades tanto en el propio diseño del sistema como en los componentes que utiliza.

Cuando este tipo de comprobación se realiza sin conocer el detalle sobre el funcionamiento interno del software (como por ejemplo, su estructura de código, todos los posibles estados en los que puede derivar, o qué casos de uso contempla), la ac-

tividad se categoriza como prueba de caja negra. Por el contrario, cuando toda esta información está disponible desde el comienzo y hay una completa transparencia de código, se denomina prueba de caja blanca.

Aunque cabría suponer que de cara a la simulación de ciberataques, un adversario no dispone (presumiblemente) de acceso al código fuente del sistema atacado y, por lo tanto, realizar pruebas de caja negra es la decisión más lógica, esto no es así en la práctica. La principal limitación de llevar a cabo exclusivamente pruebas de caja negra es que se realizan directamente contra los requisitos del sistema, sin explorar todos los estados que pueda alcanzar la ejecución ni el comportamiento ante eventos imprevistos. De este modo, es posible que cuando el sistema esté desplegado y en producción se produzcan casuísticas no detectadas durante la fase de verificación por no realizar pruebas más exhaustivas. Dado que todas las pruebas anteriormente mencionadas requieren que el código se ejecute, este paradigma se encuadra dentro de lo que se conoce como *análisis dinámico*.

Por contraparte, se cataloga como *análisis estático* cuando el código analizado no es ejecutado. Esta modalidad es considerada de mayor complejidad técnica, ya que no disponer de interacción directa con la lógica detrás del sistema dificulta la interpretación de resultados y la obtención de conclusiones de valor.

A continuación, se procede a desarrollar en mayor detalle ambas metodologías de análisis.

2.1.1. Análisis estático

El análisis estático se realiza sin la necesidad de ejecutar el código, mediante lectura del mismo o a partir del desensamblado del fichero ejecutable que se genera tras compilar. En el análisis estático es frecuente la utilización de árboles para representar el flujo de ejecución del programa (también denominados CFG, del inglés: *Control Flow Graph*) [20]. Las instrucciones se agrupan en bloques básicos, y cada bloque básico representa un nodo en el grafo. Un bloque básico es una secuencia lineal de instrucciones de programa que tiene un único punto de entrada (la primera instrucción ejecutada) y un único punto de salida (la última instrucción ejecutada) [20].

El bloque básico raíz representa el inicio de ejecución del programa, y cada vez que se ejecuta una instrucción de cambio de flujo en el programa (por ejemplo, `if/else`, `while` o `for`), se crea un arco en el grafo para representar un nuevo camino de ejecución. La Figura 2.1 plasma estas relaciones:

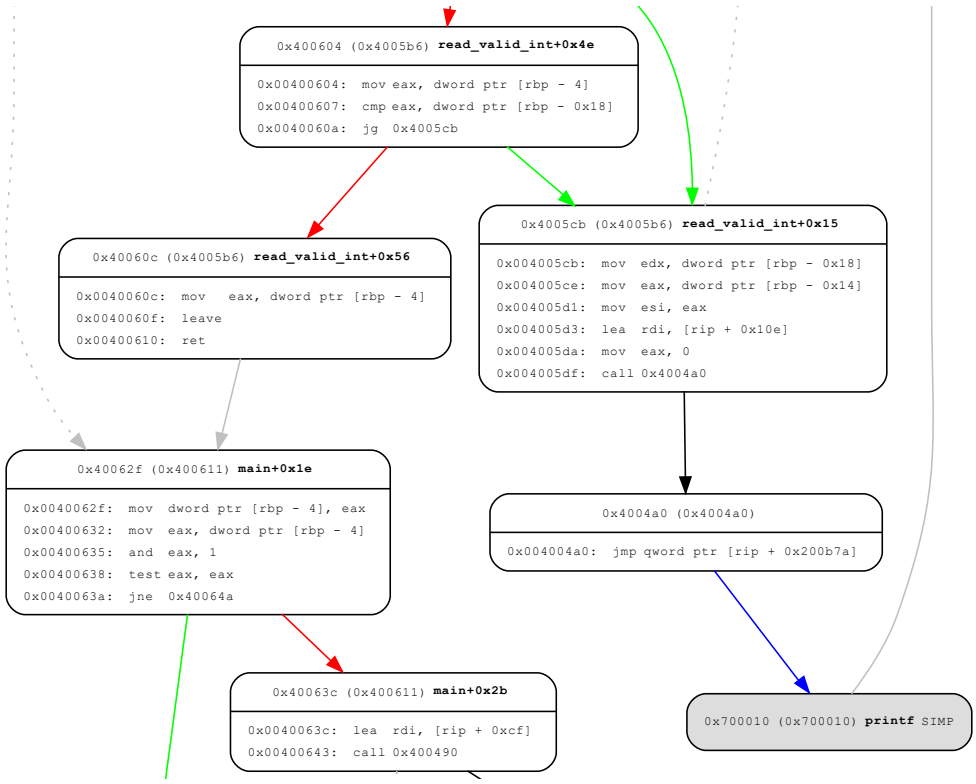


Figura 2.1: CFG demonstrando las relaciones entre bloques básicos. Fuente propia.

El análisis estático cubre todo el código del programa, mientras que el análisis dinámico cubre únicamente las fracciones del código que son ejecutadas. Alcanzar una cobertura total sirve de gran utilidad cuando se realiza un análisis estático de forma previa a la fase de análisis dinámico, ya que los resultados obtenidos se pueden utilizar para realizar un análisis dinámico más optimizado (por ejemplo, reduciendo el número de casos de prueba).

Si se dispone del código fuente, es posible obtener a nivel de compilación información como errores sintácticos, variables no inicializadas o código no alcanzable, entre otros. Cuando el único objeto de análisis disponible es un binario, se requieren herramientas automáticas con capacidades de desensamblado y (preferiblemente) de interpretación y resolución de símbolos de depuración y/o acceso a memoria.

El efectivo uso de estas herramientas no es trivial, y delegar un análisis en configuraciones y resultados obtenidos por defecto no es una opción eficaz, pues el elevado número de falsos positivos y la cantidad de diferentes protecciones a nivel de software que pueden encontrarse en la actualidad dificultan aún más la obtención e interpretación de resultados. Así pues, disponer de herramientas y mecanismos que mejoren la presentación e información presentada en los análisis, optimizar las

soluciones automáticas y, además, tener la posibilidad de configurarlas y extender sus capacidades son características que se han tornado clave en este ámbito.

2.1.2. Análisis dinámico

El análisis dinámico se realiza ejecutando la aplicación. Como consecuencia, este tipo de análisis se efectúa únicamente sobre aquellos caminos de ejecución que son efectivamente explorados, lo que implica una menor cobertura del código del programa.

Este tipo de análisis resulta especialmente útil a la hora de comprender el comportamiento en tiempo de ejecución del programa e identificar problemas que pueden no ser evidentes durante el análisis estático, como fugas de memoria o desbordamientos de búfer, entre otros.

Una de las principales ventajas del análisis dinámico es su capacidad para manejar características en tiempo de ejecución, tales como la vinculación dinámica, el polimorfismo y los hilos de ejecución de forma más precisa que el análisis estático [21]. Igualmente, esta aproximación presenta dificultades, siendo una de las principales la generación de casos de prueba. Encontrar casos de prueba que ejecuten todos los caminos lógicos de ejecución es una tarea difícil debido al problema de la explosión de estados.

Sin embargo, cabe señalar que el análisis dinámico y el análisis estático no son mutuamente excluyentes; sirven como técnicas complementarias, y cada una de ellas proporciona una visión única del sistema de software que se está analizando. Por ejemplo, para la generación de casos de prueba el análisis dinámico puede aprovecharse de un análisis estático previo, de igual modo que el análisis estático puede ayudar a una mejor planificación del análisis dinámico.

2.2. Ejecución simbólica

La ejecución simbólica es un tipo de análisis estático diseñado para el testeo de aplicaciones y la búsqueda de *bugs*. La diferencia de este tipo de análisis con respecto a otros es que se basa en la abstracción de los datos de entrada en lugar de asignarles valores concretos [22].

La abstracción consiste en utilizar valores simbólicos y un intérprete para seguir el flujo de control del programa. Los estados simbólicos se mantienen durante toda la ejecución, de modo que si una entrada sufre alguna modificación se seguirá repre-

sentando de manera simbólica (por ejemplo, $\alpha + 1$ tras un incremento de la variable simbólica α).

Con la abstracción se consigue evaluar cada camino con expresiones lógicas, que ayudan a analizar posteriormente qué valores concretos se necesitan para cubrir cada camino del árbol mediante motores SMT (del inglés: *Satisfiability Module Theory*) [24]. A diferencia de las pruebas unitarias, que solo cubren un posible camino de ejecución en cada test, la ejecución simbólica es particularmente útil para explorar sistemáticamente múltiples caminos del programa y generar valores de entrada concretos para cada uno de ellos. Cuando se encuentra una decisión en un camino de ejecución, se añade una nueva restricción a la expresión lógica que representa las condiciones necesarias para alcanzar dicho camino. Esta expresión ampliada es evaluada para comprobar si se puede satisfacer o no (es decir, si existe una posible asignación de valores a sus variables que haga cierta o falsa dicha expresión). En caso de que se pueda satisfacer, se continúa con el análisis de ese camino. En caso contrario, se descarta su análisis, puesto que no es posible una asignación de valores que permita alcanzar dicho camino.

Esta técnica de análisis es particularmente útil a la hora de identificar varios tipos de errores, tales como violaciones de aserción, excepciones no capturadas y vulnerabilidades de seguridad. No obstante, una de sus principales limitaciones es el problema conocido como *explosión de estados*. La explosión de estados es un fenómeno que afecta principalmente a la escalabilidad de la ejecución del análisis, ya que implica que los caminos que se van generando a medida que fluye la ejecución crecen exponencialmente, terminando por consumir los recursos computacionales del sistema analizador.

2.3. Ejecución concólica

La ejecución concólica es un subtipo de la ejecución simbólica que representa una técnica híbrida de análisis de programas, ya que combina la ejecución concreta y simbólica [23]. En la ejecución concólica, cada instrucción del programa se ejecuta simultáneamente tanto de forma concreta como simbólica, lo que permite la recopilación de restricciones de ruta y datos de prueba para los test unitarios a la vez.

La principal motivación de la ejecución concólica es abordar las limitaciones de la ejecución simbólica, como el problema de la explosión de estados y los retos asociados a la resolución de restricciones en programas complejos. La ejecución concólica

introduce dos optimizaciones para hacer más factible el análisis de programas grandes:

1. Nunca llamar al solucionador de restricciones durante la ejecución de un camino, evitando así atascarse cuando se encuentran restricciones complejas.
2. Realizar bifurcaciones especulativas siempre que la ejecución alcanza una rama que depende de una entrada simbólica, evitando así la necesidad de llamar al solucionador de restricciones.

La ejecución concólica invoca al solucionador de restricciones cuando selecciona un nuevo estado para ejecutar. Dado que todos los estados son el resultado de bifurcaciones especulativas, es necesario comprobar si los estados son realmente viables antes de ejecutarlos. Para ello, el solucionador de restricciones calcula las entradas concretas que permitirían alcanzar ese estado. Si el solucionador no encuentra entradas posibles, el estado se descarta. En caso contrario, las entradas concretas calculadas se utilizan para reanudar la ejecución.

2.4. Fairseq

Fairseq [59] es un conjunto de herramientas de aprendizaje profundo de código abierto desarrollado por Facebook AI Research (FAIR) [34]. Está ideado para proporcionar un marco de utilidades flexible y modular para que investigadores e ingenieros experimenten e implementen modelos de personalizados para acometer problemas relativos a procesamiento del lenguaje natural (o NLP, del inglés: *Natural Language Processing*).

Fairseq permite a los usuarios mezclar y combinar fácilmente los componentes para crear modelos personalizados. Esto resulta especialmente útil para experimentar con diferentes arquitecturas de modelos, técnicas de entrenamiento y métodos de preprocesamiento de datos.

El problema fundamental en el que se focaliza Fairseq es el conocido como $seq2seq$ (del inglés: *sequence to sequence* o modelado de secuencia a secuencia). Esta tarea consiste en transformar una secuencia de datos de entrada en una secuencia de datos de salida. En la actualidad, se utiliza especialmente para trabajar con datos secuenciales, como en la traducción de idiomas, el resumen de textos o el reconocimiento de voz.

Los dos principales componentes que conforman estos sistemas son:

- **Codificador:** Elemento que toma la secuencia de entrada y la procesa en una representación de tamaño fijo, a menudo denominada *vector de contexto* o *vector de pensamiento*. Esta representación encapsula la información esencial de la secuencia de entrada y sirve como punto de partida para generar la secuencia de salida.
- **Decodificador:** Elemento que toma el vector de contexto del codificador y genera la secuencia de salida basándose en esta representación. Produce un elemento de la secuencia de salida cada vez, condicionando cada predicción a los elementos generados anteriormente.

Gracias a estos dos componentes, las soluciones $seq2seq$ se presentan particularmente sólidas para abordar problemas de procesamiento del lenguaje, ya que permiten el manejo de entradas y salidas de longitudes variables.

Como último argumento a favor de Fairseq, cabe señalar que por debajo hace uso de la biblioteca PyTorch [56], también publicada por el grupo FAIR y que sirve como interfaz para la misma.

Capítulo 3

Estado del arte

En este capítulo se reflejan los principales resultados obtenidos a partir de la revisión sistemática de la literatura realizada en torno al problema de la inferencia de tipos de dato. En primer lugar, se recuperan los conceptos de análisis estático y dinámico para presentar los principales grupos en los que se agrupan las distintas técnicas de inferencia de tipos, exponiendo sus características clave, bondades y limitaciones. Finalmente se cierra el capítulo presentando y desarrollando en mayor profundidad los principales sistemas actuales en materia de inferencia de tipos, tanto por aceptación de la comunidad experta como por lo prometedor de sus resultados.

3.1. Sobre la inferencia de tipos

3.1.1. Técnicas para realizar inferencia de tipos

En la actualidad, se han desarrollado multitud de técnicas capaces de solucionar, o al menos aproximar, en mayor o menor medida el problema de la inferencia de tipos. Por esto, se ha decidido incluir en este estudio únicamente aquellos sistemas propuestos desde el año 2015 en adelante.

De este modo, se distinguen tres principales vertientes en las que se agrupan las distintas soluciones que abordan la inferencia de tipos:

- **Técnicas de análisis estático:** no requieren de la ejecución del binario. Estudios como [26–29] trabajan a partir del CFG y métodos de ejecución simbólica para discernir cuál es el tipo de un dato a partir de su representación abstracta [9].

- **Técnicas de análisis dinámico:** estudian el binario en tiempo de ejecución. Herramientas como la propuesta en [30] hacen uso de motores de instrumentación dinámica (como PIN [32] o Valgrind [33]), y otras como [35] identifican y hacen uso de estructuras de datos complejas presentes en la memoria.
- **Técnicas mixtas:** combinan ambas aproximaciones [28, 37, 43]. Por ejemplo, en [36] se hace uso de fuentes de código abierto para obtener y analizar distintos paquetes y binarios para, posteriormente, verificar las inferencias mediante microejecuciones de ciertas funciones específicas.

Con el avance en el campo del aprendizaje automático, en los últimos años ha aparecido un nuevo método para desmarcarse de los anteriormente expuestos. En [38, 43, 44] se ha introducido el concepto de *data-driven* o *semantic-driven* en los campos de la seguridad informática y el análisis de software. Si bien en muchos de los casos parte de la comunidad los engloba acertadamente dentro del ámbito del análisis estático o incluso mixto, las técnicas que incorporan el aprendizaje automático y el aprendizaje profundo comienzan a destacar, pudiendo considerarse de forma igualmente válida como un método aparte en lugar de un subconjunto dentro de las anteriores. Estas técnicas deciden despreciar el uso de conocimiento experto y/o reglas previamente definidas como principal pilar sobre el que basar sus capacidades de inferencia para dar lugar a los algoritmos de aprendizaje. Estos algoritmos se aplican sobre la semántica operacional y el control de flujo de bloques de código, sirviendo después como conjunto de datos de entrenamiento para realizar las inferencias. La mayoría de estas investigaciones se sirven de redes de neuronas y problemas de clasificación multiclase.

Cabe enfatizar que el éxito depende en gran medida de la calidad de las heurísticas/datos de entrenamiento y de la cantidad de información disponible para el análisis. Sin la suficiente información, las heurísticas engloban un alcance pobre, mientras que las redes neuronales no refinan de manera óptima sus clasificadores, resultando en ambos casos en una baja tasa de acierto.

3.1.2. Efectividad de cada enfoque

Resulta complejo aunar en un formato genérico las distintas métricas y resultados obtenidos por las investigaciones recopiladas para este trabajo, pues las baterías de prueba varían tanto en número como en tipología según cada caso (véanse [45, 47–49] como ejemplos de disonancia en cuanto a estándar de pruebas). Aquí también juega un papel relevante la arquitectura para la cual está construido el binario

analizado, pues añade una capa extra de complejidad a la hora de establecer métricas de efectividad genéricas, ya que heurísticas o patrones compuestos para arquitecturas CISC (como x86, x86_64) pueden resultar equívocas para programas elaborados para arquitecturas RISC (como ARM, PowerPC, MIPS).

Aún así, es posible extraer algunas conclusiones y datos de valor prestando atención a las estadísticas que aporta cada trabajo estudiado. De este modo, se establece que:

- Las técnicas de análisis estático permiten realizar la inferencia de una manera rápida y relativamente precisa [4, 22, 28]. Además, el análisis estático generalmente tiene una mayor cobertura [9]. Sin embargo, también tiene algunos puntos débiles. Por ejemplo, el análisis estático asigna los tipos de datos de forma demasiado genérica, restándole precisión [2]. Además, el análisis estático no es el más confiable si el código está diseñado de manera intencionalmente confusa o si utiliza técnicas de ofuscación para ocultar su funcionamiento [25].
- Las técnicas de análisis dinámico, por otro lado, al actuar sobre el binario en ejecución, pueden proporcionar información más detallada y precisa sobre el comportamiento del programa [30, 35]. A cambio, la cobertura que ofrecen no es tan amplia y, generalmente, son más lentas que las técnicas de análisis estático, pues se ven influidas por las características del propio binario y del entorno en el que se ejecuta [21].
- Las técnicas mixtas se presentan como una solución fiable y robusta en cuanto a tasa de acierto, velocidad y cobertura, pues aprovechan las ventajas de las técnicas anteriores. En concreto, la combinación entre heurísticas y microejecuciones de bloques de código se muestra como una aproximación que proporciona buenas garantías [36, 37].
- Con respecto a las soluciones que integran aprendizaje automático, cabe señalar la diferencia de desempeño que reflejan los resultados en función del tamaño del binario a analizar. Para ejecutables reducidos, no solo se observa un empeoramiento con respecto a las anteriores en cuanto a tiempos se refiere [39, 43], sino que además acusan una relativamente significativa disminución en la tasa de acierto [44]. Por el contrario, la realidad que reflejan para binarios de gran tamaño es radicalmente opuesta, pues ofrecen un mejor rendimiento en menos tiempo de ejecución [8]. No obstante, en ambas situaciones la calidad y cantidad de los datos utilizados en el conjunto de entrenamiento resulta determinante y decisivo [45].

En todos los casos, las tasas de acierto que reflejan los respectivos estudios se muestran favorables y con porcentajes de acierto generalmente altos, si bien es cierto que los escenarios de prueba difieren considerablemente entre unos y otros, lo cual resta contundencia a los resultados.

3.1.3. Características clave

Aunque varía la forma en la que las distintas aproximaciones estudiadas buscan llevar a cabo la inferencia de tipos, todas ellas se apoyan en estrategias similares que se pueden agrupar de la siguiente forma:

- **Análisis de patrones:** consiste en buscar patrones a partir del código desensamblado del binario que puedan indicar el tipo de dato que se está tratando. Por ejemplo, si se encuentra un patrón de bytes repetido a lo largo del archivo y que se asemeja a un número entero, se puede inferir que se trata de un dato de tipo entero.
- **Análisis de funciones:** consiste en buscar funciones conocidas en el binario sin símbolos y utilizarlas para inferir el tipo de datos que se está tratando. Por ejemplo, si se encuentra una función que se utiliza para leer un archivo, se puede inferir que la variable usada se trata de una variable de tipo `byte_t` (para el caso de C/C++).
- **Análisis de cadenas:** consiste en buscar cadenas de texto en el binario sin símbolos y utilizarlas para inferir el tipo de datos que se está tratando. Por ejemplo, si en la variable se encuentra una cadena que se asemeja a una dirección de correo electrónico, se puede inferir que se trata de un dato de tipo `char*` o `string` (para el caso de C o C++).
- **Análisis de los registros:** consiste en aprovechar el contexto de las instrucciones de acceso a memoria para inferir los tipos de variables. Se define una función denominada *Contexto de Uso de Variable* (VUC, del inglés: *Variable Use Context*), que incluye la instrucción de destino y las instrucciones anteriores y posteriores a ella. Mediante el análisis de los VUC y el uso de mecanismos de votación, se consigue inferir los tipos de variables sin depender de reglas o heurísticas.

3.1.4. Restricciones específicas al lenguaje de programación

En cuanto a características propias de cada lenguaje de programación, no se ha encontrado suficiente literatura ni suficientes experimentos empíricos que prueben o remarquen la presencia de restricciones específicas en función a los mismos.

La mayoría de los trabajos se centran en ficheros ejecutables compilados a partir de código C/C++ [10,48], con algunos casos de Java [5]. No obstante, sí existe un consenso en el que la comunidad concuerda respecto a que los lenguajes dinámicamente tipados (como Python o JavaScript) presentan una mayor dificultad a la hora de discernir los tipos de datos que están siendo procesados, ya que estos lenguajes resuelven los tipos en tiempo de ejecución [53].

3.2. Sistemas de inferencia de tipos más destacados en la actualidad

Tras realizar un proceso de revisión sistemática de la literatura actual en materia de inferencia de tipos de dato, los ejemplos que se detallan a continuación se han considerado especialmente relevantes por los resultados obtenidos.

3.2.1. EKLAVYA

El sistema EKLAVYA fue presentado en 2017 [39]. Este sistema continúa siendo usado como referencia en las evaluaciones de los trabajos presentados en las últimas publicaciones [8,38,50,52]. El sistema aprovecha técnicas de aprendizaje automático mediante redes neuronales para atajar el problema de la inferencia de tipos. A alto nivel, este sistema se compone de dos módulos:

- **Modulo de inserción:** Encargado de realizar el aprendizaje de las semánticas de cada instrucción.
- **Módulo de predicción:** Encargado de recuperar la información de los argumentos de una función, para después asignar su tipo.

A grandes rasgos, el funcionamiento de EKLAVYA ejecuta los siguientes pasos:

1. **Extracción de la información semántica de cada instrucción:** A partir de un binario compilado con información de depuración, se infieren las semánticas tomando su uso contextual en el mismo.

2. **Recuperación de argumentos de función:** Utilizando una red neuronal por cada argumento de una función, se recupera su tipo.

Mediante este proceso, EKLAVYA logra una tasa de acierto aproximada del 81 %. Si bien es cierto es un resultado notable, es necesario resaltar que EKLAVYA tiene dos principales limitaciones: por un lado recupera únicamente parámetros de funciones, con lo que variables locales y globales quedarían fuera del alcance, y por el otro, generaliza los tipos de datos inferidos a los siguientes: `int`, `char`, `float`, `void*`, `enum`, `union` y `struct`.

3.2.2. DEBIN

DEBIN es un sistema presentado en 2018 en [52] diseñado no solo para predecir información de depuración en binarios sin símbolos, sino también para recuperar nombres de variables y funciones. Utiliza el aprendizaje automático para entrenar modelos probabilísticos en miles de binarios con símbolos, y luego aplica estos modelos para predecir la información de depuración en binarios sin símbolos. Estos modelos probabilísticos se dividen principalmente en dos:

- **ERT:** del inglés: *Extremely Randomized Trees* [41], es un sistema de clasificación para extraer variables del programa.
- **CRF:** del inglés: *Conditional Random Field*, es un sistema probabilístico lineal para predecir las propiedades de las variables y otros elementos del programa obtenidos a partir del ERT.

La principal fortaleza de DEBIN reside en su capacidad de aprovechar grandes cantidades de datos de entrenamiento para mejorar la precisión de la predicción, ya que en total es capaz de predecir 17 tipos de dato. Otro uso de DEBIN es la inspección de código dañino, ya que al recuperar nombres de variables y funciones, permite buscar y comparar estos nombres obtenidos con los que típicamente manejan las muestras de código dañino. De igual modo, cabe destacar que funciona en binarios tanto para arquitecturas x86 como x86_64 y ARM.

3.2.3. TYPEMINER

Otro sistema que se ha considerado relevante de estudio para la realización de este proyecto,es TypeMiner [40], presentado en 2019. Se define como un método estático para recuperar los tipos de dato de las variables en ficheros ejecutables. TypeMiner fue precursor en considerar que los tipos de dato dejan trazas características de cada

uno al ser compilados, estimando que el código ensamblador puede ser una fuente de conocimiento para aprender a detectar estas trazas y recuperar así los tipos de los datos que maneja.

A rasgos generales, TypeMiner localiza objetos de datos (como variables locales o parámetros de funciones) en el código compilado y rastrea el flujo de datos a nivel de instrucción. Las trazas resultantes reflejan cómo se procesa este objeto, caracterizando el tipo de datos almacenados. Estas trazas se integran posteriormente en un espacio vectorial, de modo que las trazas similares están próximas entre sí y pueden asignarse a tipos etiquetados mediante un clasificador.

Cabe resaltar que la compatibilidad de TypeMiner es independiente de la arquitectura de la CPU, por lo tanto puede utilizarse sobre distintas arquitecturas una vez que se ha entrenado un modelo correspondiente para la misma.

3.2.4. CATI

CATI [50] (del inglés: *Context-Assisted Type Inference*), presentado en 2020, aprovecha la información contextual para mejorar la precisión de la inferencia de tipos. Es capaz de inferir 19 tipos de variables diferentes. Los experimentos han demostrado que alcanza una tasa de precisión del 71,2 % en binarios nunca antes vistos. En comparación con los métodos más avanzados, CATI ha demostrado un rendimiento superior tanto en precisión como en eficacia.

CATI es capaz de clasificar un total de 19 tipos de dato distintos, entre los que se distingue: `struct *`, `arithmetic *`, `void *`, `char`, `unsigned char`, `float`, `double`, `long double`, `long long int`, `unsigned int`, `unsigned long int`, `unsigned long long int`, `unsigned short int`, `short int`, `long int`, `enum`, `int`, `bool` y `struct`.

3.2.5. STATEFORMER

Tras la revisión sistemática de la literatura existente en el campo de la inferencia de tipos de dato realizada, ha resultado notoriamente llamativo el modelo de STATEFORMER [38], un sistema que aborda el problema de la inferencia de tipos de dato desde el aprendizaje automático, modelándolo como un problema de procesamiento del lenguaje natural.

Este sistema es especialmente relevante como objeto de estudio, pues la complejidad de su solución, unida con las prometedoras aportaciones que presenta, conforman el producto final sobre el que se fundamenta esta investigación.

En primer lugar, el trabajo presentado en [38] realiza dos grandes contribuciones:

- **STATEFORMER:** una arquitectura neuronal que aprende explícitamente la semántica operativa del código ensamblador para mejorar las capacidades en cuanto a inferencia de tipos de dato.
- **GSM** (del inglés: *Generative State Modeling*):, un novedoso sistema de preentrenamiento que enseña la semántica operativa del flujo de datos y de control en bloques de código a STATEFORMER para, a continuación, optimizar sus parámetros de cara a la realización de la inferencia de tipos de dato.

.

En la definición de las contribuciones de STATEFORMER es posible apreciar el paralelismo establecido entre la inferencia de tipos de dato y el procesamiento del lenguaje natural, ya que pretende hacer uso de la semántica operativa del código ensamblador para aprender de qué tipos de dato está haciendo uso de la misma manera que un sistema de procesamiento del lenguaje tradicional hace uso de la estructura de un texto para aprender a generar salidas que tengan sentido con el propio texto original. Esta casuística encaja con la que se pretende solucionar con los sistemas $seq2seq$ (véase la Sección 2.4).

Cabe reseñar que para esta investigación, si bien se ha estudiado y analizado el sistema de GSM para alcanzar un entendimiento completo de toda la arquitectura conjunta, este sistema se ha considerado como una caja negra durante la implementación y experimentación. La principal razón de esta decisión no es otra que la limitación de recursos técnicos presente durante el proyecto. Como se expone en el Anexo B, la tarea de preentrenamiento con GSM requiere de una capacidad de computación no disponible durante la realización de este trabajo.

A continuación, se descompone el algoritmo de funcionamiento de STATEFORMER, identificando los siguientes pasos:

1. **Recopilación de los estados del programa:** Para entrenar STATEFORMER con GSM, primero se obtienen las trazas de ejecución de los binarios. Esto se logra mediante microejecuciones que rastrean partes arbitrarias del ejecutable sin tener que encontrar entradas concretas del programa que maximicen la cobertura. Al no ejecutar el programa desde su punto de entrada, el motor de ejecución necesita inicializar estados intermedios del programa (registros y contenido de memoria) con valores aleatorios. Estos estados intermedios los denomina $\mu State$.

2. **Recopilación de $\mu State$:** Se recopilan las trazas de $\mu State$ y se enmascara un subconjunto aleatorio de ellas para entrenar al modelo reconstruyendo la traza completa de $\mu State$. El $\mu State$ consta de dos fuentes de información: (1) Los valores concretos de todos los registros, direcciones de memoria y desplazamientos codificados que aparecen en la instrucción (denominados $\mu DataState$), y (2) la anotación booleana que indica si cada instrucción del código se ejecuta en un determinado $\mu State$ (denominado $\mu ControlState$). La primera aparece tanto en la entrada como en la salida de STATEFORMER (es decir, subconjunto de $\mu DataState$ como entrada y conjunto completo de $\mu DataState$ como salida). Este último solo aparece en la salida.

3. **Codificación de código ensamblador:** La secuencia de código ensamblador se construye tokenizando las instrucciones de los binarios desensamblados, a distinguir (1) secuencia de código ensamblador estático, (2) secuencia $\mu DataState$, (3) secuencia de posición de instrucción, (4) secuencia de posición de *opcode* (o código de instrucción)/operando y (5) secuencia de arquitectura. Además de tratar los opcodes y los operandos como símbolos, mantiene los signos de puntuación, ya que proporcionan pistas contextuales cruciales (por ejemplo, los corchetes indican una desreferencia de una dirección de memoria). Los valores numéricos concretos se insertan en la secuencia de $\mu DataState$ y se sustituyen por un token especial.

4. **Codificación de $\mu DataState$:** Se normaliza la secuencia de $\mu DataState$ como una matriz bidimensional. Así, cada $\mu DataState$ puede verse como una secuencia de tokens en la que se transforman todos los valores numéricos en una representación hexadecimal de 8 bytes.

5. **Codificación de información espacial y sintáctica:** Como se normalizan y concatenan todas las instrucciones en una secuencia de tokens, los límites de las instrucciones y la ubicación relativa de los tokens dentro de cada instrucción se vuelven ambiguos. Para abordar esto, se introducen dos codificaciones posicionales: (1) la codificación posicional de las instrucciones y (2) la codificación posicional de los opcode/operandos. Las secuencias resultantes se anotan en cada token. Al entrenar con GSM, también se mezclan las muestras de entrenamiento de diferentes arquitecturas de conjuntos de instrucciones, lo que introduce una sintaxis dispar a nivel de ensamblador. Por tanto, se añade en última instancia la arquitectura, lo que ayuda al modelo a transferir la semántica de instrucción aprendida útil en una arquitectura a otra.

6. **Salida de STATEFORMER:** Cuando se encuentra en la fase de preentrenamiento con GSM, su salida consiste en una traza completa $\mu State$ que incluye tanto la traza de $\mu DataState$ como la traza de $\mu ControlState$. Cuando se encuentra en la fase de refinamiento de hiperparámetros para lograr la mayor tasa de acierto de cara a ejecutar la inferencia de tipos, la salida consiste en el conjunto de etiquetas con las predicciones realizadas.

La siguiente figura muestra en alto nivel la arquitectura de STATEFORMER:

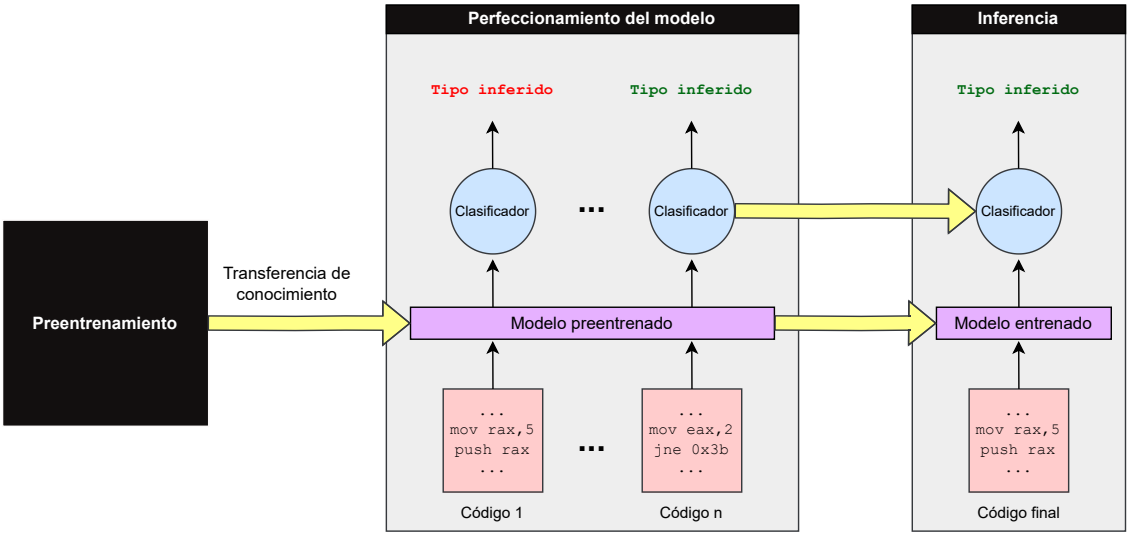


Figura 3.1: Arquitectura simplificada de STATEFORMER. Fuente propia.

Como se ha podido apreciar, el modelo de funcionamiento de STATEFORMER es complejo y abstracto. No obstante, entender sus elementos y componentes es de suma importancia para el trabajo desarrollado (se dan más detalles en cuanto a su estructura física y los diferentes elementos que lo componen en el Anexo C).

En conclusión, parece haber una tendencia en la que los sistemas para realizar inferencia de tipos desarrollados en los últimos años apuestan claramente por las capacidades del aprendizaje automático, aunque muchas de las soluciones publicadas recientemente se siguen apoyando en reglas, heurísticas, análisis estático y análisis dinámico [28, 53, 54].

Capítulo 4

Análisis y diseño del sistema desarrollado

Este capítulo describe la solución propuesta mediante su descomposición en los distintos aspectos técnicos del trabajo: primero se expone la metodología bajo la que se ha trabajado, y a continuación, se concretan las mejoras realizadas sobre la implementación original de STATEFORMER.

4.1. Metodología

Para la implementación de la solución se ha realizado, en primer lugar, un listado de las distintas herramientas con capacidad de analizar software, tanto de manera estática como de forma dinámica, con objeto de clasificar por las más extendidas entre la comunidad.

A continuación, siguiendo el procedimiento estándar para la realización de revisiones sistemáticas de la literatura [15,16], se investigó el estado del arte en materia de inferencia de tipos de dato, con apoyo de la herramienta `StArt` [18].

Una vez identificadas las técnicas más prometedoras y más comúnmente incorporadas en la actualidad, se pasó a estudiar cuál sería la más interesante de implementar en alguna de las herramientas extraídas anteriormente.

Durante la etapa de desarrollo, y puesto que el proyecto ha sido ejecutado por una sola persona, no se ha estimado oportuno seguir una metodología de trabajo específica, ya sea tradicional o ágil [6]. En su lugar, se ha aplicado una metodología genérica consistente en: (1) Análisis del problema, (2) Estudio del estado actual de la

solución original, (3) Programación de las mejoras software, (4) Testeo del software y (5) Puesta en producción.

4.2. Mejoras realizadas sobre la implementación original

STATEFORMER se ha reimplementado para las versiones de software Python 3.10, Fairseq 0.12.2 y PyTorch 2.1.0. Se ha considerado actualizarlo para que funcione sobre versiones más recientes que en las que estaba originalmente implementado (concretamente, Python 3.8, Fairseq 0.9.0 y PyTorch 1.3).

La implementación de STATEFORMER se ha llevado a cabo mediante la creación de un modelo llamado *RoBERTa* (explicado más adelante). Este modelo fue presentado en [61] como una versión mejorada y optimizada de *BERT* [62].

BERT (del inglés: *Bidirectional Encoder Representations from Transformers*) es un modelo centrado en el problema del procesamiento del lenguaje natural destacado por proponer el enfoque de *entrenamiento bidireccional*. Se entiende por entrenamiento bidireccional a aquel enfoque que analiza tanto las palabras anteriores como las palabras posteriores en una oración o un texto para obtener la mayor comprensión del contexto posible. Esto contrasta con el enfoque unidireccional, que solo considera las palabras anteriores o posteriores para la predicción de la siguiente palabra.

El *modus operandi* de *BERT* emplea modelos lingüísticos o trazas enmascaradas para permitir esta representación bidireccional. Esta aproximación enmascara aleatoriamente algunos de los tokens de la entrada, teniendo como objetivo predecir los valores originales basándose únicamente en su contexto. Este modelo reduce la necesidad de disponer de numerosas arquitecturas específicas para una misma tarea.

BERT consta de los dos pasos que ya se han comentado anteriormente:

- **Preentrenamiento:** Durante el preentrenamiento, el modelo se entrena con datos no etiquetados.
- **Ajuste:** En el ajuste, el modelo se inicializa primero con los parámetros preentrenados, y todos los parámetros se ajustan usando datos etiquetados de las tareas posteriores.

De la misma forma, *RoBERTa* (del inglés: *Robustly Optimized BERT Approach*) reutiliza esta idea incluyendo una cuidadosa revisión de los efectos que produce

ajustar en mayor medida los hiperparámetros y un mayor tamaño del conjunto de datos entrenamiento. Basándose en que el modelo *BERT* estaba significativamente infraentrenado, *RoBERTa* realiza las siguientes contribuciones: (1) entrena el modelo durante más tiempo, con lotes más grandes y sobre más datos; (2) elimina el objetivo de predicción sobre la siguiente frase; (3) entrena sobre secuencias más largas; y (4) cambia dinámicamente el patrón de enmascaramiento aplicado a los datos de entrenamiento.

Así pues, el modelo *RoBERTa* implementado incorpora también algunas modificaciones para añadir las capacidades de NAU (del inglés: *Neural Arithmetic Unit*) [46], que sirven para capturar y aprender a representar los valores numéricos que intervienen en las operaciones aritméticas.

Otro aspecto interesante a discutir es cómo se han de preparar los datos de entrada para entrenar un modelo o, mismamente, para analizar un fichero ejecutable. Como *verdad absoluta* durante el proceso de entrenamiento para verificar si sus predicciones son acertadas o no, es necesario pasarle como entrada binarios que hayan sido compilados con información de depuración (más concretamente, con lo que se conoce como formato DWARF [42]). DWARF es un formato de archivo de información de depuración utilizado por compiladores y depuradores para soportar la depuración a nivel de código fuente. Además, es independiente de la arquitectura y es aplicable a cualquier procesador o sistema operativo.

La Figura 4.1 muestra visualmente el flujo de ejecución para obtener los datos de entrada necesarios para el análisis mediante STATEFORMER y prepararlos de cara a ser consumidos:

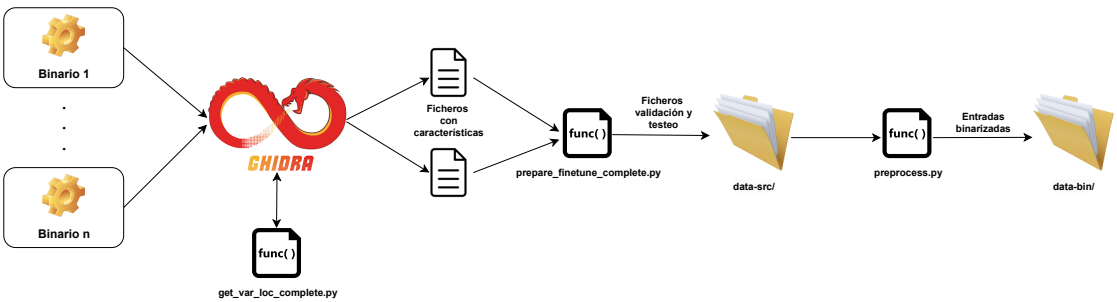


Figura 4.1: Preparación de datos para alimentar a STATEFORMER. Fuente propia.

Como se puede apreciar, la preparación de datos no es intuitiva, pues consta de varios pasos en los que intervienen distintos actores. Se ha desarrollado un script para unificar y facilitar todo el proceso de: (1) crear la estructura de directorios a

partir de la arquitectura para la que están compilados los ficheros ejecutables de entrada; (2) automatizar la creación de un proyecto en Ghidra para cada ejecutable; (3) analizar con Ghidra cada binario y extraer sus características utilizando la API de PyHidra [55]; (4) generar las entradas de validación y pruebas para cada ejecutable en su directorio correspondiente a partir de la información extraída en el paso anterior; y (5) *binarizar* (convertir de texto plano a formato binario) estos datos para su uso.

En este punto ya estarían los datos de entrada preparados para ser utilizados por STATEFORMER. De este modo, se reduce la cantidad de tiempo empleado en preparar estos datos de entrada, ya que previamente había que comprender exactamente cómo realizar este proceso y aplicarlo manualmente uno a uno para cada fichero ejecutable.

Adicionalmente, se ha extendido la solución original para proporcionar un punto de entrada unificado con el objetivo de facilitar la integración de STATEFORMER en otros sistemas o entornos de ingeniería inversa. Para ello, se ha desarrollado un script parametrizado que gestiona la acción a realizar entre preentrenar un modelo con GSM, entrenar el modelo, y utilizar el modelo para realizar predicciones (inferencias de tipos de dato). Dado que inicialmente STATEFORMER solo incorporaba las dos primeras funcionalidades (preentrenamiento y entrenamiento) y carecía de la funcionalidad de realizar predicciones a partir de un modelo (más allá de la tarea de validación que se ejecuta tras cada época), se le ha dotado de la posibilidad real de realizar inferencias sobre ficheros ejecutables desconocidos a partir de un modelo ya entrenado.

En última instancia, se ha incorporado la posibilidad de realizar toda la ejecución exclusivamente en CPU para aquellos equipos que no disponen de tarjeta gráfica NVIDIA dedicada. No obstante, este modo de uso es mucho más lento.

Capítulo 5

Experimentos y evaluación

En este capítulo se procede a evaluar los resultados obtenidos a partir de la nueva versión de STATEFORMER implementada. Para ello, se presenta el entorno de pruebas que se ha dispuesto y las métricas utilizadas para los experimentos. Después, se presenta una discusión de los resultados y, por último, las limitaciones encontradas.

5.1. Entorno de evaluación y métricas

Las pruebas se han realizado desde una máquina virtual Ubuntu 22.04.3, con un microprocesador Intel Core i9, 8 núcleos dedicados y 16GB de memoria RAM.

Para obtener la verdad absoluta, todo el conjunto de datos de binarios utilizados han sido compilados con toda la información de depuración incluida para, posteriormente, extraer las secciones DWARF utilizando un script no oficial de la herramienta Ghidra [11] proporcionado por el autor original de STATEFORMER [38] y ligeramente modificado para adaptarlo a las mejoras implementadas.

Durante las pruebas, no se ha dispuesto de tarjeta gráfica NVIDIA dedicada, lo cual ha condicionado las mismas al necesitar una excesiva cantidad de tiempo la ejecución de cada una de ellas.

Para evaluar el desempeño de STATEFORMER en distintos potenciales escenarios, se ha medido su rendimiento sobre binarios compilados para las arquitecturas MIPS, ARM, x86 y x86_64 con distintos niveles de optimización (concretamente, niveles de optimización O0, O1, O2 y O3) Estos escenarios se han evaluado con modelos entrenados en 50, 100 y 300 épocas.

Como conjunto de datos para las pruebas se ha usado un corpus de 33 proyectos software de código abierto, con un total de 1.670 millones de variables de distintos tipos. Este corpus se ha obtenido directamente del autor original de STATEFORMER [38]. Los proyectos se compilaron para las 4 arquitecturas objetivos utilizando GCC [31] versión 11.4.

Respecto a las métricas de resultados, se han utilizado *Precision*, *Recall* y *F1-score*. La *Precision* (P) es el valor que da la proporción de verdaderos positivos respecto al total de positivos que predice el modelo, es decir, el número de tipos de dato predichos correctamente, y se calcula mediante la fórmula:

$$P = \frac{TP}{(TP + FP)}$$

donde TP son los verdaderos positivos (del inglés, *True Positives*; es decir, aquellos tipos de datos identificados correctamente y FP son los falsos positivos (del inglés, *False Positives*; es decir, los tipos de dato que se han predicho erróneamente).

Por su parte, el *Recall* (R) es el valor que calcula lo bueno que es el modelo a la hora de encontrar todos los positivos, es decir, el número de tipos de dato que se han predicho correctamente respecto al total a predecir, y se calcula mediante la fórmula:

$$R = \frac{TP}{(TP + FN)}$$

entendiendo por FN los falsos negativos (del inglés, *False Negatives*; es decir, aquellos tipos de datos que no se han predicho correctamente).

Por último, el *F1-score* ($F1$) es la medida que combina los dos valores anteriores. Al existir un equilibrio entre P y R, $F1$ puede utilizarse para medir la eficacia general del modelo. Se calcula con la fórmula:

$$F1 = \frac{2 \quad P \quad R}{(P + R)}$$

5.2. Análisis de resultados

La Tabla 5.1 muestra las puntuaciones obtenidas con modelos entrenados 50 épocas sobre los distintos niveles de optimización y para cada arquitectura objetivo. De igual modo, la Tabla 5.2 hace lo propio para modelos entrenados 100 épocas. Finalmente, la Tabla 5.3 contiene los resultados obtenidos a partir de modelos entrenados 300 épocas.

Tabla 5.1: Resultados con modelo entrenado 50 épocas

Arquitectura	Optimización	*Precision*	*Recall*	*F1-score*
ARM	O0	50.2	40.1	44.5
	O1	41.5	30.9	35.4
	O2	40.3	28.8	33.5
	O3	43.2	31.0	36.0
x86	O0	49.8	32.5	39.2
	O1	50.8	39.3	44.3
	O2	47.0	40.2	43.3
	O3	45.9	38.5	41.8
x64	O0	54.1	30.4	38.9
	O1	41.7	30.6	35.2
	O2	42.9	29.6	35.0
	O3	48.1	27.9	35.3
MIPS	O0	56.0	37.2	44.7
	O1	53.3	35.9	42.9
	O2	44.2	30.7	36.2
	O3	41.4	31.6	35.8

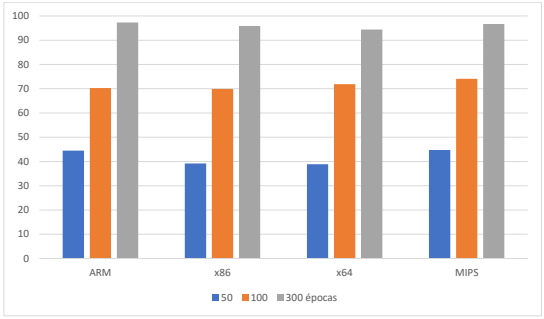
Tabla 5.2: Resultados con modelo entrenado 100 épocas

Arquitectura	Optimización	*Precision*	*Recall*	*F1-score*
ARM	O0	73.9	67.1	70.3
	O1	72.2	66.8	69.3
	O2	74.3	72.9	73.5
	O3	75.0	61.9	67.8
x86	O0	72.3	67.8	69.9
	O1	71.8	63.8	67.5
	O2	73.5	64.5	68.7
	O3	67.9	57.7	62.3
x64	O0	73.2	70.8	71.9
	O1	74.5	65.2	69.5
	O2	72.9	62.5	67.3
	O3	73.1	60.7	66.3
MIPS	O0	77.7	70.9	74.1
	O1	78.1	65.3	71.1
	O2	73.2	64.3	68.4
	O3	69.1	62	65.3

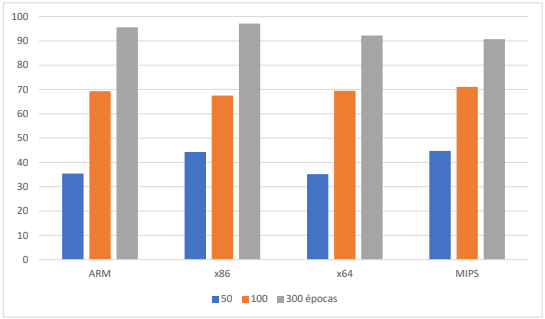
Tabla 5.3: Resultados de experimento 1 con modelo entrenado 300 épocas

Arquitectura	Optimización	*Precision*	*Recall*	*F1-score*
ARM	O0	98.7	96.0	97.3
	O1	97.1	94.3	95.6
	O2	94.5	94.2	94.3
	O3	92.6	91.3	91.9
x86	O0	97.6	94.4	95.9
	O1	97.3	97.1	97.1
	O2	95.5	94.1	94.7
	O3	96.0	94.4	95.1
x64	O0	97.9	91.2	94.4
	O1	93.8	90.8	92.2
	O2	92.3	91.9	92.0
	O3	96.1	90.2	93.0
MIPS	O0	97.7	95.9	96.7
	O1	93.1	88.6	90.7
	O2	95.2	91.3	93.2
	O3	89.9	84.2	86.9

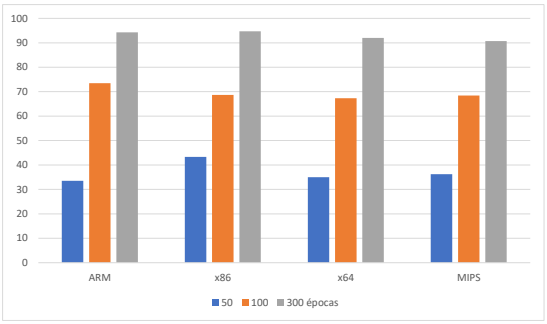
A continuación, la Figura 5.1 muestra los resultados de *F1-score* de los modelos entrenados en distintas épocas para cada arquitectura y para cada nivel de optimización, respectivamente.



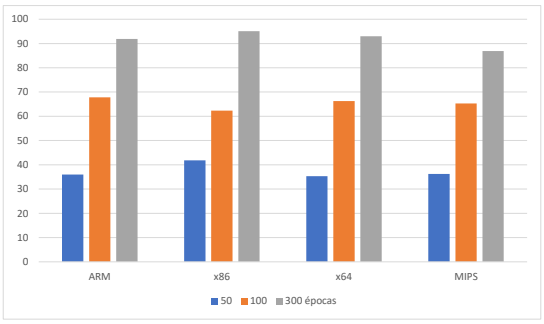
(a) Nivel de optimización O0



(b) Nivel de optimización O1



(c) Nivel de optimización O2



(d) Nivel de optimización O3

Figura 5.1: *F1-score*. Fuente propia.

Como se puede apreciar, STATEFORMER se comporta de forma robusta entre las diferentes arquitecturas y niveles de optimización. Como se intuía, a mayor número de épocas utilizadas para el entrenamiento, mejores resultados se alcanzan.

No obstante, resultan llamativas las puntuaciones alcanzas con los modelos entrenados 300 épocas, ya que se acercan al 100 % de tasa de acierto durante la fase de validación. Estos valores hacen sospechar de un posible sobreentrenamiento de los modelos. Sería interesante comprobar si este es el caso, ya que durante la fase de entrenamiento de un modelo, Fairseq se encarga de gestionar internamente la división pruebas/validación precisamente para evitar una posible situación de sobreentrenamiento, garantizando que cuando valida el modelo tras cada época lo hace sobre un conjunto de los datos desconocido. Sin embargo, esta fase de validación no se debe confundir con una situación real de inferencia sobre un binario desconocido. Para ello, Fairseq proporciona otra funcionalidad distinta (a través de `fairseq-generate`), con la cual se podría realizar esta comprobación.

5.2.1. Limitaciones

A pesar de los buenos resultados de STATEFORMER, el sistema todavía presenta algunas limitaciones. La principal carencia detectada es la incapacidad de realizar predicciones completas sobre tipos de dato complejos, como por ejemplo estructuras (e.g., el tipo de dato `struct` en C). Si bien es cierto que es capaz de revelar que cierto dato es una estructura, actualmente no es capaz de trabajar sobre los tipos de los datos de los campos de la estructura. Esto afecta negativamente en su puesta en producción sobre software complejo que maneje gran cantidad de estos elementos.

Otra limitación es que actualmente STATEFORMER requiere valores concretos en las trazas que utiliza tanto para validación como para entrenamiento. De cara a potenciar las capacidades de microejecución de código que emplea en la fase de preentrenamiento con GSM, debería ser compatible con la microejecución de trazas con valores simbólicos.

También se ha detectado que, si bien STATEFORMER funciona sobre binarios cuyo código se encuentra ofuscado, no funciona si el binario en sí se encuentra empaquetado o cifrado, como suele ser el caso para la mayoría del código dañino. Esto obliga a un preprocesamiento del binario previo a su uso con STATEFORMER.

Para concluir, es requisito indispensable al momento de redacción de este documento que el binario esté compilado utilizando una versión concreta de DWARF (concretamente, menor a la 5 y mayor o igual a la 2).

Capítulo 6

Conclusiones y trabajo futuro

Tras todo lo expuesto a lo largo del presente trabajo, se concluye este documento con el conjunto de conclusiones que se derivan del mismo, concretando las aportaciones realizadas. Se continúa con las líneas de trabajo futuro propuestas para continuar este proyecto, y se cierra con los problemas encontrados durante su ejecución.

6.1. Conclusiones

Durante el proyecto se ha realizado una revisión sistemática de la literatura sobre los trabajos comprendidos desde el año 2015 hasta la actualidad en el campo de la inferencia de tipos, con el objetivo de estudiar y comprender los mecanismos actuales con mejores resultados para, posteriormente, investigar su funcionamiento, examinar las técnicas de análisis que implementan y categorizar los fundamentos sobre los que se sustentan. Una de las principales conclusiones que se extraen del mismo es la creciente tendencia a apostar por los modelos centrados en abordar el problema de la inferencia de tipos de dato desde el aprendizaje automático. Esta conjunción entre los campos de la ciberseguridad y la inteligencia artificial empieza a resultar cada vez más lógica dados los avances de la tecnología, la hiperconectividad y el tratamiento masivo de datos hacia el que la globalización se dirige.

Las contribuciones realizadas mediante este trabajo permiten a futuros usuarios utilizar el sistema STATEFORMER de manera más intuitiva, al mejorar la documentación de uso y aportar scripts que facilitan su integración en otros entornos. Además, se ha refactorizado todo el código para que sea compatible con las últimas versiones de las tecnologías que maneja internamente, con las mejoras y optimizaciones que eso conlleva. De igual modo, se ha habilitado la posibilidad de entrenar

modelos exclusivamente en CPU, para aquellos que no dispongan de sistemas con tarjetas gráficas NVIDIA. Finalmente, se ha evaluado el modelo en mayor profundidad, pudiendo identificar nuevas limitaciones, y se han generado modelos entrenados en 50, 100 y 300 épocas para las distintas arquitecturas y niveles de optimización.

6.2. Trabajo futuro

Esta investigación puede continuarse de múltiples formas. A continuación, se lista un conjunto de posibles contribuciones que resultan interesantes:

- **Integración en entornos de ingeniería inversa:** Acoplar STATEFORMER a un entorno actual para la realización de ingeniería inversa (como por ejemplo Angr [12]), no solo incrementaría exponencialmente su adopción y visibilidad, sino que también generaría modelos mucho más robustos y precisos. Además, dotaría a STATEFORMER de cierta interfaz visual.
- **Mayor experimentación:** Sería interesante continuar estudiando el comportamiento de STATEFORMER en otros casos de uso como (1) distintos mecanismos de ofuscación de código, (2) otras arquitecturas (como PowerPC) o (3) diferentes fases de preentrenamiento con GSM.
- **Predecir trazas simbólicas:** En la actualidad, STATEFORMER predice sobre trazas concretas. Por tanto, sería interesante que pudiera predecir sobre trazas simbólicas.
- **Predicciones más complejas:** STATEFORMER es capaz de predecir la estructura de un tipo de dato pero no lo que almacena. Sería de interés extenderlo para que incorporar jerarquía de tipos estructurados o tipos de dato nuevos.
- **Mejorar la fase de preentrenamiento con GSM:** Puesto que para este trabajo se ha considerado la fase de preentrenamiento con GSM como una caja negra, sería de utilidad ahondar en dicho sistema y tratar de obtener mejoras que permitan una trasferencia de conocimiento de mayor calidad a STATEFORMER.
- **Mejorar la recolección de metadatos:** Podría ayudar a obtener un mejor conocimiento de la semántica del código el investigar posibles mejoras a la hora de extraer metadatos del mismo.

6.3. Problemas encontrados

La ejecución de este proyecto ha supuesto un reto desafiante a la par que enriquecedor. Su realización ha servido como oportunidad de adquirir un profundo entendimiento sobre las técnicas y los sistemas emergentes en el ámbito de la inferencia de tipos de dato dentro del análisis de binarios.

Dado que el trabajo involucraba conceptos relativamente avanzados de aprendizaje automático, y unido a que no se disponía de experiencia alguna en dicho campo antes de comenzar esta tesis, ha sido necesario enfrentar una considerable curva de aprendizaje para lograr entender todos los conceptos, entornos y algoritmos involucrados. Esto ha sido, sin lugar a duda, la principal barrera a sortear, pudiendo concluir que ha sido superada con éxito tras la conclusión de este trabajo.

Con respecto a la implementación, el principal inconveniente ha sido actualizar el código para que funcione únicamente en CPU y con las versiones más recientes de Fairseq, NumPy y PyTorch, ya que ha sido necesaria una profunda refactorización de todo el código base. También ha resultado un obstáculo el hecho de que STATEFORMER se encontraba en una versión más cercana a la prueba de concepto que a ser verdaderamente explotado en producción, siendo necesario aportar funcionalidad básica para acercar el proyecto a una versión lista para desplegarse en producción.

Por último, el otro gran obstáculo ha estado en la experimentación, ya que si bien estos modelos de aprendizaje automático se pueden llegar a ejecutar en CPU, están pensados para ejecutarse en equipos no solo con una, sino varias GPU en paralelo, sumado a gran capacidad de memoria RAM. Debido a que no ha sido posible contar con dichas especificaciones técnicas, se ha requerido una ingente cantidad de horas para conseguir obtener los modelos entrenados. Esta traba ha sido especialmente molesta ya que en varias ocasiones la alta demanda de recursos computacionales ha ocasionado reinicios involuntarios durante el proceso de entrenamiento, ralentizándolo excesivamente.

Bibliografía

[1] D. Badampudi, R. Britto y M. Unterkalmsteiner, “Modern code reviews - Preliminary results of a systematic mapping study”, *Conference: Evaluation and Assessment in Software Engineering*, 2019, pp. 340-345.

[2] L. Harris y B. Miller, “Practical analysis of stripped binary code”, *SIGARCH Computer Architecture News*, vol. 33, 2005, pp. 63-68.

[3] L. Chen, “What Exactly Determines the Type? Inferring Types with Context”, *50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental*, 2020, pp. 71-72.

[4] J. Caballero y Z. Lin, (2016). “Type Inference on Executables”, *ACM Computing Surveys*, 2016, vol. 48, pp. 1-35.

[5] T. Wang y S.F. Smith, “Precise Constraint-Based Type Inference for Java”, *European Conference on Object-Oriented Programming*, 2005, vol. 2072, pp. 99-117.

[6] M. Javanmard y M. Alian, “Comparison between Agile and Traditional software development methodologies”, *2nd National Conference on Applied Research in Computer Science and Information Technology*, 2015.

[7] D. Vujičić y S. Ranđić, “New Challenges in Computer Architecture Education”, *9th International scientific conference Technics and Informatics in Education – TIE 2022*, 2022, pp. 141-147.

[8] Z. Xu, C. Wen, S. Qin, “Type Learning for Binaries and Its Applications”, *IEEE Transactions on Reliability*, 2018, pp. 1-20.

[9] S. Comeau, “Symbolic Execution for Function Matching”, *Worcester Polytechnic Institute*, 2017.

[10] A. Fokin, E. Derevenetc, A. Chernov, K. Troshina, “SmartDec: approaching C++ decompilation”, *Reverse Engineering*, 2011, pp. 347–356

[11] *GHidra - A software reverse engineering suite of tools developed in support of the Cybersecurity mission*. NSA Research Directorate. [Online]. Disponible: https://ghidra-sre.org/

[12] Angr analysis framework Angr. [Online]. Disponible: https://angr.io/

[13] Binary Ninja analysis platform: built by reverse engineers, for reverse engineers Vector 35. [Online]. Disponible: https://binary.ninja/

[14] *The IDA Pro and Hex-Rays*. Hex-Rays. [Online]. Disponible: https://www.hex-rays.com/idapro/

[15] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey y S. Linkman, "Systematic literature reviews in software engineering—A systematic literature review", *Inf. Softw. Technol.*, 2009, vol. 51, pp. 7–15.

[16] A. P. Siddaway, A. M. Wood y L. V. Hedges, "How to do a systematic review: A best practice guide for conducting and reporting narrative reviews, meta-analyses, and meta-syntheses", *Annu. Rev. Psychol.*, 2019, vol. 70, pp. 747–770.

[17] *Preferred Reporting Items for Systematic Reviews and Meta-Analyses*. PRISMA. [Online]. Disponible: https://www.prisma-statement.org/

[18] A. Zamboni, A. Thommazo, E. Hernandes y S. Fabbri, "StArt uma ferramenta computacional de apoioà revisão sistemática", *Congresso Brasileiro de Softw. (CBSoft)*, 2010, pp. 91–96.

[19] E. Hernandes, A. Zamboni, S. Fabbri y A. Di Thommazo, "Using GQM and TAM to evaluate StArt—A tool that supports systematic review", *CLEI Electron. J.*, 2012, vol. 15.

[20] F. Allen, "Control Flow Analysis", *Association for Computing Machinery*, 1970, vol. 5, pp. 1-19.

[21] A. Gosain y G. Sharma, "A Survey of Dynamic Program Analysis Techniques and Tools", *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA)*, 2015, pp. 113-122.

[22] R. Baldoni, E. Coppa, D. Cono D'elia, C. Demetrescu e I. Finocchi, "A Survey of Symbolic Execution Techniques", *ACM Comput. Surv.*, 2018, vol. 51, pp. 1-39.

[23] S. Parsa, "Automatic Test Data Generation Symbolic and Concolic Executions", *Software Testing Automation: Testability Evaluation, Refactoring, Test Data Generation and Fault Localization*, 2023, pp. 503-542.

[24] D. Monniaux, “A Survey of Satisfiability Modulo Theory”, 2016, vol. 9890, pp. 401-425.

[25] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel and G. Vigna, “SoK: (State of) The Art of War: O□ensive Techniques in Binary Analysis”, *IEEE Symposium on Security and Privacy*, 2016.

[26] L. Xun, “A linux executable editing library”, *Masters Dissertation, National University of Singapore*, 1999. [Online]. Disponible: http://www.geocities.com/fasterlu/leel.htm

[27] M. Prasad y T. Chiueh, “A Binary Rewriting Defense Against Stack-based Bu□er Overflow Attacks”, *USENIX Annual Technical Conference*, 2003.

[28] J.H. Lee, T. Avgerinos y D. Brumley, “Tie: principled reverse engineering of types in binary programs”, *Network and Distributed System Security Symposium*, 2011.

[29] M. Noonan, A. Loginov y D. Cok, “Polymorphic type inference for machine code”, *ACM Sigplan Conference on Programming Language Design and Implementation*, 2016, pp. 27–41

[30] Z. Lin, X. Zhang y D. Xu, “Automatic reverse engineering of data structures from binary execution”, *Network and Distributed System Security Symposium*, 2010.

[31] GCC, the GNU Compiler Collection. Proyecto GNU. Disponible: https://gcc.gnu.org/

[32] *Pin - A Dynamic Binary Instrumentation Tool*. Intel. [Online]. Disponible: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html

[33] *Valgrind framework*. Valgrind. [Online]. Disponible: https://valgrind.org/

[34] *Facebook AI Research*. FAIR. [Online]. Disponible: https://opensource.fb.com/

[35] T. Rupprecht, X. Chen, D. White, J. Tobias, H. Bos y G. Lüttgen, “POSTER: Identifying Dynamic Data Structures in malware”, *In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS ’16). Association for Computing Machinery*, 2016, pp. 1772–1774.

[36] S. Alrabaee, P. Shirani, L. Wang, M. Debbabi, “FOSSIL: A resilient and e□cient system for identifying FOSS functions in malware binaries”, *ACM Transactions on Privacy and Security*, 2018, vol. 21, pp. 1-34.

[37] Q. Jing, S. Xiaohong y M. Peijun, “Using Reduced Execution Flow Graph to Identify Library Functions in Binary Code”, *IEEE Transactions on Software Engineering*, 2016, pp. 1–15

[38] K. Pei, J. Guan, M. Broughton, Z. Chen, S. Yao, D. Williams-King, V. Ummadisetty, J. Yang, B. Ray y S. Jana, “StateFormer: fine-grained type recovery from binaries using generative state modeling”, *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 690-702.

[39] Z. Leong Chua, S. Shen, P. Saxena y Z. Liang, “Neural Nets Can Learn Function Type Signatures From Binaries”, *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 99-116.

[40] A. Maier, H. Gascon, C. Wressnegger y K. Rieck, “TypeMiner: Recovering types in binary programs using machine learning”, *International Conference on Detection of Intrusions and malware, and Vulnerability Assessment*, 2019, pp. 288-308.

[41] P. Geurts, D. Ernst y L. Wehenkel, “Extremely randomized trees”, *Machine Learning 63*, 2006, pp. 3–42.

[42] *The DWARF Debugging Standard*. [Online]. Disponible: http://dwarfstd.org/

[43] E. Robbins, A. King y T. Schrijvers, “From MinX to MinC: semantics-driven decompilation of recursive datatypes”, *ACM SIGPLAN Notices*, 2016, vol. 51, pp. 191-203.

[44] Z. Pavlinovic, Y. Su y T. Wies, “Data flow refinement type inference”, *Proceedings of the ACM on Programming Languages*, 2021, vol. 5, pp. 1-31.

[45] I. Haq y J. Caballero, “A Survey of Binary Code Similarity”, *ACM Computing Surveys*, 2021, vol. 54, pp. 1-38.

[46] A. Madsen y A. R. Johansen, “Neural Arithmetic Units”, *International Conference on Learning Representations*, 2020

[47] B. Chang, C. Evan, N. Adam, C. George, Schneck y R. Robert, “Type-Based Verification of Sssembly Language for Compiler Debugging”, *Proceedings of the*

2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, 2005, pp. 91–102.

[48] B. McNamara y Y. Smaragdakis, "Functional Programming in C++", in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, 2000, pp. 118–129.

[49] K. Elwazeer, K. Anand, A. Kotha, M. Smithson, R. Barua, "Scalable variable and data type detection in a binary rewriter", *ACM Sigplan Conference on Programming Language Design and Implementation*, 2013, pp. 51–60.

[50] L. Chen, Z. He y B. Mao, "CATI: Context-Assisted Type Inference from Stripped Binaries", *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020, pp. 88–98.

[51] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, y J. Dean, "Distributed representations of words and phrases and their compositionality", *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[52] J. He, P. Ivanov, P. Tsankov, V. Raychev, M. Vechev, "Debin: Predicting Debug Information in Stripped Binaries", *Conference: the 2018 ACM SIGSAC Conference* 2018, pp. 1667-1680.

[53] M. Hassan, C. Urban, M. Eilers y P. Müller, "MaxSMT-based type inference for Python 3", *International Conference on Computer Aided Verification*, 2018.

[54] E. Çiçek, W. Qu, G. Barthe, M. Gaboardi y D. Garg, "Bidirectional type checking for relational properties", *40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019.

[55] *PyHidra*. Department of Defense Cyber Crime Centre (DC3). [Online]. Disponible: https://github.com/dod-cyber-crime-center/pyhidra

[56] *PyTorch*. Facebook's AI Research Lab. [Online]. Disponible: https://pytorch.org/

[57] *TensorFlow*. Google Brain. [Online]. Disponible: https://www.tensorflow.org/

[58] *Keras: Deep Learning for Humans*. F. Chollet. [Online]. Disponible: https://keras.io/

[59] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier y M. Auli, "fairseq: A Fast, Extensible Toolkit for Sequence Modeling", *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[60] *Awesome Info Inferring Binary*. Yasong. [Online]. Disponible: https://github.com/yasong/Awesome-Info-Inferring-Binary

[61] L. Zhuang, L. Wayne, S. Ya y Z. Jun, "A Robustly Optimized BERT Pre-training Approach with Post-training", *Proceedings of the 20th Chinese National Conference on Computational Linguistics*, 2021, pp. 1218-1227.

[62] J. Devlin, M.W. Chang, K. Lee, K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019. vol. 1, pp. 4171-4186.

Anexo A

Horas invertidas

El presente anexo refleja el seguimiento realizado durante el desarrollo del trabajo, así como el desglose de horas que se han dedicado a las distintas fases del mismo.

El seguimiento del trabajo con el tutor se ha llevado a cabo mediante reuniones principalmente semanales, con algún período bisemanal. En estas reuniones se ha hablado sobre el trabajo llevado a cabo en esas semanas, aclarado dudas y realizado planificaciones para las siguientes reuniones.

El proyecto ha tenido una estimación de 300 horas, habiendo dedicado un total de 455 horas tras la conclusión de todos los hitos. La Tabla A.1 refleja las distintas fases en las que se ha dividido el trabajo. Adicionalmente, en la Figura A.1 se muestra el diagrama de Gantt del proyecto:

Tabla A.1: Desglose de hitos y horas invertidas

Hito	Descripción	Horas invertidas
1. Revisión sistemática de la literatura	Investigación del estado del arte en busca de los sistemas de inferencia de tipos de dato relacionados con el trabajo	70
2. Análisis de sistemas	Estudio en profundidad de los sistemas destacados y extracción de sus características	30
3. Familiarización con STATEFORMER	Exploración del sistema STATEFORMER y sus tecnologías, así como su funcionalidad y modos de operación	25
4. Implementación	Refactorización del código de STATEFORMER y desarrollo de scripts para mejorar su operabilidad	40
5. Experimentación	Obtención de modelos entrenados y ejecución de las pruebas pertinentes para verificar su comportamientos	245
6. Redacción de la memoria	Redacción de la memoria del proyecto	50
7. Revisión de la memoria	Realización de cambios y ajustes en la memorias tras el *feedback* recibido por parte del tutor	10

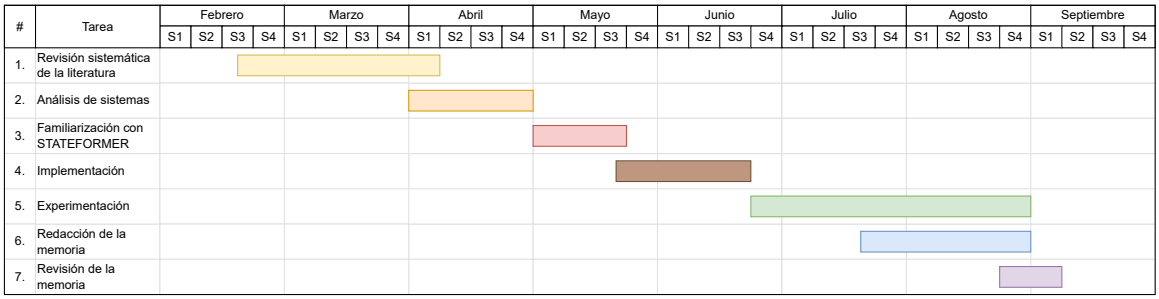


Figura A.1: Diagrama de Gantt. Fuente propia.

Anexo B

Preentrenamiento de STATEFORMER con GSM

En este anexo se expone el funcionamiento de *Generative State Modeling* (GSM). Puesto que no es una explicación trivial, se recurre al gráfico que representa la visión completa de lo que es STATEFORMER en conjunción con GSM a modo de apoyo visual:

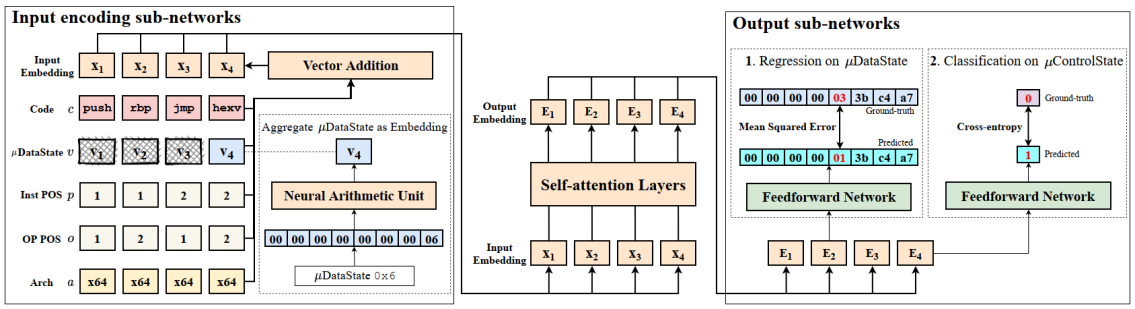


Figura B.1: Modelo de entradas-salida de STATEFORMER cuando se entrena con GSM. Fuente [38].

Cuando se hace una fase de preentrenamiento con GSM, se ejecutan una serie de pasos previos a STATEFORMER con el objetivo de aprender de manera explícita la semántica operativa del código, cuyo objetivo es lograr una inferencia de tipos más precisa y robusta. Así, esta tarea se descompone en los siguientes elementos:

1. **Módulo de representación numérica:** Cada valor de $\mu DataState$ se representa como una secuencia de 8 bytes. Para aprender las interdependencias entre bytes altos y bajos, se utiliza una NAU [46].

2. **Subconjunto de muestreo de $\mu DataState$:** Se extrae un subconjunto aleatorio de $\mu DataState$ y se sustituye por <MASK> en la entrada del modelo, de modo que el modelo esté entrenado para reconstruir el $\mu DataState$ eliminado.

3. **Módulo de codificación:** La NAU se aplica solo a secuencias de $\mu DataState$. Se termina conformando una secuencia final en la que cada elemento es fruto de la inserción del valor posicional relativo a cada token de los cinco descritos con anterioridad (caja en el extremo izquierdo en la Figura B.1). A continuación, se calcula la suma vectorial de estas cinco inserciones y se obtiene un único valor que es el que finalmente se inserta en el vector que sirve como entrada a la red neuronal de GSM (parte central en la Figura B.1).

4. **Funciones de pérdida:** Finalmente, tras obtener el resultado del paso anterior en forma de vector de elementos, este se pasa a dos redes de dos capas independientes, que dan como salida la predicción de $\mu DataState$ y de $\mu ControlState$ (caja en el extremo derecho en la Figura B.1).

Como se puede entrever, la fase de preentrenamiento con GSM requiere de unos recursos computacionales y temporales inabordables en el alcance del proyecto. Por ello, se ha utilizado en su lugar un modelo ya preentrenado con GSM durante 10 épocas.

No obstante, en este punto, y refiriendo al estudio original [38], cabe resaltar los prometedores resultados en los que se hace referencia a que los modelos preentrenados con GSM presentan un incremento en la tasa de acierto de hasta el 33 % con respecto a los que no lo utilizan.

Anexo C

Estructura física de STATEFORMER

En este anexo se documentan las rutas y los archivos más relevantes de cara al uso y la modificación/extensión del sistema STATEFORMER. Entre estos componentes se encuentra Fairseq (explicada brevemente en la Sección 2.4). Si bien la estructura completa de Fairseq no se va a desarrollar, sí se hará lo propio con alguno de los archivos que integra, ya que definen el núcleo de lo que es STATEFORMER.

La principal motivación de esto es que la estructura de ficheros y directorios no es especialmente simple, pudiendo llegar a requerir emplear más tiempo del previsto en localizar los diferentes actores que toman parte en el sistema y en discernir dónde o sobre cuál trabajar para implementar la funcionalidad deseada.

De esta forma, los principales directorios y ficheros a tener en cuenta son:

- `command/`: Directorio que alberga los scripts encargados de gestionar las ejecuciones de STATEFORMER y de preparar los datos de entrada.
    - `finetune/`:

        `finetune.sh`: Programa encargado de asignar los valores de configuración de STATEFORMER y de lanzar la tarea de entrenamiento y validación, ajustando en cada iteración o época los hiperparámetros de la red neuronal.

        `preprocess.py`: Script encargado del preprocesamiento de los datos requeridos para la tarea de entrenamiento y validación, esto es, creación de vocabularios y binarización de los mismos.

  - `pretrain/`: Contiene los scripts encargados de preparar los datos y lanza la tarea para preentrenar STATEFORMER con GSM. Puesto que durante el proyecto se ha utilizado un modelo ya preentrenado, no se ha hecho uso de estos.

    `preprocess_pretrain.py`: Script encargado del preprocesamiento de los datos requeridos para la tarea de preentrenamiento.

    `pretrain.sh`: Script que lanza la tarea de preentrenamiento con GSM pasándole la configuración y argumentos oportunos.

  - `get_var_loc_complete.py`: Plugin de Ghidra para extraer todas la características del binario requeridas para preparar los datos de entrada que utiliza STATEFORMER. Genera como resultado un fichero "nombre_stacks" con toda esta información.

  - `prepare_finetune_complete.py`: Script que recoge el fichero "nombre_stacks" anterior y genera todos los archivos de entrada para alimentar a STATEFORMER

- `data-src/`: Directorio donde guardar los archivos obtenidos tras ejecutar `prepare_finetune_complete.py`. Estos ficheros contienen la información necesaria tanto para el proceso de entrenamiento como el de validación. Estos ficheros tienen la información en texto plano.
- `data-bin/`: Directorio que almacena los mismos datos que `data-src/` una vez se han binarizado, ya que STATEFORMER no trabaja con entradas en texto plano.
- `checkpoint/`: Directorio en el que se guardan tanto el último como el mejor resultado obtenido durante las fases de preentrenamiento y entrenamiento respectivamente.
- `result/`: Directorio donde se registra el resultado de STATEFORMER tras el proceso de validación.
- `fairseq/`: Directorio raiz del *framework* fairseq. Las rutas de mayor relevancia son:

  - `criterions/`: Directorio que almacena los scripts encargados de calcular la función de pérdida dado el modelo y la traza analizada.

    `masked_lm_mf.py`: Script que calcula la función de pérdida para el modelo que utiliza STATEFORMER

`signature.py`: Script encargado de ajustar los hiperparámetros para predecir los tipos de dato.

- `models/`: Contiene los diferentes modelos de aprendizaje.

  `roberta_mf/`: Modelo implementado, desarrollado en mayor detalle en la Sección 4.2 para facilitar la lectura de este esquema.

- `modules/`: Directorio con multitud de funcionalidades que utilizan por debajo los modelos de aprendizaje automático (véase eliminar ruido, aplicar transformaciones, gestionar capas, etc). A destacar:

  `transformer_sentence_encoder_mf_nau`: Implementación del codificador de trazas para la red neuronal. Más información en la Sección 4.2.

- `tasks/`: Almacena las funciones para cargar el *dataset* y recorrerlo, además de inicializar el modelo y aplicar las funciones de pérdida.

  `masked_lm_mf_.py`: Script encargado de preparar el conjunto de datos para preentrenar con GSM.

  `signature.py`: Script que controla cómo ajustar el conjunto de datos al modelo y cómo entrenarlo.

El resto de archivos y directorios que contiene STATEFORMER no son relevantes para este trabajo, pues o bien: (1) no han requerido modificación alguna y, además, no han sido necesarios para el correcto entendimiento del sistema, (2) son scripts de instalación/configuración/funcionalidad secundaria, o (3) son archivos con documentación o scripts para realizar pruebas funcionales y/o de integración.