



universidad
de león

Escuela de Ingenierías I.I.



Industrial, Informática y Aeroespacial

MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN CIBERSEGURIDAD

Trabajo de Fin de Máster

Evaluación comparativa de representaciones
intermedias para la capacidad de abstracción de
arquitecturas de hardware

Comparative Evaluation of Intermediate
Representations for Hardware Architecture
Abstraction Capability

Autor: Pablo de Baro Escapa

Tutora: Camino Fernández Llamas

Co-Tutor: Ricardo Julio Rodríguez Fernández

(Septiembre, 2026)

UNIVERSIDAD DE LEÓN
Escuela de Ingenierías I.I.
MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN CIBERSEGURIDAD
Trabajo de Fin de Máster

ALUMNO: Pablo de Baro Escapa

TUTORA: Camino Fernández Llamas

CO-TUTOR: Ricardo Julio Rodríguez Fernández

TÍTULO: Evaluación comparativa de representaciones intermedias para la capacidad de abstracción de arquitecturas de hardware

TITLE: Comparative Evaluation of Intermediate Representations for Hardware Architecture Abstraction Capability

CONVOCATORIA: Septiembre, 2026

RESUMEN:

El análisis estático de binarios se apoya en una traducción previa del código máquina a una representación intermedia (IR) que abstrae las particularidades de la arquitectura de origen. Existen diversos sistemas para llevar a cabo estas traducciones. Sin embargo, cada sistema diseña su propia IR con filosofías y niveles de abstracción heterogéneos, sin que exista en la literatura un estudio que las compare de forma sistemática. Este Trabajo de Fin de Máster aborda esta laguna comparando seis representaciones intermedias de uso extendido: P-code, BNIL, VEX IR, ESIL, LLVM IR y Hex-Rays Microcode. La comparativa cualitativa establece catorce dimensiones (nivel de abstracción, modelo de memoria, expresividad semántica, forma SSA, entre otras), contrastadas contra fuentes primarias y documentación oficial. Por su parte, la comparativa cuantitativa define cuatro bloques de métricas (verbosidad, coste, robustez y agnosticismo interarquitectura) evaluados sobre un corpus de 960 binarios en formato ELF, compilados para ocho combinaciones de arquitectura y anchura de registro, con seis compiladores y cuatro niveles de optimización. Los resultados confirman que el nivel de abstracción no predice la verbosidad, pero sí el agnosticismo interarquitectura dentro de una misma familia; identifican a P-code y a Hex-Rays Microcode como las IR con mayor invarianza entre arquitecturas, aunque con un coste temporal muy distinto entre ambos: P-code es el más lento del conjunto, mientras que Hex-Rays Microcode se sitúa en una posición intermedia; señalan que RetDec no genera LLVM IR en el 31,04 % de las ejecuciones del corpus; y validan el diseño metodológico mediante el par de control MIPS-32 y MIPSEB-32, que difieren únicamente en el orden de bytes.

Palabras clave: representaciones intermedias; ingeniería inversa; abstracción de hardware; análisis de binarios

Firma del alumno:

VºBº Tutora:

VºBº Co-Tutor:

Agradecimientos

Quiero expresar mi agradecimiento a Ricardo J. Rodríguez por su guía, dedicación y apoyo a lo largo de todo el proyecto, así como por facilitarme el acceso a **Binary Ninja**. Gracias por sus consejos, por la atención prestada durante el desarrollo del trabajo y por ayudarme a llevarlo hasta el resultado final.

A Javier Carrillo, por facilitarme el acceso a **IDA Pro**, lo que permitió incluir esta herramienta en el desarrollo del trabajo.

A mi familia, por su apoyo incondicional, por estar siempre ahí y por acompañarme durante todo este camino.

Y a los chavales del Memphis, por estar presentes durante estos meses, por los buenos momentos y por hacer que el camino hasta aquí fuese mucho más llevadero.

A todos, gracias.

Abstract

Static binary analysis relies on translating native machine code into an intermediate representation (IR) that abstracts away the particularities of the source architecture. Several systems are available for performing these translations. However, each system defines its own IR, following heterogeneous design philosophies and abstraction levels, and no systematic comparison of them has been presented in the literature. This Master’s Thesis addresses this gap by comparing six widely used intermediate representations: P-code, BNIL, VEX IR, ESIL, LLVM IR, and Hex-Rays Microcode. The qualitative comparison establishes fourteen dimensions, including abstraction level, memory model, semantic expressiveness, and SSA form, based on primary sources and official documentation. The quantitative comparison defines four metric blocks (verbosity, cost, robustness, and cross-architecture agnosticism) evaluated on a corpus of 960 ELF binaries compiled for eight architecture and register-width combinations, using six compilers and four optimization levels. The results confirm that abstraction level does not predict verbosity, but does predict cross-architecture agnosticism within a single family. They identify P-code and Hex-Rays Microcode as the IRs with the highest cross-architecture invariance, although their runtime costs differ substantially: P-code is the slowest backend, whereas Hex-Rays Microcode occupies an intermediate position. The results also show that `RetDec` fails to generate LLVM IR for 31.04% of the corpus executions, and validate the methodological design through the MIPS-32 and MIPSEB-32 control pair, which differ only in byte order.

Keywords: intermediate representations, reverse engineering, hardware abstraction, binary analysis

Índice general

Agradecimientos	II
Abstract	III
Índice de figuras	VI
Índice de tablas	VII
Glosario de términos	IX
1. Introducción	1
1.1. Motivación	1
1.2. Objetivos	2
1.3. Estructura del trabajo	2
2. Conocimientos previos	4
2.1. Análisis estático de binarios	4
2.2. Representaciones intermedias en el análisis de software	6
3. Estado del arte	9
3.1. Comparación de las IR	9
3.2. Carencias identificadas	11
4. Comparativa cualitativa	13
4.1. Arquitecturas de conjunto de instrucciones consideradas	13
4.2. Herramientas y entornos seleccionados	14
4.3. Dimensiones de comparación	17
4.4. Comparación y análisis por dimensión	20
5. Metodología de la comparativa cuantitativa de las representaciones intermedias	29

5.1. Diseño general del marco de evaluación	29
5.2. Corpus de pruebas	34
5.3. Bloques de métricas	35
5.3.1. Verbosidad y estructura	35
5.3.2. Coste	38
5.3.3. Robustez	39
5.3.4. Agnosticismo interarquitectura	40
6. Experimentos y evaluación	43
6.1. Entorno de ejecución	43
6.2. Resultados por bloque de métricas	45
6.3. Discusión comparativa de las IR	71
6.4. Limitaciones	75
7. Conclusiones y trabajo futuro	79
Bibliografía	81
A. Tiempo invertido y diagrama de Gantt	88

Índice de figuras

2.1. Esquema directo <i>frontend</i> a <i>backend</i> : cada compilador debe generar código para cada arquitectura de destino ($N \times M$ compiladores); inspirada en [1]	6
2.2. Esquema con IR común como <i>middle-end</i> : el mismo lenguaje intermedio conecta todos los <i>frontends</i> con todos los <i>backends</i> ($N + M$ componentes); inspirada en [1]	7
5.1. Arquitectura general del <i>pipeline</i> de extracción y cálculo de métricas.	32
A.1. Diagrama de Gantt con la planificación del proyecto	88

Índice de tablas

4.1. Comparativa cualitativa de las seis representaciones intermedias consideradas.	28
6.1. Versiones del software empleado en el entorno de ejecución.	44
6.2. Ratio de expansión (ops, total), mediana (IQR), por arquitectura y <i>backend</i>	55
6.3. Ratio de expansión (ops, aritmética), mediana (IQR), por arquitectura y <i>backend</i>	55
6.4. Ratio de expansión (ops, control de flujo), mediana (IQR), por arquitectura y <i>backend</i>	56
6.5. Ratio de expansión (ops, acceso a memoria), mediana (IQR), por arquitectura y <i>backend</i>	56
6.6. Ratio de expansión (ast, total), mediana (IQR), por arquitectura y <i>backend</i>	57
6.7. Ratio de expansión (ast, aritmética), mediana (IQR), por arquitectura y <i>backend</i>	57
6.8. Ratio de expansión (ast, control de flujo), mediana (IQR), por arquitectura y <i>backend</i>	58
6.9. Ratio de expansión (ast, acceso a memoria), mediana (IQR), por arquitectura y <i>backend</i>	58
6.10. Temporales por instrucción, mediana (IQR), por arquitectura y <i>backend</i>	59
6.11. Profundidad de anidamiento media, mediana (IQR), por arquitectura y <i>backend</i>	59
6.12. Profundidad de anidamiento máxima, mediana (IQR), por arquitectura y <i>backend</i>	60
6.13. Operaciones SSA, mediana (IQR), por arquitectura y <i>backend</i>	60
6.14. Tiempo de lifting (interno), mediana (IQR) en segundos, por arquitectura y <i>backend</i>	61
6.15. Memoria pico en MiB, mediana (IQR), por arquitectura y <i>backend</i>	61

6.16. Tasa de error, categoría aritmética, media, por arquitectura y <i>backend</i> .	62
6.17. Tasa de error, categoría de control de flujo, media, por arquitectura y <i>backend</i>	62
6.18. Tasa de error, categoría de acceso a memoria, media, por arquitectura y <i>backend</i>	63
6.19. Tasa de error, categoría otras, media, por arquitectura y <i>backend</i> . . .	63
6.20. Fracción de escape, media, por arquitectura y <i>backend</i>	64
6.21. Similitud ponderada de Jaccard, mediana, por par de arquitectura pura y <i>backend</i>	65
6.22. Similitud de Jensen-Shannon, mediana, por par de arquitectura pura y <i>backend</i>	66
6.23. Coeficiente de variación del tamaño de la IR, mediana, por par de arquitectura pura y <i>backend</i>	67
6.24. Delta de complejidad ciclomática, media, por par de arquitectura pura y <i>backend</i>	68
6.25. Similitud ponderada de Jaccard, mediana, por par de anchura pura y <i>backend</i>	69
6.26. Similitud de Jensen-Shannon, mediana, por par de anchura pura y <i>backend</i>	69
6.27. Coeficiente de variación del tamaño de la IR, mediana, por par de anchura pura y <i>backend</i>	70
6.28. Delta de complejidad ciclomática, media, por par de anchura pura y <i>backend</i>	70
A.1. Estimación de horas invertidas por fase y tarea.	89

Glosario de términos

ABI (*Application Binary Interface*) Conjunto de convenciones de bajo nivel (paso de parámetros, uso de registros, alineación de datos) que rigen la interoperabilidad entre código compilado para una misma arquitectura.

Agnosticismo interarquitectura Grado en que una representación intermedia produce resultados estructuralmente semejantes al elevar un mismo programa compilado para arquitecturas distintas. Bloque de métricas central de este trabajo, evaluado mediante similitud de Jaccard ponderada, divergencia de Jensen-Shannon, coeficiente de variación del tamaño de la IR y delta de complejidad ciclomática.

Análisis estático Paradigma de análisis de software que examina un programa sin ejecutarlo, en contraposición al análisis dinámico. Este trabajo se enmarca íntegramente en este paradigma, obteniendo las seis representaciones intermedias estudiadas mediante técnicas de análisis estático sobre binarios en formato ELF.

Arquitectura Harvard Modelo de diseño de procesador en el que la memoria de instrucciones y la memoria de datos están físicamente separadas, con espacios de direcciones y buses independientes, en contraste con la arquitectura Von Neumann, de memoria unificada.

AST (*Abstract Syntax Tree*) Árbol de sintaxis abstracta. Estructura jerárquica que representa la composición recursiva de una expresión o instrucción, empleada por las representaciones intermedias de mayor nivel de abstracción (BNIL, Hex-Rays Microcode) frente a las de estructura plana.

Backend En el contexto de este trabajo, cada una de las ocho combinaciones de herramienta y nivel de representación empleadas para obtener una IR a partir de un binario (por ejemplo, `pyghidra` para P-code, o cada una de las tres capas de `Binary Ninja` para BNIL).

BinKit Conjunto de binarios multi-arquitectura, multi-compilador y multi-optimización de uso extendido en la literatura de análisis binario, del que procede el corpus de utilidades empleado en este trabajo.

Bloque básico Secuencia máxima de instrucciones consecutivas sin saltos ni puntos de entrada intermedios, salvo en la primera y la última. Constituye la unidad fundamental de nodos del grafo de flujo de control.

BNIL (*Binary Ninja Intermediate Language*) Familia de representaciones intermedias de **Binary Ninja**, organizada en tres capas de abstracción progresiva: *Low Level IL* (LLIL), *Medium Level IL* (MLIL) y *High Level IL* (HLIL).

CFG (*Control Flow Graph*) Grafo de flujo de control. Representación dirigida de los bloques básicos de una función y las transiciones de control entre ellos.

CISC (*Complex Instruction Set Computer*) Filosofía de diseño de ISA caracterizada por instrucciones de longitud variable y modos de direccionamiento complejos, en contraste con RISC. x86 es la ISA CISC considerada en este trabajo.

Complejidad ciclomática Métrica de complejidad estructural de un grafo de flujo de control, definida como $M = E - N + 2$, donde E es el número de aristas y N el número de nodos del grafo.

Demangling Proceso de recuperar el nombre original de una función o símbolo a partir de su forma codificada (*mangled*) por el compilador. No empleado en el emparejamiento de funciones del bloque de agnosticismo de este trabajo, que se realiza por coincidencia exacta de nombre.

Endianness Orden en que se almacenan los bytes de un valor multi-byte en memoria. *Little-endian* almacena el byte menos significativo primero; *big-endian*, el más significativo.

ESIL (*Evaluable Strings Intermediate Language*) Representación intermedia de bajo nivel empleada por **radare2**, basada en un intérprete de pila con notación postfija.

Fracción de escape Métrica de robustez que mide la proporción de operaciones de una IR que recurren a mecanismos de “escape” genéricos (llamadas a funciones auxiliares opacas) en lugar de a construcciones propias de la representación.

Fidelidad semántica Grado en que una representación intermedia preserva el comportamiento real de la instrucción nativa de la que procede. Distinta de

la robustez operativa, ya que una función puede elevarse sin error y, aun así, contener una traducción semánticamente incorrecta.

GVN (*Global Value Numbering*) Técnica de optimización de compiladores que identifica expresiones equivalentes en un programa, empleada por Hex-Rays Microcode en sus niveles de madurez superiores.

Guest state Bloque de memoria especial en el que VEX IR representa los registros de la arquitectura nativa como *offsets* numéricos.

IQR (*Interquartile Range*) Rango intercuartílico. Medida de dispersión estadística definida como la diferencia entre el tercer y el primer cuartil de una distribución.

IR (*Intermediate Representation*) Representación intermedia. Lenguaje formal situado a un nivel de abstracción intermedio entre el código máquina nativo y un lenguaje de alto nivel, empleado por una herramienta de análisis o compilación.

IRSB (*IR Super Block*) Unidad básica de traducción de VEX IR, correspondiente a un superbloque que agrupa entre una y varias decenas de instrucciones nativas.

ISA (*Instruction Set Architecture*) Arquitectura de conjunto de instrucciones. Especificación del repertorio de instrucciones, registros y modelo de memoria de un procesador.

Jaccard ponderado (índice de Ruzicka) Medida de similitud entre dos histogramas, calculada como el cociente entre la suma de los mínimos y la suma de los máximos de cada categoría presente en ambos.

Lifter Componente encargado de traducir el código máquina nativo de una arquitectura concreta a una representación intermedia.

Lifting Proceso de traducción de código máquina nativo a una representación intermedia.

LLVM IR Representación intermedia central del proyecto LLVM, definida en forma SSA estricta, empleada en este trabajo a través del decompilador **RetDec**.

MemorySSA Extensión de la forma SSA que aplica la propiedad de asignación única a los accesos a memoria, además de a las variables del programa. En LLVM IR, permite representar y analizar las dependencias entre dichos accesos.

Microcode Representación intermedia interna del decompilador Hex-Rays de IDA Pro, generada en ocho niveles de madurez progresivos.

Microoperaciones Operaciones atómicas y primitivas en las que P-code descompone cada instrucción nativa (por ejemplo, COPY, LOAD, STORE, BRANCH), con nombre propio y semántica bien definida.

Nodos ϕ Nodos especiales insertados en los puntos de confluencia de un grafo de flujo de control durante la construcción de la forma SSA. Su función es seleccionar cuál de las definiciones previas de una variable es la vigente según el camino de ejecución seguido hasta ese punto.

P-code Representación intermedia empleada por Ghidra, generada mediante SLEIGH, un lenguaje de especificación de procesadores. Se distingue entre *low P-code* (la traducción directa desde código nativo, sin análisis posterior) y *high P-code* (la versión obtenida tras el proceso de decompilación de Ghidra, con forma SSA ya construida).

Pipeline Conjunto de *scripts* desarrollados en este trabajo para automatizar la extracción de las seis representaciones intermedias y el cálculo de las métricas cuantitativas sobre el corpus de binarios.

PyVEX Biblioteca de Python, desarrollada por el proyecto **angr**, que traduce código máquina nativo a VEX IR y proporciona acceso programático a la representación resultante.

Ratio de expansión Métrica de verbosidad que mide el cociente entre el número de operaciones (u operaciones de árbol de sintaxis abstracta) de la IR resultante y el número de instrucciones nativas de las que procede.

RISC (*Reduced Instruction Set Computer*) Filosofía de diseño de ISA caracterizada por instrucciones de longitud fija y una arquitectura *load-store*, en contraste con CISC. ARM y MIPS son las ISA RISC consideradas en este trabajo.

RSS (*Resident Set Size*) Memoria residente. Cantidad de memoria física que un proceso mantiene ocupada en un instante dado, distinta de la memoria virtual reservada.

Sintaxis Forth Estilo de notación en el que los operandos preceden al operador (notación polaca inversa) y las operaciones manipulan directamente una pila, en lugar de expresarse mediante la sintaxis infija habitual. Empleada por ESIL para representar la semántica de cada instrucción nativa.

SSA (*Static Single Assignment*) Forma estática de asignación única. Propiedad de una representación intermedia en la que cada variable se asigna exactamente una vez en el código.

Timeout Estado de una ejecución que excede el tiempo límite establecido, registrado de forma diferenciada de una ejecución completada o fallida.

Variables temporales Variables introducidas por un *lifter* para nombrar valores intermedios de un cómputo que en la instrucción nativa no requerían una ubicación propia, empleadas como medida de verbosidad de una representación intermedia.

Varnode Unidad básica de representación de P-code, definida como una tripleta de espacio de direcciones, *offset* y tamaño.

VEX IR Representación intermedia originalmente desarrollada para el instrumentador dinámico **Valgrind**, reutilizada para análisis estático a través de **PyVEX** en el entorno de **angr**.

Capítulo 1

Introducción

1.1. Motivación

El análisis estático de binarios constituye una pieza fundamental en disciplinas como la ingeniería inversa, el análisis de vulnerabilidades, la detección de código dañino y la verificación de software cuando no se dispone del código fuente. La mayoría de las herramientas modernas de análisis de binarios no operan directamente sobre el código máquina nativo, sino que lo traducen previamente a una representación intermedia (IR, del inglés *Intermediate Representation*), un lenguaje abstracto e independiente de la arquitectura que facilita la aplicación de técnicas de análisis de software, como la reconstrucción del flujo de control, la ejecución simbólica o el análisis de flujo de datos, sin necesidad de implementarlas de nuevo para cada arquitectura o conjunto de instrucciones [2].

Sin embargo, cada entorno de análisis binario ha diseñado su propia representación intermedia, siguiendo filosofías y niveles de abstracción muy distintos entre sí: desde representaciones de bajo nivel que preservan de forma casi literal la semántica de la instrucción nativa, hasta representaciones de alto nivel que reconstruyen estructuras de control y variables con una fidelidad cercana a la de un lenguaje de programación de alto nivel [3]. Esta heterogeneidad tiene consecuencias prácticas directas: la elección de una herramienta de análisis y, por tanto, de su IR subyacente, condiciona qué tipo de análisis es viable, qué coste computacional conlleva, cuán fiable resulta frente a binarios reales (como binarios optimizados, sin símbolos y de arquitecturas diversas) y hasta qué punto sus resultados son comparables entre distintas arquitecturas.

A pesar de la relevancia de esta elección, la literatura existente aborda las represen-

taciones intermedias de forma parcial: o bien se limita a describir una IR concreta en el contexto de la herramienta que la implementa, o bien compara un subconjunto reducido de ellas bajo un criterio único, normalmente cualitativo. No existe, en la literatura consultada, un estudio que compare de forma sistemática y bajo un mismo marco metodológico, tanto cualitativo como cuantitativo, las representaciones intermedias de uso extendido en el análisis estático de binarios (concretamente, P-code, BNIL, VEX IR, ESIL, LLVM IR y Hex-Rays Microcode) [4–6]. Este Trabajo de Fin de Máster nace para cubrir ese hueco, proporcionando a analistas e investigadores un criterio informado y empíricamente fundamentado a la hora de seleccionar o combinar estas representaciones según las necesidades de su análisis.

1.2. Objetivos

El objetivo general de este trabajo consiste en realizar una comparación exhaustiva, tanto cualitativa como cuantitativa, de las seis representaciones intermedias seleccionadas, con el fin de identificar sus fortalezas, limitaciones y ámbitos de aplicación más adecuados.

Este objetivo general se concreta en los siguientes objetivos específicos:

- Elaborar una comparativa cualitativa de las IR consideradas, atendiendo a características como el nivel de abstracción, el modelo de memoria, la expresividad semántica o la disponibilidad de forma SSA, y contrastar la información con fuentes primarias y la documentación oficial de cada herramienta.
- Diseñar un marco de evaluación cuantitativo que permita comparar objetivamente las IR consideradas conforme a cuatro bloques de métricas: verbosidad y estructura, coste computacional, robustez frente a binarios reales y agnosticismo o invarianza interarquitectura.
- Discutir e integrar los resultados cualitativos y cuantitativos obtenidos sobre un corpus de binarios, con el fin de identificar qué representación intermedia resulta más adecuada según distintos escenarios de uso.

1.3. Estructura del trabajo

El presente documento se organiza tal y como se indica a continuación. El Capítulo 2 presenta el contexto necesario para el estudio: el análisis estático de binarios y el concepto de representación intermedia. El Capítulo 3 revisa los trabajos previos de comparación de IR y delimita el hueco que este TFM pretende cubrir. El Capítulo 4

desarrolla la comparativa cualitativa de las seis representaciones intermedias consideradas: arquitecturas y herramientas seleccionadas, dimensiones de comparación, y la tabla comparativa resultante. El Capítulo 5 describe la metodología cuantitativa utilizada: el *pipeline* de extracción, el corpus de 960 binarios, y los cuatro bloques de métricas definidos. El Capítulo 6 presenta y discute los resultados obtenidos. El Capítulo 7 cierra el documento con las conclusiones y las líneas de trabajo futuro. Completa el trabajo el Anexo A, con el desglose de tareas y el diagrama de Gantt del proyecto.

Capítulo 2

Conocimientos previos

En este capítulo se incluyen los conceptos necesarios para la comprensión del resto del trabajo. En primer lugar, se ofrece una visión general del análisis estático de binarios. A continuación, se describen los aspectos más relevantes de las representaciones intermedias en el contexto de dicho análisis.

2.1. Análisis estático de binarios

El análisis de software se divide en dos grandes paradigmas: el análisis dinámico y el análisis estático. Ambos persiguen el objetivo de comprender el comportamiento de un programa, pero difieren en la forma de obtener esa información. El análisis dinámico la obtiene observando el programa durante su ejecución, mediante técnicas como la instrumentación, la depuración, el *fuzzing* o la ejecución simbólica guiada por entradas concretas. Su principal ventaja es que lo observado ocurrió realmente; su principal limitación es que solo cubre los caminos de ejecución recorridos por las entradas empleadas. El análisis estático, por el contrario, examina el programa sin ejecutarlo y razona directamente sobre su representación, ya sea el código fuente o, en su ausencia, el código máquina compilado. Su objetivo es cubrir, al menos en principio, la totalidad de los caminos de ejecución posibles, a costa de una menor precisión frente a comportamientos que solo se manifiestan en tiempo de ejecución, como la resolución dinámica de símbolos o el código auto-modificable.

Este trabajo se sitúa exclusivamente en el paradigma estático y, más concretamente, en el análisis estático de binarios. En este contexto, el programa se estudia a partir de ficheros ejecutables ya compilados y sin acceso al código fuente. Las representaciones intermedias analizadas en este trabajo se obtienen directamente a partir de binarios en formato ELF, sin ejecutar en ningún momento el programa bajo estudio.

El análisis estático de binarios presenta ventajas relevantes en escenarios como el análisis forense o el estudio de código dañino. No requiere un entorno de ejecución controlado, permite en principio cubrir la totalidad de los caminos de ejecución y resulta más seguro al no ejecutar el binario analizado. Como contrapartida, debe enfrentarse a limitaciones intrínsecas del código máquina, entre ellas la ausencia de símbolos de depuración y de información de tipos, la ofuscación del flujo de control o la ambigüedad entre código y datos, especialmente en arquitecturas como x86-64. En la práctica, el desensamblado estático preciso sigue siendo un problema abierto. La recuperación de información estructural, como los límites de función, resulta especialmente difícil en binarios reales. Esto se debe, en parte, a que los trabajos previos suelen evaluarse en condiciones simplificadas y rara vez abordan en detalle el tratamiento de errores necesario para estas tareas [7]. Esta dificultad muestra que problemas como la ambigüedad entre código y datos o la reconstrucción de límites de función no son casos marginales de laboratorio, sino una fuente real de divergencia entre herramientas de análisis binario.

Además, el análisis de código fuente dispone de información que puede perderse parcial o totalmente tras la compilación, como nombres de variables y funciones, información de tipos, estructura léxica del programa o comentarios. El compilador traduce estos elementos a una secuencia lineal de instrucciones máquina y puede aplicar transformaciones que reordenan, fusionan o eliminan construcciones del programa original, sobre todo en niveles altos de optimización.

Por ello, el análisis estático de binarios debe reconstruir parte de esa información a partir de la secuencia de bytes del ejecutable. En ese proceso de reconstrucción suelen distinguirse varias tareas. El desensamblado traduce la secuencia de bytes del binario a instrucciones máquina legibles. Esta tarea suele abordarse mediante dos estrategias principales: por barrido lineal, que analiza el flujo de bytes de forma secuencial, y descendente recursivo, que sigue el flujo de control a partir de puntos de entrada conocidos y trata de evitar la confusión entre código y datos. La primera favorece la cobertura, mientras que la segunda suele ofrecer mejores resultados cuando el binario contiene datos embebidos o saltos de control explícitos [8,9].

La reconstrucción de límites de función consiste en identificar dónde comienza y termina cada función del binario. Se trata de una tarea especialmente difícil en ausencia de símbolos de depuración, y su precisión depende en gran medida de las heurísticas empleadas por cada herramienta [2,9].

Por último, la reconstrucción del grafo de flujo de control (CFG) construye, a partir de las funciones y bloques básicos ya identificados, el grafo dirigido de transiciones de control entre ellos. Este paso constituye la base de numerosas técnicas de análi-

sis posteriores y resulta de interés para métricas estructurales como la complejidad ciclomática. La etapa final de este flujo es la elevación (o *lifting*, en inglés) a una representación intermedia, es decir, la traducción de instrucciones nativas dependientes de la arquitectura a una IR semánticamente equivalente e independiente de ella, con el fin de facilitar análisis posteriores ya mencionados, como la identificación de patrones o la reconstrucción del flujo de datos [10, 11].

2.2. Representaciones intermedias en el análisis de software

Una representación intermedia es un lenguaje formal situado a un nivel de abstracción intermedio entre el código máquina nativo de una arquitectura concreta y un lenguaje de programación de alto nivel, empleado internamente por una herramienta de análisis o compilación para razonar sobre el comportamiento de un programa de forma independiente de los detalles específicos de la arquitectura en la que dicho programa fue compilado.

El uso de una IR no es exclusivo del análisis de binarios: es un concepto tomado directamente del diseño de compiladores, donde tradicionalmente se organiza en una arquitectura de tres fases: *frontend*, IR y *backend*, que desacopla el lenguaje de entrada de la arquitectura de salida, de forma que compilar N lenguajes hacia M arquitecturas requiere solo N *frontends* y M *backends*, en lugar de $N \times M$ compiladores completos [12]. Este fenómeno se ilustra en las Figuras 2.1 y 2.2.

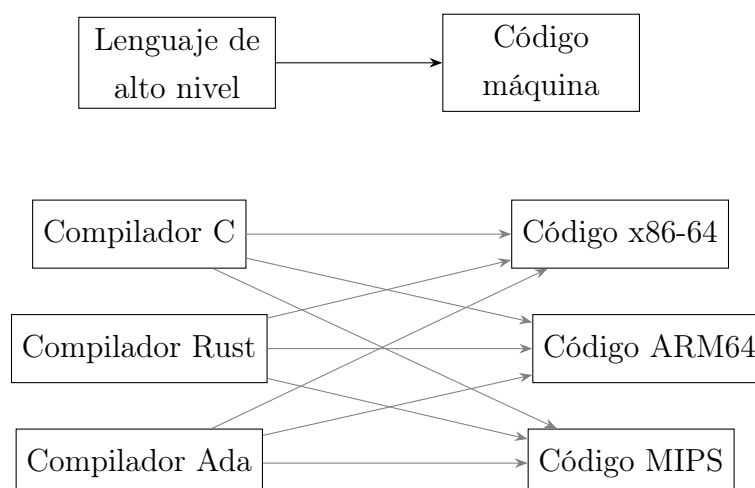


Figura 2.1: Esquema directo *frontend* a *backend*: cada compilador debe generar código para cada arquitectura de destino ($N \times M$ compiladores); inspirada en [1]

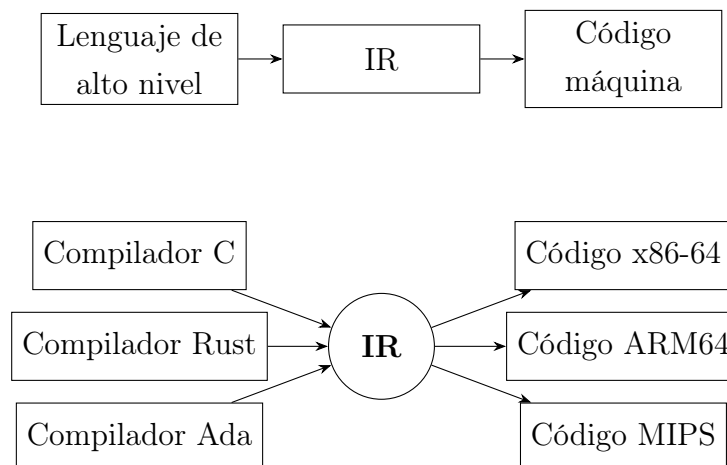


Figura 2.2: Esquema con IR común como *middle-end*: el mismo lenguaje intermedio conecta todos los *frontends* con todos los *backends* ($N + M$ componentes); inspirada en [1]

En el contexto del análisis binario, el papel de esta arquitectura se invierte: el *frontend* o el *lifter* traduce el código máquina nativo de una arquitectura concreta a la IR, y es sobre esta IR donde se implementan los algoritmos de análisis, consiguiendo que dichos algoritmos se escriban una única vez y funcionen para cualquier arquitectura soportada por el *lifter* correspondiente. En [13] se ilustra directamente esta motivación en el diseño de BAP, uno de los primeros marcos de análisis binario contruidos explícitamente alrededor de un lenguaje intermedio (en inglés, *intermediate language* o IL) propio: representar de forma explícita todos los efectos secundarios de cada instrucción nativa es precisamente lo que hace viable un análisis dirigido por sintaxis sobre cientos de miles de instrucciones ensambladas. Las representaciones intermedias empleadas en herramientas de análisis binario difieren notablemente entre sí en varias dimensiones de diseño, como por ejemplo:

- **Nivel de abstracción:** desde IR de bajo nivel, que preservan una correspondencia casi uno a uno con la instrucción nativa, hasta IR de alto nivel, que reconstruyen estructuras de control y variables con una fidelidad cercana a un lenguaje de alto nivel.
- **Modelo de memoria:** cómo se representan los accesos a memoria, los registros y las banderas de estado del procesador.
- **Forma SSA (*Static Single Assignment*):** una propiedad de diseño, tomada también de la teoría de compiladores, en la que cada variable o valor temporal se asigna exactamente una vez en el código [14], lo que simplifica notablemente ciertos análisis de flujo de datos. No todas las IR consideradas

en este trabajo la implementan, y entre las que lo hacen, no todas la mantienen a lo largo de todos sus niveles de abstracción.

- **Expresividad semántica:** la capacidad de la IR para representar de forma explícita construcciones complejas del conjunto de instrucciones nativo, frente a la necesidad de recurrir a mecanismos de “escape” genéricos como llamadas a funciones auxiliares cuando la IR no dispone de una construcción propia equivalente.

Estas dimensiones no son meras características descriptivas: condicionan directamente los resultados esperables en los cuatro bloques de métricas cuantitativas que se definen posteriormente (verbosidad y estructura, coste, robustez y agnosticismo interarquitectura).

Estas diferencias de diseño entre *lifters* tienen, además, consecuencias prácticas ya documentadas. En [10] se muestra que estas diferencias condicionan de forma medible el comportamiento de las aplicaciones que consumen la IR resultante, tales como la decompilación, la detección de similitud de código o el análisis de discriminabilidad. Por su parte, en [11] se constata que incluso *lifters* maduros y ampliamente utilizados incurren en errores semánticos no triviales al traducir a su propia IR.

Capítulo 3

Estado del arte

Este capítulo revisa los trabajos previos relacionados con la comparación de representaciones intermedias en el análisis de binarios, tanto dentro de este dominio como en ámbitos afines de compilación, y cierra identificando las carencias de la literatura.

3.1. Comparación de las IR

La comparación de representaciones intermedias ha recibido atención en la literatura de compiladores desde hace décadas. Stanier y Watson realizaron una revisión sistemática de las IR empleadas en compiladores imperativos, en la que propusieron una taxonomía de diseños y destacaron que la elección de una IR condiciona la forma de razonar sobre un programa, así como las optimizaciones y transformaciones que pueden aplicarse sobre él [3]. Este trabajo constituye un antecedente metodológico importante, ya que muestra que la comparación de IR no es un ejercicio meramente descriptivo, sino una manera de estudiar propiedades estructurales que afectan al comportamiento del sistema completo.

Aunque el foco de este trabajo se sitúa en el análisis estático de binarios, conviene señalar que la comparación de IR también ha recibido atención en otros dominios de compilación. En particular, Cardama et al. revisan la evolución de las representaciones intermedias en computación cuántica y subrayan la necesidad de capas de abstracción que faciliten la portabilidad y la optimización [1]. Este precedente refuerza la idea de que la evaluación sistemática de IR constituye un problema transversal, aunque los objetivos y restricciones concretas difieran del ámbito binario.

En el ámbito del análisis de binarios, la adopción de IR responde a una necesidad similar, pero con restricciones adicionales derivadas de la ausencia de código fuente

y de la necesidad de reconstruir la semántica a partir de instrucciones máquina. BAP [13] fue uno de los primeros marcos en formalizar esta idea de forma explícita: el sistema eleva el código máquina a un IR con semántica definida, expone de manera explícita los efectos secundarios de cada instrucción y permite ejecutar análisis dirigidos por sintaxis sobre binarios. Este enfoque hizo visible una idea que hoy es central en el análisis binario: la IR no es solo una representación auxiliar, sino el punto donde se concentra la semántica necesaria para el análisis posterior.

Trabajos sucesivos profundizaron en esa misma dirección desde perspectivas distintas. B2R2 [15] se diseñó como un *frontend* eficiente para el análisis de binarios, poniendo el foco en el coste de la traducción y en la velocidad con la que un binario puede traducirse a una representación intermedia utilizable. De forma complementaria, otros trabajos propusieron IR propias con distintas prioridades de diseño, como la expresividad semántica [13], la formalización de la semántica [16,17] o la facilidad para realizar análisis posteriores [15]. Sin embargo, la mayoría de estas propuestas presentan una IR concreta o un marco de análisis específico, pero no comparan de forma sistemática varias representaciones entre sí.

Los primeros estudios comparativos aparecen cuando la atención pasa de la mera construcción de una IR a la evaluación de sus propiedades. Kim et al. propusieron en [11] un análisis sistemático de las representaciones intermedias generadas por varios *lifters* y mostraron que la corrección semántica de estas representaciones puede evaluarse mediante pruebas específicas, encontrando además errores no triviales en herramientas ampliamente utilizadas. Este trabajo es especialmente relevante porque establece que la calidad de una IR no puede darse por supuesta, sino que debe verificarse empíricamente.

Más recientemente, Liu et al. [10] ampliaron esta perspectiva y demostraron que las diferencias de diseño entre *lifters* tienen consecuencias medibles en aplicaciones posteriores, como la decompilación, la discriminabilidad de código y el análisis de punteros. Su estudio es importante porque desplaza el foco desde la definición de la IR hacia su impacto práctico: dos IR que parecen equivalentes desde el punto de vista formal pueden producir resultados muy distintos cuando se usan como entrada para tareas reales de seguridad y análisis. En la misma línea, trabajos como [18] reforzaron la idea de que la fidelidad semántica de las IR debe comprobarse con técnicas específicas y no asumirse como una propiedad garantizada.

En conjunto, la literatura existente demuestra que las IR desempeñan un papel decisivo en el análisis de binarios, pero también que su comparación sistemática sigue siendo limitada. La mayor parte de los trabajos se centra en una única IR, en una propiedad concreta o en una aplicación posterior. El precedente más cercano en

términos de comparación empírica [4] ilustra bien el alcance de esta limitación: su comparación cubre solo tres de las seis IR consideradas en este TFM y se restringe a una única aplicación centrada en la clasificación de *malware* mediante *n*-gramas de códigos de instrucción ensamblador. Existen, por tanto, estudios valiosos sobre diseño, validación e impacto práctico de representaciones intermedias, pero no una comparación unificada que combine criterios cualitativos y cuantitativos sobre varias IR ampliamente utilizadas bajo un mismo marco metodológico. Esa es precisamente la laguna que aborda este trabajo.

3.2. Carencias identificadas

Como se ha visto, existen tres carencias específicas de la literatura existente, cada una de las cuales motiva directamente una decisión de diseño de este TFM.

Carencia de amplitud. Ningún trabajo localizado compara conjuntamente las seis representaciones intermedias consideradas en este TFM. Cannan y Morris cubren tres de ellas (LLIL, P-code y Hex-Rays Microcode) [4]; el resto de precedentes, como BAP, B2R2, los estudios de Kim et al. y Liu et al., se centran en una única IR o en un único *lifter* [10, 11, 13, 15]. Esta limitación no es meramente cuantitativa: al no incluir de forma conjunta P-code, BNIL, VEX IR, ESIL, LLVM IR y Hex-Rays Microcode, ningún trabajo previo permite determinar si los patrones observados en un subconjunto de varias IR se generalizan al resto ni comparar directamente representaciones de diseño muy distinto entre sí, desde una IR de bajo nivel como ESIL hasta una de mayor nivel de abstracción como el Microcode de Hex-Rays en su forma más elevada.

Carencia de generalidad en la comparación cuantitativa. El único precedente que compara empíricamente varias IR [4] centra su análisis en una única aplicación enfocada en tareas subsiguientes al *lifting*: el rendimiento de un clasificador de *malware* basado en *n*-gramas de códigos de operación extraídos de instrucciones de ensamblador. Esto deja sin resolver una cuestión distinta y, para los propósitos de este trabajo, más fundamental: cómo se comparan las IR en sus propiedades intrínsecas (verbosidad, coste computacional, robustez frente a binarios reales y agnosticismo interarquitectura), con independencia de la tarea concreta en la que se vayan a emplear después. Un resultado favorable a una IR en clasificación de *malware* no permite inferir, por ejemplo, si esa misma IR generaliza mejor entre arquitecturas o si su coste de generación resulta asumible en un *pipeline* de análisis a gran escala.

Los cuatro bloques de métricas definidos en el Capítulo 5 responden precisamente a esta carencia.

Carencia de integración metodológica. Ningún precedente combina una comparación cualitativa de las dimensiones de diseño de las IR, nivel de abstracción, modelo de memoria, forma SSA y expresividad semántica, con una comparación cuantitativa bajo un mismo marco metodológico. Los trabajos de naturaleza más cualitativa (Stanier y Watson en compiladores clásicos [3], o las descripciones de diseño de BAP y B2R2 [13, 15]) no incorporan una evaluación empírica sistemática. Los trabajos cuantitativos [4, 10, 11, 18] no relacionan sus hallazgos con las dimensiones de diseño que podrían explicarlos. Este TFM aborda ambas perspectivas de forma explícitamente conectada: una comparativa cualitativa que identifica las dimensiones de diseño de cada IR, y un marco cuantitativo que contrasta empíricamente si dichas dimensiones se traducen en diferencias medibles, por ejemplo, en la verbosidad estructural o en el coste de generación.

En conjunto, estas tres carencias delimitan con precisión la contribución de este trabajo: una comparación de seis IR de uso extendido en el análisis estático de binarios [4–6], evaluadas en sus propiedades intrínsecas mediante un marco de métricas independiente de cualquier aplicación derivada, y presentada de forma que la comparativa cualitativa y la cuantitativa se retroalimenten bajo un único marco metodológico coherente.

Capítulo 4

Comparativa cualitativa

En este capítulo se presenta la comparativa cualitativa de las seis IR estudiadas en este TFM. Esta comparativa constituye la primera de las dos aportaciones originales del trabajo. En primer lugar, se describen las arquitecturas de conjunto de instrucciones incluidas en el corpus de evaluación. A continuación, se presentan las herramientas seleccionadas para obtener cada representación intermedia, así como los criterios aplicados durante su selección y descarte. Después, se definen las dimensiones de comparación. Por último, se presenta la tabla comparativa y se analizan sus resultados por dimensión.

4.1. Arquitecturas de conjunto de instrucciones consideradas

La selección de las arquitecturas de conjunto de instrucciones (del inglés, *Instruction Set Architecture* o ISA) determina el alcance de la evaluación interarquitectura. Sobre estas arquitecturas se generan los binarios empleados para medir la invarianza de las seis representaciones intermedias.

En este trabajo, se consideran tres familias de ISA: x86, ARM y MIPS. Para MIPS se incluyen configuraciones *little-endian* y *big-endian*, identificadas como MIPS y MIPSEB, respectivamente. Cada configuración se evalúa en sus variantes de 32 y 64 bits. De este modo, se obtienen ocho configuraciones experimentales: x86-32, x86-64, ARM-32, ARM-64, MIPS-32, MIPS-64, MIPSEB-32 y MIPSEB-64.

La selección se fundamenta en tres criterios:

- **Diversidad de diseño.** Se incluyen una ISA CISC y dos ISA RISC. x86 presenta instrucciones de longitud variable, modos de direccionamiento complejos

y una semántica de instrucción potencialmente compuesta. Por su parte, ARM y MIPS se basan principalmente en un modelo RISC, con instrucciones más regulares y una separación entre accesos a memoria y operaciones de cómputo, propia de las arquitecturas *load-store* [19]. Esta diversidad permite observar cómo se adapta cada *lifter* a instrucciones con distintos niveles de complejidad semántica.

- **Cobertura de variantes de 32 y 64 bits.** Se incluyen configuraciones de 32 y 64 bits para cada familia. Así, se puede distinguir si las diferencias observadas se asocian a la familia de ISA o a la variante concreta evaluada. Esta distinción resulta especialmente relevante en ARM: AArch32 y AArch64 corresponden a estados de ejecución diferentes y emplean conjuntos de instrucciones, modelos de registros y convenciones de llamada distintos [19]. Una situación comparable se produce entre i386 (x86-32) y x86-64.
- **Control del orden de bytes.** MIPS y MIPSEB difieren únicamente en el orden de bytes. Este par permite evaluar si las métricas de invarianza reflejan diferencias asociadas a la ISA o variaciones introducidas por el orden de bytes. Como hipótesis de control, se espera una divergencia reducida entre MIPS y MIPSEB, y una divergencia mayor entre configuraciones de familias distintas, como x86 y ARM.

La inclusión de MIPS en variantes *little-endian* y *big-endian* sigue la estrategia de BinKit [20], que incluye ambas configuraciones para estudiar el efecto del orden de bytes en el análisis de código binario. El corpus utilizado en este trabajo fue compilado para las ocho configuraciones indicadas.

4.2. Herramientas y entornos seleccionados

Para cada una de las seis representaciones intermedias consideradas se selecciona la herramienta que permite obtenerla de forma reproducible sobre el corpus definido en este trabajo. La selección se fundamenta en la cobertura de arquitecturas, la disponibilidad de una interfaz programática, el mantenimiento del proyecto y la posibilidad de integrar la herramienta en el *pipeline* automatizado de evaluación. La descripción técnica de los procesos de elevación empleados por cada herramienta se presenta más adelante. En esta sección se justifican únicamente las decisiones de selección y descarte.

P-code — Ghidra. Para obtener P-code se selecciona **Ghidra**, la herramienta de ingeniería inversa de código abierto desarrollado por la NSA y distribuida bajo licencia Apache 2.0 desde su publicación pública en 2019 [21]. **Ghidra** utiliza SLEIGH, un lenguaje de especificación de procesadores que define la traducción de instrucciones máquina a P-code. P-code es la representación intermedia nativa de **Ghidra** y no dispone de implementaciones alternativas con relevancia comparable para el análisis estático. Por tanto, la selección de P-code determina directamente la selección de **Ghidra**.

BNIL — Binary Ninja. Se selecciona **Binary Ninja** para obtener BNIL, la plataforma comercial de Vector 35, cuya IR se expone a través de una API abierta bajo licencia MIT, independiente de la licencia comercial de la propia herramienta [22]. La plataforma expone una familia de representaciones intermedias con varios niveles de abstracción: *Low Level IL* (LLIL), *Medium Level IL* (MLIL) y *High Level IL* (HLIL). LLIL conserva una semántica próxima a las instrucciones nativas, MLIL introduce variables, tipos e información de flujo de datos y HLIL incorpora construcciones de control de mayor nivel y simplificaciones adicionales. Esta jerarquía permite evaluar el efecto del nivel de abstracción manteniendo constante el *lifter* y la herramienta de análisis.

VEX IR — angr. Se selecciona **angr** para obtener VEX IR. VEX nació como parte de **Valgrind**, el instrumentador dinámico distribuido bajo GPLv2 [23], y fue posteriormente adoptada para análisis puramente estático por **angr**, a través de **PyVEX**, una biblioteca de Python que proporciona una interfaz para traducir instrucciones nativas a VEX IR y manipular sus representaciones desde este lenguaje. **angr** integra VEX como representación intermedia para el análisis de binarios y proporciona una API programática adecuada para automatizar la extracción de su IR. VEX IR abstrae el código máquina en una representación independiente de la arquitectura, diseñada para facilitar el análisis de programas. La selección de **angr** permite además utilizar una herramienta ampliamente empleada en investigación sobre análisis de binarios.

ESIL — radare2. Se selecciona **radare2** para obtener *Evaluable Strings Intermediate Language* (ESIL). **radare2** distribuye su núcleo bajo LGPLv3, con licencias variables para sus numerosos complementos y extensiones [24]. ESIL es la representación intermedia propia de **radare2** y permite expresar la semántica de las instrucciones mediante una notación basada en expresiones evaluables. En la taxonomía

adoptada en este trabajo, ESIL se considera una representación de bajo nivel por su proximidad a la semántica de las instrucciones nativas. Dado que ESIL es una IR propia de **radare2**, la elección de la representación determina directamente la de la herramienta.

Hex-Rays Microcode — IDA Pro. Se selecciona IDA Pro con el decompilador Hex-Rays para obtener Microcode. IDA Pro es la herramienta comercial de referencia del sector desde los años noventa, cuyo modelo de licenciamiento migró de la venta perpetua tradicional a un modelo de suscripción en septiembre de 2024 [25]. Esta representación intermedia interna fue descrita públicamente por Ilfak Guilfanov en 2018 [26]. La herramienta proporciona una API de *scripting* que permite acceder al Microcode de forma programática desde IDA 7.1. La elección de Microcode determina, por tanto, el uso de IDA Pro y del decompilador Hex-Rays.

LLVM IR — RetDec. LLVM IR puede obtenerse mediante diversos *lifters*. LLVM IR constituye la representación central del proyecto LLVM, distribuido bajo Apache 2.0 con una excepción específica que permite enlazar con código propietario sin obligaciones de *copyleft* [27], y a diferencia de las cinco IR anteriores, no fue concebida originalmente para el análisis de binarios, sino para su compilación. Por ello, se realiza una selección explícita entre las alternativas disponibles para obtenerla a partir de binarios. Se selecciona RetDec por su disponibilidad como herramienta de línea de comandos, su integración directa en un *pipeline* automatizado y su cobertura de las arquitecturas incluidas en el corpus, salvo MIPS-64 y MIPSEB-64 [28]. En consecuencia, RetDec no completa las ejecuciones correspondientes a estas dos configuraciones, por lo que no genera LLVM IR y sus resultados aparecen como no disponibles en las métricas descritas posteriormente.

Se consideran también las siguientes alternativas:

- **rev.ng** ofrece una cobertura amplia mediante componentes basados en QEMU y dispone de una interfaz de línea de comandos. Sin embargo, se descarta porque el soporte de arquitecturas presenta distintos niveles de madurez: la documentación del proyecto indica que la mayor parte de las pruebas se concentra en x86-64 y que ARM y MIPS requieren validación adicional [29]. Además, el *roadmap* del proyecto mantiene como objetivo la ejecución estable del *pipeline* completo sobre conjuntos razonables de binarios x86-64 [30].
- **Remill** se ha empleado en trabajos previos sobre elevación binaria y LLVM IR [11]. No obstante, se descarta por no cubrir MIPS, MIPSEB y ARM de

32 bits en la configuración requerida. Existe un proyecto con soporte parcial para MIPS de 32 bits, pero no cubre el conjunto completo de arquitecturas evaluadas y no presenta mantenimiento reciente [31,32].

- **mcsema** depende de **Remill** y utiliza **IDA Pro** para apoyar la recuperación del flujo de control. Se descarta por esta dependencia y porque su repositorio oficial permanece archivado desde 2022 [33].
- **BinRec** se limita a binarios ELF para x86 de 32 bits y, por tanto, no cubre el corpus multi-arquitectura de este trabajo [34]. Además, no encaja en este proyecto, ya que se trata de una herramienta de elevación dinámica que traza la ejecución real del binario con entradas concretas y eleva a IR únicamente el código efectivamente ejecutado en dichas trazas.
- **reopt** se limita a binarios ELF para x86 de 64 bits. Además, se ha observado que puede reutilizar secciones del binario original cuando la elevación no puede completarse, lo que impide considerarlo una fuente homogénea de LLVM IR para esta comparación [35,36].
- **llvm-mctoll** no cubre ARM de 32 bits ni MIPS, por lo que no satisface los requisitos de cobertura definidos para el corpus [37].
- **McNinja** no eleva directamente código máquina a LLVM IR, sino que traduce MLIL de **Binary Ninja** a LLVM IR. Su adopción introduciría una dependencia directa entre los resultados de BNIL y LLVM IR, lo que comprometería la independencia entre ambas representaciones en la comparación [38].

4.3. Dimensiones de comparación

En esta sección se definen las dimensiones empleadas en la comparación cualitativa. Para cada dimensión se describe el criterio de evaluación y se justifica su relevancia en el análisis estático de binarios.

En primer lugar, se identifica la herramienta principal que genera, transforma o consume cada representación intermedia. Este campo tiene carácter descriptivo y permite contextualizar cada IR dentro del entorno técnico en el que se implementa y utiliza. Sus características dependen, en buena medida, de las decisiones de diseño adoptadas por la herramienta asociada.

El nivel de abstracción evalúa el grado en que la IR se separa del conjunto de instrucciones nativo y se aproxima a una semántica de mayor nivel. Se consideran tres niveles de referencia: alto, medio y bajo. También se admiten valores intermedios,

como medio-alto o bajo-medio, cuando una IR no se ajusta por completo a una única categoría. Una IR de nivel alto normaliza operaciones y representa de forma explícita elementos como los efectos secundarios, las banderas de estado y la memoria. Una IR de nivel medio abstrae parte de los detalles de la ISA, pero conserva construcciones dependientes de la arquitectura. Una IR de nivel bajo mantiene una correspondencia estrecha con las instrucciones nativas y puede considerarse una re-codificación estructurada de la ISA original. Un mayor nivel de abstracción puede facilitar la portabilidad del análisis y la aplicación de técnicas independientes de arquitectura.

La forma normal describe la organización formal utilizada para representar el flujo de datos y el flujo de control. Se evalúa si la IR emplea forma SSA, si esta se encuentra disponible de forma nativa, en capas superiores de abstracción o como transformación opcional. También se considera si la representación se organiza de forma lineal, mediante una máquina de pila, en torno a temporales o mediante árboles de expresión. Este criterio resulta relevante porque la estructura de la IR condiciona las técnicas de análisis que pueden aplicarse sobre ella. La forma SSA hace explícitas las relaciones de definición y uso de las variables y facilita análisis de flujo de datos, optimizaciones y análisis de dependencias [14]. También condiciona la implementación de técnicas como la ejecución simbólica o la construcción de grafos de dependencia.

El modelo de memoria describe cómo se representan los accesos a memoria, los registros y los estados auxiliares del procesador y otros elementos auxiliares del estado de ejecución. Se consideran aspectos como la existencia de espacios de direcciones diferenciados, la representación de los registros mediante posiciones dentro de un estado separado, su tratamiento como variables o valores independientes y el uso de operaciones explícitas de lectura y escritura, como **load** y **store**, para acceder a los distintos elementos del estado. En el análisis estático de binarios, esta dimensión condiciona la fidelidad con la que se representa el estado de ejecución del programa. Por tanto, puede afectar a la precisión de análisis de alias, recuperación de variables, seguimiento de dependencias y análisis de accesos a datos.

La granularidad de operaciones evalúa el nivel de descomposición utilizado durante la traducción de las instrucciones nativas. Se consideran representaciones basadas en microoperaciones, instrucciones de tres direcciones, expresiones anidadas, instrucciones agrupadas en bloques o distintas capas de abstracción. Una granularidad fina permite observar con precisión los efectos semánticos de cada instrucción. Una granularidad más gruesa puede reducir el tamaño de la representación y favorecer la eficiencia de análisis de mayor nivel. Esta dimensión se relaciona con las métricas de

verbosidad y coste definidas en el capítulo dedicado a la metodología cuantitativa. La expresividad evalúa la capacidad de la IR para representar la semántica de las instrucciones nativas. Se consideran cuatro aspectos: la representación explícita de efectos secundarios, el tratamiento de comportamientos indefinidos o parcialmente definidos por la arquitectura, la fidelidad ante accesos a memoria no alineados y la cobertura de construcciones específicas de arquitectura. También se analiza la necesidad de recurrir a mecanismos auxiliares para representar operaciones que no disponen de un equivalente directo en la IR. Estos mecanismos pueden adoptar la forma de operaciones especiales, llamadas a funciones auxiliares o construcciones opacas. Su uso puede indicar que una parte de la semántica nativa no se expresa directamente en la representación. Esta dimensión no se reduce a una escala cerrada de tres valores. Una IR puede ser expresiva para representar cómputo arbitrario, como ocurre habitualmente con las IR diseñadas para compiladores, sin preservar con la misma fidelidad los detalles propios de una ISA. Entre estos detalles se incluyen las banderas de estado del procesador, la semántica de instrucciones vectoriales o comportamientos indefinidos documentados por el fabricante.

Las transformaciones integradas indican si la IR incorpora, o permite aplicar, transformaciones que simplifican, normalizan o enriquecen el código para facilitar su análisis. Entre ellas se incluyen la propagación de constantes, la eliminación de código muerto, la conversión a forma SSA y la abstracción de registros en variables. Este criterio se evalúa en una escala de baja a muy alta, según la variedad de transformaciones disponibles y el grado de automatización proporcionado por la herramienta. Estas transformaciones pueden reducir la carga analítica y permitir el trabajo sobre representaciones más adecuadas para análisis posteriores sin requerir implementaciones externas.

El soporte multi-arquitectura cuantifica el número de arquitecturas de conjunto de instrucciones que pueden elevarse a la IR correspondiente. El dato se obtiene de la documentación oficial o de los repositorios de cada herramienta en el momento de realizar la evaluación. La cobertura arquitectónica condiciona el rango de binarios que puede analizarse con cada herramienta. Este criterio también influyó en la selección y el descarte de alternativas descritos en la Sección 4.2.

La extensibilidad e integración describe la disponibilidad de APIs, bibliotecas para lenguajes habituales en investigación, como Python, Rust o JavaScript, y mecanismos de interoperabilidad con herramientas externas de análisis, depuración, emulación o ejecución simbólica. Este campo agrupa deliberadamente la extensibilidad propia y la integración con terceros. Ambos criterios se solapan de forma significativa, ya que una API abierta y estable facilita, en la práctica, la integración con

herramientas externas.

La paralelización evalúa si la herramienta o el *pipeline* de análisis asociado proporciona soporte para la ejecución concurrente. Se distinguen tres niveles: soporte nativo, soporte parcial y ausencia de soporte documentado. Esta dimensión resulta relevante en escenarios de análisis a escala, como el propio *pipeline* de extracción desarrollado en este trabajo, que procesa un corpus de 960 binarios.

El soporte de comunidad evalúa cualitativamente el tamaño, la actividad y la madurez de la comunidad de usuarios y desarrolladores asociada a cada herramienta. Se consideran indicadores como la actividad en repositorios públicos, la existencia de foros técnicos, la frecuencia de actualizaciones y la adopción en entornos académicos o profesionales. Una comunidad activa puede favorecer la resolución de incidencias, la incorporación de nuevas arquitecturas y el acceso a ejemplos, documentación complementaria y recursos de aprendizaje. La documentación evalúa la calidad, completitud y accesibilidad de la documentación oficial y no oficial disponible para cada IR. Se distingue entre documentación oficial centralizada, documentación abundante pero distribuida entre múltiples fuentes y documentación funcional con escasa profundidad técnica. La calidad de la documentación condiciona la curva de aprendizaje, la correcta utilización de la herramienta y la reproducibilidad de los análisis realizados.

La madurez se aproxima mediante el año de publicación de la primera versión pública estable de la herramienta o del componente principal de su entorno. Este dato permite contextualizar temporalmente cada IR y estimar su grado de consolidación. Una mayor antigüedad no implica necesariamente una mayor calidad. Sin embargo, puede asociarse a una API más estable, una cobertura arquitectónica más amplia y una comunidad de usuarios más consolidada.

Por último, la facilidad de uso y accesibilidad evalúa la interacción requerida para trabajar con la herramienta y con su IR. Se consideran la disponibilidad de interfaz gráfica, línea de comandos, API programática, documentación introductoria, tutoriales y ejemplos. La clasificación se realiza según el perfil de usuario requerido, desde herramientas con una interfaz gráfica accesible hasta entornos que requieren integración programática y conocimiento especializado. Esta dimensión afecta a la productividad del analista y a la barrera de entrada para nuevos usuarios.

4.4. Comparación y análisis por dimensión

En la Tabla 4.1 se recogen los valores asignados a las seis representaciones intermedias según las catorce dimensiones definidas en la sección anterior (Sección 4.3).

Recuérdese que el nivel de abstracción, primera dimensión de la comparativa, mide cuánto se aleja cada IR del conjunto de instrucciones nativo. P-code se clasifica como una IR de nivel alto para análisis binario. Las instrucciones nativas se traducen a operaciones semánticas de propósito general y los efectos sobre registros, memoria y banderas pueden expresarse de forma explícita [17]. No obstante, la representación conserva referencias a espacios de direcciones y a tamaños de datos definidos por la arquitectura. BNIL alcanza un nivel de abstracción igualmente alto, pero no de forma monolítica, sino mediante una familia de tres capas encadenadas (*Low Level IL*, *Medium Level IL* y *High Level IL*), donde cada capa elimina progresivamente más información dependiente de arquitectura [39]. VEX IR, en cambio, se sitúa en un nivel medio: abstrae la instrucción nativa en sentencias atómicas, pero mantiene una dependencia arquitectónica explícita, ya que los registros se representan como posiciones dentro de un bloque de memoria propio en vez de como entidades independientes, sin ofrecer las abstracciones de variables o tipos que sí alcanzan P-code o BNIL en sus niveles superiores, quedando a medio camino entre el ensamblador y una IR de análisis de alto nivel [40]. ESIL se sitúa entre bajo y medio: es fundamentalmente una traducción directa de la semántica de cada instrucción a expresiones evaluables, sin abstraer registros ni pila, pero a diferencia de una simple recodificación de la ISA, sí normaliza esa semántica en una notación uniforme e independiente del juego de instrucciones original, lo que la separa mínimamente del nivel puramente bajo [41]. LLVM IR alcanza un nivel medio: abstrae por completo la sintaxis de la ISA de origen (no persiste ninguna instrucción ni registro nativo en su representación), pero mantiene tipos de bajo nivel (enteros de N bits, punteros, vectores) propios de una IR orientada a compilación, y su fidelidad semántica depende de si se genera desde código fuente o, como en este trabajo, desde un binario ya compilado, donde buena parte de esa información de tipos debe reconstruirse en vez de heredarse [42]. Hex-Rays Microcode, por último, no tiene un nivel de abstracción fijo, sino variable a lo largo de sus ocho niveles de madurez progresivos: comienza próximo al ensamblador (*maturity* 0) y culmina en una representación optimizada de alto nivel (*maturity* 7). Su clasificación como medio-alto refleja el nivel en el que habitualmente se realiza el análisis programático de esta IR en la práctica, no su techo máximo alcanzable [43].

Esta misma disociación entre abstracción declarada y diseño real se aprecia con más claridad al examinar la forma normal de cada IR, y en particular la propiedad SSA. LLVM IR es la única representación del conjunto cuya sintaxis se define en forma SSA global [16]. P-code bajo no se genera inicialmente en SSA, sino que Ghidra lo transforma durante una de las fases del decompilador [44]. BNIL proporciona varian-

tes SSA en los niveles superiores de su jerarquía a partir de LLIL [45, 46], mientras que VEX restringe la asignación única a temporales internos de un superbloque [47]. ESIL no proporciona una forma SSA [48]. En Hex-Rays Microcode, la información de equivalencia de valores se incorpora en niveles de madurez altos mediante identificadores GVN, pero no se expone como un formato SSA global equivalente al de LLVM IR [49].

El tratamiento de la memoria separa a las seis IR en familias de diseño que no coinciden con las fronteras anteriores. P-code generaliza el modelo plano mediante el concepto de *varnode* (una tripleta compuesta por espacio de direcciones, *offset* y tamaño), permitiendo que cada procesador defina múltiples espacios y dando soporte nativo a arquitecturas Harvard [50]. VEX IR representa los registros como posiciones dentro de un bloque de memoria propio llamado *Guest state*, con la memoria del programa accedida mediante operaciones de lectura y escritura, e incorpora además soporte explícito para mecanismos de segmentación en arquitecturas como x86 mediante registros de segmento [40]. ESIL distingue sintácticamente ambos conceptos: los registros se referencian directamente por su nombre en minúsculas, mientras que los accesos a memoria requieren una construcción explícita mediante corchetes (`[addr]` para lectura, `= [addr]` para escritura) [41]. Hex-Rays Microcode, de forma análoga, distingue registro y memoria a nivel de microinstrucción: las operaciones de carga y almacenamiento (`m_ldx`, `m_stx`) direccionan la memoria mediante un par de operandos *selector:offset* (`sel:off`) [43]. BNIL retrasa esta distinción hasta su segunda capa: en LLIL, tanto los registros como los accesos a memoria permanecen explícitos, y es en MLIL donde registros y accesos a pila se traducen conjuntamente a variables, mientras que los accesos a memoria general permanecen como operaciones explícitas de carga y almacenamiento [51]. LLVM IR no dispone de un espacio de registros propio. Los valores se expresan mediante nombres SSA y los accesos a memoria se representan mediante punteros e instrucciones como `load` y `store`. Sobre esta representación puede construirse una estructura de análisis adicional que aplica la misma propiedad de asignación única de SSA a las dependencias entre accesos a memoria, denominada MemorySSA [52].

La granularidad de operaciones, la unidad mínima de traducción, tampoco varía en la dirección que sugeriría el nivel de abstracción de cada IR. P-code y ESIL comparten una orientación a la instrucción nativa, pero con una diferencia real entre ambas: P-code la descompone en una o varias operaciones primitivas con nombre propio (`COPY`, `LOAD`, `STORE`, `BRANCH`, `CBRANCH`, `CALL`, entre otras) [50], mientras que ESIL traduce cada instrucción a una lista de expresiones sin una descomposición equivalente en primitivas nombradas, manteniendo una correspondencia más direc-

ta con la instrucción original [41]. VEX IR, por su parte, opera sobre el superbloque como unidad básica, agrupando entre una y aproximadamente cincuenta instrucciones nativas en una lista plana de sentencias atómicas [47]. LLVM IR adopta un formato de tres direcciones con 65 instrucciones básicas definidas [53]. BNIL se aparta de las anteriores al representar cada instrucción como un árbol de expresiones de longitud variable, donde los operandos pueden componerse recursivamente a partir de otras operaciones [45], y Hex-Rays Microcode permite que sus microinstrucciones se aniden como subinstrucciones, de modo que la complejidad crece a medida que la representación madura [54].

La expresividad confirma que un mayor nivel de abstracción no garantiza mayor fidelidad semántica, y en ocasiones la reduce. P-code hace explícitos efectos secundarios que en ensamblador permanecen implícitos, como la modificación de las banderas del procesador [17]. VEX IR, pese a compartir un nivel de abstracción comparativamente bajo con P-code, es menos fiel en construcciones concretas: Kim et al. documentan que VEX no es ni explícita ni autocontenida al calcular las EFLAGS de x86, ya que recurre a una función externa y a las variables de estado abstractas en vez de representar cada bandera como una variable propia, y catalogan bugs reales de fidelidad semántica en PyVEX, incluyendo un caso en el que la traducción de la instrucción `bt` altera el estado de la CPU de una forma no definida por la propia instrucción [11]. ESIL modela explícitamente las banderas del procesador y los accesos a memoria, aunque su sintaxis de tipo Forth dificulta representar semántica compleja [41]. BNIL sacrifica expresividad nativa a cambio de legibilidad analítica a medida que se asciende de capa: *Lifted IL* (la fase inicial de BNIL) preserva la semántica completa de la instrucción nativa, pero MLIL abstrae ya los efectos sobre las banderas [39]. LLVM IR captura semántica completa a nivel de lenguaje fuente, pero esa fidelidad se degrada parcialmente al elevar binarios ya compilados en vez de generarse desde código fuente [42]. Hex-Rays Microcode preserva semántica completa en sus niveles de madurez más bajos, abstrayéndola progresivamente hasta el pseudocódigo C final [54].

Estas cinco dimensiones de diseño semántico condicionan directamente las capacidades operativas de cada herramienta, empezando por la riqueza de transformaciones integradas. LLVM IR dispone del *pipeline* más extenso, con un sistema de transformaciones modular que incluye la sustitución de llamadas a una función por su cuerpo, vectorización y análisis de alias, entre otras optimizaciones inter e intraprocedurales [55]. BNIL aplica transformaciones acumulativas y específicas por capa: MLIL abstrae la pila y propaga constantes y HLIL elimina código muerto y recupera estructuras de control, un *pipeline* rico pero, a diferencia de LLVM IR, acoplado a

las tres capas fijas de BNIL en vez de a un sistema de transformaciones configurable [51, 56]. P-code y Hex-Rays Microcode aplican un núcleo de transformaciones comparable: propagación de constantes, eliminación de código muerto o construcción de forma SSA, exclusivamente como parte de sus respectivos procesos de compilación y no en la traducción inicial [44, 54]; Hex-Rays Microcode va un paso más allá en sus niveles de madurez superiores al incorporar identificadores GVN, una optimización global de la que P-code carece [49]. VEX IR aplica optimizaciones básicas durante el propio *lifting* (nivel 1 por defecto en PyVEX), dejando cualquier transformación avanzada al entorno de análisis que lo consuma [57], mientras que ESIL no incorpora ningún *pipeline* de normalización propio, ni siquiera a ese nivel básico [41].

El soporte multi-arquitectura, por el contrario, no guarda relación directa con la riqueza de transformaciones: Hex-Rays Microcode y ESIL, en extremos opuestos de expresividad y transformación, son precisamente las dos IR con mayor cobertura declarada (60+ y 40+ arquitecturas respectivamente) [24, 58]. P-code se sitúa en una posición intermedia, con 35 familias de procesadores soportadas actualmente en la versión 12.1 de Ghidra [59]. Por el contrario, VEX IR, BNIL y LLVM IR ofrecen la cobertura nativa más reducida de las seis: 10 arquitecturas nativas para VEX IR, 11 arquitecturas oficiales para BNIL (ampliables mediante complementos de la comunidad, en la misma línea que la extensibilidad vía SLEIGH de P-code [22, 47], y 20+ *backends* oficiales para LLVM IR, cuya cobertura real es considerablemente mayor si se cuentan los numerosos *backends* extraoficiales mantenidos por la comunidad fuera del árbol principal del proyecto [55].

La extensibilidad e integración de cada herramienta también revela una tensión entre apertura y dependencia de un entorno concreto. BNIL destaca por una API completamente abierta bajo licencia MIT, con integración para C++, Python y Rust, una ventaja diferencial frente a Ghidra, cuya extensibilidad está acoplada al entorno Java y a SLEIGH, y frente a VEX IR, cuya extensibilidad depende casi por completo del entorno de **angr**: modificar libVEX directamente es una tarea compleja, motivo por el cual PyVEX incorpora **Gymrat**, un nuevo entorno que desplaza el foco desde la manipulación de bits de bajo nivel hacia una API en Python de mayor nivel para quien quiera extender el soporte de arquitecturas [60]. ESIL se apoya en **r2pipe**, que permite controlar **radare2** desde prácticamente cualquier lenguaje (Python, JavaScript, Go, entre otros) a través de *stdin/stdout*, además de un sistema de *plugins* en C para extender arquitecturas y análisis; su integración con herramientas externas incluye GDB, WinDbg y Frida [24]. LLVM IR ofrece el entorno más amplio de *frontends* y herramientas de análisis externas [27], mientras que Hex-Rays Microcode,

pese a su SDK propietario en C++ e IDAPython, ha ampliado recientemente su accesibilidad mediante el nuevo IDA Domain API [61].

La paralelización debe interpretarse como una propiedad de la herramienta y de su interfaz de ejecución, no solo de la IR. LLVM IR proporciona infraestructura para ejecutar determinadas transformaciones de forma concurrente [62]. Binary Ninja incorpora análisis asíncrono [39]. Ghidra, radare2 y angr carecen de un mecanismo de paralelización nativo documentado; cualquier procesamiento concurrente de varios binarios debe implementarse externamente lanzando múltiples instancias o procesos independientes [24, 57, 63]. En particular, idalib facilita el uso no interactivo de IDA Pro, lo que permite integrarlo en flujos paralelos externos [25].

El soporte de comunidad muestra que la antigüedad no predice por sí sola el tamaño ni la naturaleza. Ghidra, pese a ser la más reciente de las seis herramientas (2019), goza de una adopción masiva tanto profesional como académica [59], mientras que IDA Pro, la más antigua (1991), mantiene una comunidad profesional consolidada pero de acceso restringido por su modelo de licencia [25]. BNIL cuenta con una comunidad activa pero comparativamente menos numerosa, de perfil marcadamente más orientado al pasatiempo que a la práctica profesional: la propia encuesta anual de Vector 35 a su comunidad de usuarios recoge que un 63% de los encuestados practica ingeniería inversa fuera de su trabajo, con un 79% de esa actividad centrada en retos recreativos como CTFs [5]. VEX IR se articula casi por completo en torno a angr, cuyo trabajo original [2] acumula cientos de citas y ha dado lugar a herramientas construidas sobre él como Firmalice, Driller o Rex, conformando una comunidad predominantemente académica. ESIL, por su parte, cuenta con una comunidad activa en GitHub y en canales de Telegram, pero fragmentada por una documentación inconsistente y una curva de aprendizaje elevada que limitan su adopción masiva frente al resto de herramientas de código abierto del conjunto [24]. LLVM IR tiene, con diferencia, la comunidad más amplia de las seis, con contribuciones activas de organizaciones como Apple, Google, Meta, ARM e Intel, además de numerosas universidades [64]. Hex-Rays Microcode, por último, hereda directamente la comunidad profesional consolidada de IDA Pro, con el mismo matiz de acceso restringido por licencia ya señalado [25].

La documentación presenta un patrón de calidad independiente de la madurez de cada herramienta. P-code combina dos realidades distintas: su documentación práctica es buena pero está repartida entre varias fuentes no centralizadas [44, 50]; su documentación formal, en cambio, presentaba una carencia real hasta fechas recientes: el propio trabajo de Naus et al. señala que la semántica de *high P-code* carecía de descripción formal oficial antes de su publicación [17]. BNIL cuenta con la docu-

mentación más estructurada de las seis, mantenida oficialmente por Vector 35 con guías separadas para cada capa del IL, ejemplos y referencia completa de API [39]. En el caso de VEX IR, la documentación se encuentra más dispersa, ya que no dispone de una documentación centralizada propia. La documentación de **angr** ofrece la información más accesible, aunque la referencia principal sigue siendo la cabecera `libvex_ir.h` de su repositorio, cuyos comentarios describen el funcionamiento de la IR [40, 47]. ESIL comparte ese carácter disperso pero con un problema adicional de vigencia: su referencia principal está desactualizada en varias secciones [41]. LLVM IR ofrece, con diferencia, la documentación más completa y centralizada del conjunto, con un manual de referencia del lenguaje mantenido de forma oficial y continua por el propio proyecto, además de guías específicas para cada componente de su entorno [64]. Hex-Rays Microcode cuenta con documentación oficial y con el SDK de Hex-Rays como principal referencia técnica, cuya guía se describe como detallada y bien estructurada por fuentes externas especializadas en ingeniería inversa [65], aunque la documentación disponible sobre su representación intermedia y sus APIs de bajo nivel ha sido históricamente menos accesible y exhaustiva que la de otras interfaces de IDA. Esta carencia de documentación pública ha favorecido la aparición de recursos comunitarios, como entradas de blog, proyectos de investigación y herramientas de visualización de terceros [43].

La madurez, aproximada mediante el año de la primera versión estable de cada herramienta, revela una distinción que las propias cifras podrían ocultar a primera vista: en varios casos, la antigüedad de la herramienta no coincide con la antigüedad de la capacidad concreta que este trabajo estudia. **Ghidra** (2019) [21] y **Binary Ninja** (2016) [22] son, de las seis, las herramientas donde la fecha de la herramienta y la de su IR coinciden, al haber sido P-code y BNIL parte de su diseño desde el origen. VEX IR, en cambio, existe desde el año 2002 como IR de **Valgrind** [66], pero su empleo para análisis puramente estático, en el que se centra este trabajo, solo se popularizó con la presentación pública de **angr** [67] en 2015, más de una década después. Un patrón similar, aunque más acusado, se observa en Hex-Rays Microcode: IDA **Pro** existe desde 1991 [68], pero la representación intermedia en sí permaneció como una estructura interna no documentada hasta que su API se hizo pública en IDA 7.1, en 2018, veintisiete años después del origen de la herramienta que lo contiene [26]. **radare2**, por su parte, es de 2009 [69], pero ESIL, su mecanismo de emulación, se introdujo posteriormente, en 2015 [70]. LLVM IR, surgida en 2002 como representación intermedia de un proyecto de compilación de propósito general [71], es la única de las seis IR cuyo propósito original no estuvo ligado al análisis de binarios. Posteriormente, fue incorporada a herramientas de este ámbito,

como **RetDec**.

La facilidad de uso, última dimensión de este bloque, separa a las IR comerciales de las académicas de forma más consistente que ninguna otra. P-code ofrece una curva de aprendizaje media: **Ghidra** dispone de una GUI completa y manuales oficiales, pero la interacción programática con P-code exige conocer la API Java o sus librerías de Python [50]. **Binary Ninja** e **IDA Pro** ofrecen las interfaces más pulidas del conjunto, con la curva de aprendizaje más baja [25, 72]. **angr** carece de GUI propia y exige una interacción programática en Python, con una curva de aprendizaje alta [73]. **radare2** va un peldaño más allá: además de carecer también de GUI propia, su sintaxis de tipo Forth hace de su línea de comandos la más exigente de las seis, con la curva de aprendizaje más alta del conjunto [41]. LLVM IR presenta una asimetría propia: su curva de aprendizaje es baja para quien ya trabaja en el ámbito de los compiladores, dada su documentación extensa y su diseño pensado para ese público, pero se eleva considerablemente para el análisis de binarios, donde su uso exige herramientas adicionales como **RetDec**, ajenas al propio proyecto LLVM [42]. Hex-Rays Microcode reproduce esta misma asimetría dentro de una sola herramienta: **IDA Pro** tiene la GUI más madura del sector, pero la interacción directa con la representación requiere un dominio avanzado del SDK en C++ o IDAPython [25].

Tabla 4.1: Comparativa cualitativa de las seis representaciones intermedias consideradas.

Característica	P-code	BNIL	VEX IR	ESIL	LLVM IR	Hex-Rays Microcode
Herramienta	Ghidra (NSA)	Binary Ninja (Vector 35)	Valgrind / angr / PyVEX	radare2	LLVM Project	IDA Pro (Hex-Rays)
Nivel de abstracción	Alto	Alto; familia multi-capa	Medio	Bajo-Medio	Medio	Medio-Alto; variable según madurez
Forma normal (SSA)	SSA durante de compilación, no en <i>low</i> P-code	SSA nativo en capas superiores	SSA en temporales de superbloque	Lineal, basada en pila; sin SSA	SSA global nativo	GVN en niveles altos; sin SSA global
Modelo de memoria	Espacios de direcciones y <i>varnodes</i>	Registros/pila como variables (MLIL)	<i>Guest state</i> y segmentación x86	Registros nombrados; memoria []	Sin registros propios; MemorySSA opcional	<code>sel:off</code> (registro y memoria)
Granularidad de operaciones	Microoperaciones atómicas y primitivas	Árbol de expresiones; operandos compuestos recursivamente	Instrucciones atómicas agrupadas en superbloques	Expresión por instrucción; sin descomposición en primitivas	Instrucciones de tres direcciones	Microinstrucciones anidables (<code>minsn_t</code>)
Expresividad	Explícita (banderas como variables)	Decreciente por capa	Implícita en EFLAGS; bugs de fidelidad	Explícita; sintaxis limitada	Fidelidad dependiente del lifter	Completa en niveles bajos
Transformaciones integradas	Media-Alta	Alta	Media	Baja	Muy alta	Alta
Soporte multi-arquitectura	35 familias de procesadores	11 arquitecturas oficiales + <i>plugins</i>	10 arquitecturas nativas	40+ arquitecturas	20+ <i>backends</i> oficiales	60+ familias de procesadores
Extensibilidad e integración	SLEIGH + API Java/Python	API abierta (C++/Py/Rust)	PyVEX + <i>Gymrat</i> ; ligada a <i>angr</i>	<i>r2pipe</i> multi-lenguaje	Transformaciones y múltiples <i>frontends</i>	SDK propietario (C++/IDAPython)
Paralelización	Externa por procesos	Análisis asíncrono	Limitada; depende del entorno	Externa, mediante <i>scripting</i>	Soporte nativo (procesos paralelos)	Soporte vía <i>idalib</i> headless
Soporte de comunidad	Adopción masiva (RE y academia)	Activa y especializada	Predominio académico (vía <i>angr</i>)	Activa, distribuida	Muy amplia (industrial y académica)	Comunidad profesional consolidada
Documentación	Buena pero dispersa	Oficial, centralizada	Funcional, no centralizada	Dispersa, centrada en <i>angr</i>	Amplia y centralizada	Oficial, limitada por licencia
Madurez	2019	2016	2002 (<i>Valgrind</i>); 2015 (<i>angr</i>)	2009 (<i>radare2</i>); 2015 (ESIL)	2002	1991 (IDA); 2018 (Microcode)
Facilidad de uso y accesibilidad	Media; GUI y API	Alta; GUI pulida	Baja; interacción programática	Baja; CLI compleja	Alta para compiladores; compleja para binarios	UI muy pulida; Microcode compleja

Capítulo 5

Metodología de la comparativa cuantitativa de las representaciones intermedias

Este capítulo presenta la segunda aportación original de este TFM: un marco de evaluación cuantitativa para las seis representaciones intermedias estudiadas. Primero se describe el diseño general del marco, incluyendo la arquitectura del *pipeline* de extracción y la implementación de los ocho *lifters* empleados. A continuación se define el corpus de binarios sobre el que se ejecuta el *pipeline*. Por último, se presentan los cuatro bloques de métricas cuantitativas: verbosidad y estructura, coste, robustez y agnosticismo interarquitectura.

5.1. Diseño general del marco de evaluación

El marco de evaluación cuantitativa se basa en un *pipeline* de extracción desarrollado específicamente para este trabajo, capaz de generar y analizar cada una de las representaciones intermedias a partir de un mismo corpus de binarios. Esta sección describe primero la arquitectura general del *pipeline* y las decisiones de diseño que garantizan la comparabilidad de los resultados; después se detalla la implementación de cada uno de los *lifters* utilizados.

Arquitectura general

El *pipeline* de extracción se organiza en torno a ocho *backends* de *lifting*¹, uno por cada combinación de herramienta y nivel de representación: P-code mediante `pyghidra`, VEX IR mediante `angr/PyVEX`, ESIL mediante `r2pipe`, LLVM IR mediante `RetDec`, Hex-Rays Microcode mediante `IDA Pro` en modo *headless* y BNIL mediante `Binary Ninja`. BNIL se desdobra en tres *backends* independientes, uno por cada nivel de abstracción (LLIL, MLIL, HLIL), lo que eleva a ocho el número total de *backends* pese a corresponder a seis representaciones intermedias.

Además de los ocho *backends* de *lifting*, el *pipeline* emplea `Capstone` como motor de desensamblado compartido para construir una base de referencia de instrucciones nativas independiente de cada *backend*. Esta base cumple una doble función: actúa como denominador común del ratio de expansión y sustenta la clasificación de cada instrucción nativa en una de las cuatro categorías (aritmética, control de flujo, memoria u otras) empleada en el bloque de robustez. El uso de una única taxonomía basada en `Capstone`, en lugar de la clasificación semántica propia de cada herramienta, evita introducir un sesgo de clasificación distinto según el *backend* de origen, y aprovecha una dependencia ya presente de forma transitiva a través de `angr/PyVEX`, sin añadir una herramienta nueva al *pipeline*.

Cada combinación (binario, *backend*) se ejecuta como un subproceso aislado, sin fusionar la ejecución de varios *backends* o binarios en un mismo proceso. Esta decisión, deliberada y transversal a todo el *pipeline*, responde a dos motivos. En primer lugar, garantiza la fidelidad de las mediciones de tiempo y memoria: al no compartir proceso con ninguna otra ejecución, el tiempo y el pico de memoria observados para una combinación concreta no pueden verse contaminados por la actividad de otro *backend* o binario procesándose en paralelo dentro del mismo intérprete. En segundo lugar, aísla los fallos: si un *backend* concreto falla o cuelga al procesar un binario determinado, el fallo no se propaga ni afecta a la ejecución del resto de combinaciones pendientes.

Los resultados de cada ejecución se escriben en formato JSONL, con una línea por combinación (binario, arquitectura, nivel de optimización, *backend*) escrita de forma incremental según se completa su procesamiento, en lugar de acumularse en memoria y volcarse al finalizar todo el corpus. Esta convención de escritura evita la pérdida de resultados ya obtenidos ante un fallo del *pipeline* en un binario concreto y permite reanudar o depurar la ejecución sin necesidad de repetir el experimento completo desde el principio.

¹Código fuente disponible en <https://github.com/reverseame/multi-arch-ir-analysis>.

Una vez completada la fase de *lifting*, un segundo conjunto de *scripts* del *pipeline* se encarga de procesar los resultados en bruto almacenados en formato JSONL para calcular cada uno de los cuatro bloques de métricas descritos.

Cada bloque de métricas expone sus resultados con un doble nivel de desglose, agrupando por combinación de arquitectura y anchura de registro (por ejemplo, x86-32, ARM-64, MIPS-32, MIPSEB-32) y agregando sobre binario, compilador y nivel de optimización dentro de cada grupo. Para cada celda (*backend*, arquitectura), se reporta la mediana y el rango intercuartílico (del inglés, IQR) junto con las estadísticas descriptivas habituales (media, mínimo, máximo, número de muestras).

El bloque de robustez incorpora, además, un contador de errores por celda de *backend* y arquitectura (número de errores del total de ejecuciones), compartido entre las cuatro categorías de esa combinación, dado que los errores observados en este trabajo se concentran de forma dispersa en un subconjunto reducido de combinaciones en lugar de distribuirse de forma continua, y una mediana o un IQR por sí solos no bastarían para hacer visible esa concentración. Cuando una métrica no está definida para la totalidad de los ocho *backends*, como ocurre con la fracción de escape respecto a **RetDec**, dicha ausencia se marca explícitamente como no disponible en lugar de rellenarse con un valor por defecto, evitando así que una celda vacía se confunda con un valor nulo real. La profundidad de anidamiento, por su parte, solo se calcula para VEX IR, las tres capas de BNIL y Hex-Rays Microcode, mientras que las operaciones SSA se calculan únicamente para BNIL y VEX IR, dos conjuntos de *backends* aplicables que no coinciden entre sí.

Cada ejecución del *pipeline* se lanza una única vez, sin mecanismo de reintento. Las ejecuciones que exceden el tiempo límite establecido se registran con un estado distintivo, diferenciado tanto de una ejecución completada con éxito como de una fallida, y quedan excluidas de las cifras de los cuatro bloques de métricas, al no producir ninguna representación intermedia sobre la que calcularlas, en lugar de contabilizarse como un error o descartarse de forma silenciosa sin dejar rastro en los resultados. La Figura 5.1 recoge de forma esquemática el funcionamiento completo del *pipeline* descrito, desde el corpus de binarios hasta las salidas agregadas de cada bloque de métricas.

***Lifters* individuales**

Los ocho *lifters* que componen el *pipeline* se han desarrollado como *scripts* independientes entre sí, ejecutables de forma aislada sin depender de un orquestador común ni de estado compartido con el resto de *backends*. Esta independencia es consistente con la decisión de aislamiento por subproceso, y responde a un objetivo

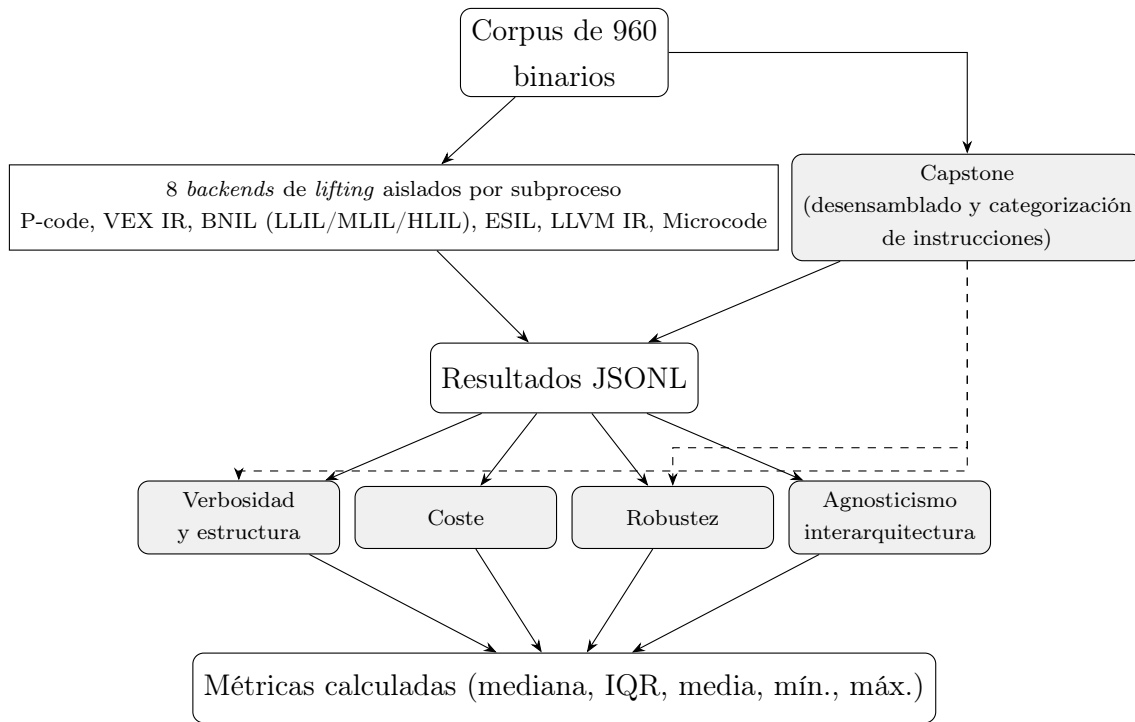


Figura 5.1: Arquitectura general del *pipeline* de extracción y cálculo de métricas.

adicional de escalabilidad: al no existir acoplamiento entre *lifters*, cada uno puede ejecutarse, depurarse o incluso reescribirse de forma aislada sin afectar al resto, y el conjunto puede distribuirse o paralelizarse en el futuro sobre varias máquinas o procesos sin necesidad de rediseñar el *pipeline*.

Esta propiedad resultó especialmente relevante dadas las restricciones de recursos del entorno de ejecución empleado en este trabajo, con una memoria disponible limitada y sin memoria de intercambio. Frente a este escenario, algunos *backends*, principalmente **RetDec** con picos de memoria cercanos a los 2,8 GiB en binarios de mayor tamaño y niveles de optimización altos, exigen un control de recursos por ejecución que solo es viable si cada *lifter* opera de forma aislada y desacoplada del resto.

P-code (pyghidra). La extracción de P-code se apoya en **pyghidra**, la biblioteca oficial de Python para integrar **Ghidra** con aplicaciones desarrolladas en este lenguaje, que sustituye al proyecto **pyhidra**, actualmente obsoleto, sobre el que se basaban versiones iniciales de este *lifter*. El *script* extrae exclusivamente *high P-code* para cada función del binario, sin extraer en ningún momento *low P-code*. Durante el desarrollo se corrigió un error de colisión de *offsets* de *varnode*, que provocaba que operaciones de P-code correspondientes a direcciones de memoria distintas se identificaran incorrectamente como la misma entidad al no considerar de forma con-

junta los tres componentes de la tripleta (espacio de direcciones, *offset*, tamaño) que define un *varnode* [50].

VEX IR (angr/PyVEX). La extracción de VEX IR se realiza a través de **angr**, operando a nivel de bloque básico: cada bloque del binario se eleva de forma independiente a su correspondiente *IR Super Block* (IRSB) mediante **PyVEX** [47].

BNIL (Binary Ninja). La extracción de BNIL se implementa como tres scripts independientes, uno por cada nivel de abstracción (LLIL, MLIL, HLIL), seleccionable mediante la API de **Binary Ninja**. Para evitar que el análisis de una repetición contamine el de la siguiente, dado que **Binary Ninja** cachea internamente las representaciones intermedias ya calculadas de un binario, cada repetición de medición se ejecuta sobre una instancia de la herramienta recién creada, en lugar de reutilizar una ya analizada.

ESIL (r2pipe). La extracción de ESIL se realiza mediante **r2pipe** [24]. El consumo de memoria se obtiene mediante **ru_maxrss**, que representa el punto máximo de memoria residente registrado para un proceso y sus procesos hijos, recuperado al finalizar la ejecución mediante la llamada al sistema **wait4**. Dado que **r2pipe** lanza **radare2** como un proceso hijo independiente que se recoge antes de finalizar el *script* de Python, el valor de **ru_maxrss** incorpora también el pico de memoria registrado por dicho proceso hijo. En este caso, el valor resultante se determina tomando el máximo entre los picos de memoria del proceso padre y de sus hijos, y no mediante su suma. Por tanto, puede subestimar el consumo conjunto real cuando ambos procesos mantienen una ocupación de memoria significativa de forma simultánea, ya que no hay garantía de que sus máximos se hayan producido en el mismo instante.

LLVM IR (RetDec). La extracción de LLVM IR se realiza invocando la interfaz de línea de comandos de **RetDec** sobre cada binario del corpus.

Hex-Rays Microcode (IDA Pro *headless*). La extracción de Microcode se realiza mediante **idalib**, que ejecuta el núcleo de IDA dentro del proceso de Python y permite medir directamente su consumo de memoria. Se descartó **headless-ida** porque se comunica con un proceso servidor independiente mediante RPC, lo que impide medir el consumo de ese proceso con el mismo método empleado para las demás herramientas.

5.2. Corpus de pruebas

El corpus de binarios empleado en la evaluación cuantitativa se compone de cinco utilidades reales (`grep`, `ls`, `nm`, `readelf` y `objdump`), seleccionadas por tres motivos. Primero, son programas de uso extendido cuyo código fuente es representativo de software de sistemas real, en contraste con binarios sintéticos o de *benchmark* contruidos específicamente para pruebas de rendimiento. Segundo, cubren una diversidad de tamaño y complejidad estructural: desde utilidades relativamente compactas (como `ls`), hasta herramientas de análisis de formato binario con una lógica de control de flujo considerablemente más rica, como `readelf` y `objdump` (ambas, junto con `nm`, pertenecientes a GNU Binutils), lo que permite que las métricas de verbosidad y complejidad estructural se evalúen sobre un espectro realista de tamaños de función y densidad de flujo de control, en lugar de sobre un único perfil de programa. Tercero, las cinco utilidades forman parte del conjunto de binarios distribuido por BinKit [20], lo que permite reutilizar directamente binarios ya compilados y verificados por trabajos previos en lugar de introducir una nueva cadena de compilación propia, reduciendo así una fuente adicional de variabilidad no controlada entre este trabajo y la literatura existente sobre similitud y análisis de código binario.

Cada una de las cinco utilidades ya está compilada para las ocho combinaciones de arquitectura y anchura de registro usadas en este trabajo (x86-32, x86-64, ARM-32, ARM-64, MIPS-32, MIPS-64, MIPSEB-32, MIPSEB-64), con seis compiladores (`gcc-4.9.4`, `gcc-6.4.0`, `gcc-8.2.0`, `clang-4.0`, `clang-6.0` y `clang-7.0`). Se trata por tanto de un subconjunto de los nueve compiladores que ofrece BinKit, seleccionado para cubrir tanto la familia de GCC como la de Clang en distintas generaciones de cada una, sin elevar en exceso el tamaño total del corpus. Cada combinación (utilidad, arquitectura, compilador) está compilada, además, en los cuatro niveles de optimización estándar de GCC/Clang (`-O0`, `-O1`, `-O2`, `-O3`), dado que el nivel de optimización es una de las variables que más condiciona tanto el coste de *lifting* como la estructura resultante de cada representación intermedia.

El resultado es un corpus factorial completo de $5 \times 8 \times 6 \times 4 = 960$ binarios (utilidades $\times$ combinaciones de arquitectura/anchura $\times$ compiladores $\times$ niveles de optimización), sobre el que se ejecutan los ocho *backends* de *lifting* descritos anteriormente, dando lugar a un total de 7 680 combinaciones (binario, *backend*) procesadas por el *pipeline*. Esto supone la traducción y el análisis de un total de 9,3 millones de funciones, cuya distribución entre las distintas IR no es homogénea. LLVM IR representa aproximadamente 750 000 funciones, mientras que VEX IR es la que concentra el mayor número, con un total de 1,8 millones.

5.3. Bloques de métricas

El marco de evaluación cuantitativa se organiza en cuatro bloques de métricas, cada uno dirigido a una propiedad distinta de las representaciones intermedias y calculado de forma independiente por el segundo conjunto de *scripts* del *pipeline*. Los bloques de verbosidad, coste y robustez se calculan sobre cada combinación (binario, *backend*) de forma individual; el bloque de agnosticismo, por el contrario, se calcula sobre pares de arquitecturas dentro de un mismo *backend*, tal como se justifica en su propia sección.

5.3.1. Verboisidad y estructura

El bloque de verbosidad y estructura mide cuánto se expande (o se contrae) el código nativo al elevarse a cada representación intermedia, y cómo se distribuye esa expansión entre distintos tipos de operación. La métrica central de este bloque es el ratio de expansión, definido como el cociente entre el número de operaciones (o líneas, según la naturaleza de la IR) de la representación resultante y el número de instrucciones nativas de las que procede, calculado a nivel de función. Esta métrica se reporta en dos variantes según la fuente de conteo empleada para cada *backend*: *expansion_ratio_ops*, que cuenta operaciones directamente sobre la IR nativa de cada herramienta, y *expansion_ratio_ast*, que cuenta nodos del árbol de sintaxis abstracta cuando el *backend* lo expone; para los *backends* cuya IR es intrínsecamente plana, ambas variantes coinciden por construcción, al no existir una estructura arbórea distinta del propio flujo de operaciones.

El ratio de expansión se desglosa en tres categorías: aritmética, control de flujo y acceso a memoria. La categoría “otras” agrupa, según la arquitectura, entre el 12,6 % y el 22,5 % de las instrucciones nativas e incluye movimientos de datos, conversiones de tipo e instrucciones *nop*. Aunque queda fuera del ratio de expansión por no encajar en las categorías anteriores, se incluye en la tasa de error del bloque de robustez, aplicable a cualquier instrucción con independencia de su categoría semántica. El desglose se incorporó tras detectar un error de clasificación en el *backend* de BNIL que reducía la categoría aritmética a valores próximos a cero al ignorar el contenido de las expresiones de asignación. Esto muestra que una cifra agregada puede ocultar comportamientos específicos de un *backend*.

El ratio de expansión se define, para la variante de operaciones, como

$$\text{ER}_{\text{ops}}(f) = \frac{\sum_{c \in \mathcal{C}} \text{IR}_{\text{ops}_c}(f)}{\text{native_instr}(f)},$$

donde $\mathcal{C} = \{\text{aritmética, control, memoria}\}$ es el conjunto de categorías de operación, y $\text{native_instr}(f)$ es el número de instrucciones nativas de la función. Este valor se obtiene directamente del registro de *lifting*, en lugar de recalcularse como la suma de las categorías nativas. Esta decisión permite evitar que un fallo parcial de desensamblado por parte de **Capstone** provoque un recuento incorrecto de instrucciones y, en consecuencia, un denominador artificialmente reducido o nulo. Para cada categoría, el ratio se calcula de forma independiente, relacionando las operaciones de la IR con las instrucciones nativas de la misma categoría:

$$\text{ER}_{\text{ops},c}(f) = \frac{\text{IR_ops}_c(f)}{\text{native_instr}_c(f)}, \quad c \in \mathcal{C}.$$

La variante *expansion_ratio_ast* sigue la misma fórmula sustituyendo el numerador por el número de nodos del árbol de sintaxis abstracta cuando el *backend* lo expone (BNIL, Hex-Rays Microcode); para las IR intrínsecamente planas (P-code, ESIL, LLVM IR), ambos conteos son, por construcción del *pipeline*, el mismo campo subyacente, no solo valores numéricamente iguales.

Junto al ratio de expansión, este bloque incluye tres métricas complementarias, todas ellas calculadas a nivel de función y agregadas mediante la media de los ratios individuales de cada función, no como un cociente global de sumas, de modo que cada función pondera por igual en el resultado agregado con independencia de su tamaño.

El número de variables temporales por instrucción se define como

$$\text{temp_ratio}(f) = \frac{n_{\text{temp}}(f)}{\text{native_instr}(f)},$$

es decir, el cociente entre el número de temporales introducidos por el *backend* y el número de instrucciones nativas de la función (no de operaciones de la IR resultante, a diferencia del denominador empleado en el ratio de expansión), con valor estructuralmente nulo para ESIL, cuya evaluación basada en pila no introduce temporales con nombre.

La profundidad de anidamiento mide cuántos niveles de composición recursiva alcanza la expresión más profunda de una función. Se define recursivamente: un nodo hoja (un registro, una constante, una referencia a variable) tiene profundidad 1 por definición; un nodo con hijos tiene profundidad igual a 1 más la profundidad de su hijo más profundo.

$$\text{depth}(n) = \begin{cases} 1 & \text{si } n \text{ no tiene hijos de expresión (nodo hoja)} \\ 1 + \max_{c \in \text{hijos}(n)} \text{depth}(c) & \text{en caso contrario,} \end{cases}$$

Por ejemplo, en la expresión $a + (b * c)$, el subárbol $(b * c)$ tiene profundidad 2 (sus dos hojas b y c aportan profundidad 1 cada una, más 1 por el propio operador de multiplicación), y la expresión completa tiene profundidad 3 (1 más la profundidad de su hijo más profundo, $(b * c)$). Esta métrica se limita, además, a expresiones aritméticas y lógicas: los operandos puramente estructurales de control de flujo, como el cuerpo de un bucle o los casos de un *switch* en HLIL, se excluyen del árbol antes de aplicar la fórmula, para no confundir la anidación léxica de una estructura de control con la anidación de una expresión de cómputo.

Se reportan dos estadísticos por función: la profundidad media sobre el conjunto de operaciones de la función, y la profundidad máxima de su instrucción de nivel superior más profunda. Esta métrica se calcula sobre las IR que admiten dicha composición recursiva de operandos: VEX IR, las tres capas de BNIL (LLIL, MLIL, HLIL) y Hex-Rays Microcode. Se excluye de P-code, LLVM IR y ESIL, cuyo ratio entre nodos AST y operaciones planas es exactamente 1,00 sobre el corpus de este trabajo, confirmando empíricamente la ausencia de estructura recursiva ya establecida cualitativamente en el Capítulo 4.

El número de operaciones SSA, por último, no admite una única definición común a los *backends* que la calculan, ya que cada uno expone una noción distinta de “forma SSA” sobre la que construir el cociente. Para las distintas capas de BNIL, se define como

$$\text{ssa_ratio}_{\text{BNIL}}(f) = \frac{n_{\text{SSA-instr}}(f)}{n_{\text{IL-instr}}(f)},$$

el cociente entre el número de instrucciones de la vista en forma SSA que **Binary Ninja** calcula internamente para cada nivel (LLIL, MLIL, HLIL) y el número de instrucciones de la vista base de ese mismo nivel. Es una auténtica razón de expansión entre dos representaciones del mismo código: la inserción de nodos ϕ añade operaciones reales, en lugar de limitarse a renombrar las ya existentes, por lo que esta razón nunca es inferior a 1. VEX IR, en cambio, no dispone de una vista SSA alternativa equivalente sobre la que construir ese mismo cociente. Su métrica se reformula, por tanto, como una magnitud distinta:

$$\text{ssa_ratio}_{\text{VEX}}(f) = \frac{n_{\text{temp}}(f)}{n_{\text{stmt}}(f)},$$

que define una densidad de declaraciones de variables temporales por sentencias de VEX IR (las mismas que ya son SSA por diseño dentro de cada superbloque), y no una razón de expansión entre dos formas del mismo código. Ambas métricas comparten nombre de columna por conveniencia de agregación en el *pipeline*, pero miden magnitudes conceptualmente distintas (expansión frente a densidad), por lo que no deben interpretarse como comparables entre sí. Esta distinción queda documentada

en el propio *pipeline* mediante un campo de advertencia adjunto a la salida de VEX IR.

LLVM IR y P-code quedan excluidas de esta métrica por el motivo opuesto: ambas producen, dadas las decisiones de extracción de este *pipeline*, un valor trivialmente constante y sin varianza. En el caso de LLVM IR, esto se debe a que se trata de una propiedad estructural garantizada por el formato para el 100 % de sus instrucciones. En el caso de P-code, porque el *backend* de este *pipeline* extrae exclusivamente *high P-code*, la representación posterior al proceso de decompilación de Ghidra, y nunca *low P-code*, de modo que toda operación medida procede ya de una etapa posterior a la construcción de SSA. Hex-Rays Microcode, pese a exponer SSA mediante identificadores GVN en sus niveles de madurez superiores, queda fuera del alcance de esta métrica en el presente trabajo: su extracción actual se apoya en la API de decompilación de alto nivel de IDA Pro en lugar de en `gen_microcode()`, por lo que incorporar esta métrica exigiría una segunda vía de extracción independiente sobre la estructura de la IR en bruto, un coste de implementación que se deja como línea de trabajo futuro.

5.3.2. Coste

El bloque de coste mide el consumo de recursos (tiempo y memoria) que exige obtener cada representación intermedia. Ambas magnitudes se miden a nivel de ejecución completa (binario, *backend*), no a nivel de función, y se agregan directamente mediante mediana e IQR sobre el conjunto de ejecuciones de cada celda, sin normalización adicional.

El tiempo de *lifting* se mide mediante dos cronómetros complementarios, comunes a los ocho *backends*: el tiempo de reloj del subproceso completo (`lifting_time_s`), medido externamente desde que se lanza el proceso hasta que termina, y el tiempo interno (`internal_lifting_time_s`), medido por el propio *script* de *lifting* una vez en ejecución, que excluye el arranque del intérprete y la carga de librerías. Ambas mediciones envuelven la totalidad del proceso de obtención de la IR propiamente dicho, incluyendo la fase de descubrimiento o análisis automático del binario (reconocimiento de funciones, construcción preliminar del CFG) en los *backends* que la requieren. Esta fase no se excluye de la medición: el tiempo reportado incluye siempre el coste de descubrimiento, ya que separarlo de forma fiable y comparable entre *backends* con arquitecturas de análisis tan distintas (el análisis automático de Ghidra, el *CFGFast* de `angr`, el análisis automático de `radare2`) resultó impracticable sin introducir una fuente de sesgo mayor que la que se pretendía evitar. La única excepción es el arranque de la máquina virtual de Java que requiere `pyghidra`,

excluido de ambas mediciones no por diseño, sino porque se inicializa antes de que comience el cronometraje de cada ejecución individual.

El consumo de memoria se obtiene mediante `ru_maxrss`, conforme al procedimiento descrito en la Sección 5.1. En el caso de ESIL, la medición presenta la limitación específica asociada al proceso hijo de `radare2`, descrita anteriormente.

Las ejecuciones que exceden el tiempo límite establecido se registran con un estado distintivo, sin reintento, y se excluyen tanto de las cifras de coste como de las de robustez, en lugar de contabilizarse como un fallo o descartarse de forma silenciosa.

5.3.3. Robustez

El bloque de robustez mide dos aspectos distintos de la fiabilidad de cada *backend* frente a binarios reales: la concentración de fallos parciales dentro de ejecuciones por lo demás completadas con éxito, y la dependencia de mecanismos de escape genéricos frente a construcciones propias de la representación intermedia cuando el *lifting* de una función concreta sí se completa.

La tasa de error por categoría se calcula únicamente sobre funciones pertenecientes a ejecuciones que completaron su procesamiento, reclasificando de forma independiente cada función según el tipo de instrucción nativa predominante en su rango de bytes (aritmética, de control de flujo, de acceso a memoria u otras) y definiéndose, para cada categoría c , como

$$\text{error_rate}_c = \frac{\text{error_counts}_c}{\text{ok_counts}_c + \text{error_counts}_c},$$

es decir, la proporción de funciones de la categoría c que fallaron al elevarse frente al total de funciones de esa misma categoría intentadas en la ejecución. Este cociente se calcula sobre el conjunto de funciones de cada ejecución. Además, para cada celda (*backend*, arquitectura), se informa del número de ejecuciones que presentan al menos un fallo y del total de ejecuciones completadas. A diferencia del ratio de expansión, la tasa de error sí incluye la categoría “otras”, aplicable a cualquier instrucción sin depender de su categoría semántica. Por construcción, esta tasa solo considera ejecuciones completadas y no captura los fallos que impiden completar una ejecución. Esta es una limitación del bloque tal como está definido.

La fracción de escape mide la proporción de operaciones de la IR que recurren a mecanismos de “escape” genéricos en lugar de a construcciones propias de la representación intermedia. Se define, para cada *backend* (salvo ESIL y LLVM IR) como

$$\text{escape_fraction}(f) = \frac{n_{\text{escape}}(f)}{n_{\text{IR}}(f)},$$

el cociente entre el número de operaciones de escape de la función y el tamaño total de su IR en dicho *backend*. ESIL, al carecer de una construcción de escape equivalente a nivel de operador, sustituye este cociente por

$$\text{escape_fraction}_{\text{ESIL}}(f) = \frac{n_{\text{uncovered}}(f)}{n_{\text{native_instr}}(f)},$$

que mide la proporción de instrucciones nativas no cubiertas por la traducción a ESIL sobre el total de instrucciones nativas de la función, una magnitud distinta que aproxima el mismo fenómeno de pérdida de fidelidad por una vía distinta, dada la ausencia de un mecanismo de escape explícito en esta IR. **RetDec** se excluye de esta métrica de forma explícita en la etapa de selección de ejecuciones a procesar, y dado que no se pudo confirmar durante el desarrollo del *pipeline* ninguna construcción de escape única y consistente en su salida.

5.3.4. Agnosticismo interarquitectura

El bloque de agnosticismo interarquitectura mide en qué medida cada representación intermedia consigue abstraer las diferencias entre arquitecturas de origen, de modo que un mismo programa, compilado para arquitecturas distintas, produzca una IR estructuralmente semejante. A diferencia de los tres bloques anteriores, sus métricas no se calculan sobre una arquitectura individual, sino sobre pares de combinaciones arquitectura/anchura, manteniendo fijos el binario, el compilador, la versión del compilador y el nivel de optimización entre ambos miembros del par: sin este control, la métrica mediría diferencias introducidas por el compilador o por el nivel de optimización, no por el proceso de *lifting* en sí.

La anchura de registro, sin embargo, no forma parte de esta agrupación, por lo que las ocho combinaciones arquitectura/anchura consideradas en este trabajo dan lugar a $\binom{8}{2} = 28$ pares por grupo y *backend*. Estos 28 pares mezclan tres comparaciones conceptualmente distintas, que se clasifican de forma separada. Los pares de arquitectura pura (misma anchura, distinta ISA; por ejemplo, ARM-32 frente a x86-32) aíslan la arquitectura como única variable y constituyen la comparación principal de este bloque (12 pares, seis por cada anchura). Los pares de anchura pura (misma ISA, distinta anchura; por ejemplo, ARM-32 frente a ARM-64) aíslan en cambio la anchura de registro como variable (4 pares), mientras que los pares mixtos (varían simultáneamente arquitectura y anchura; por ejemplo ARM-32 frente a x86-64) no permiten atribuir una diferencia observada a ninguna de las dos variables por separado (los 12 pares restantes). Las cuatro métricas de este bloque se reportan tanto por par individual como agregadas dentro de los pares de arquitectura pura y de anchura pura, de modo que la discusión de resultados pueda apoyarse en los primeros

como comparación principal, sin descartar la información adicional que aportan los segundos.

Las funciones se emparejan entre ambas combinaciones de un par por coincidencia exacta de nombre, sin mecanismos de *demangling* ni coincidencia aproximada para binarios sin símbolos, lo que puede subrepresentar el conjunto de funciones comparables en binarios muy optimizados donde el compilador realiza transformaciones en función de la arquitectura destino. Sobre las funciones emparejadas, se calculan cuatro métricas. La similitud ponderada de Jaccard (índice de Ruzicka) compara los histogramas de operaciones de una misma función entre ambas arquitecturas del par:

$$J_w(A, B) = \frac{\sum_k \min(A_k, B_k)}{\sum_k \max(A_k, B_k)},$$

donde A_k y B_k son las frecuencias de la operación k en cada histograma, calculado sobre la unión de claves presentes en ambos.

La similitud de Jensen-Shannon normaliza ambos histogramas a distribuciones de probabilidad (P, Q) y se define como

$$S_{JS}(P, Q) = 1 - \sqrt{JSD(P \parallel Q)}, \quad JSD(P \parallel Q) = \frac{1}{2}D_{KL}(P \parallel M) + \frac{1}{2}D_{KL}(Q \parallel M),$$

con $M = \frac{1}{2}(P + Q)$ y logaritmo en base 2. Es decir, se trata de uno menos la distancia de Jensen-Shannon, y no la divergencia en bruto, de modo que la métrica queda acotada en $[0, 1]$, con 1 indicando distribuciones idénticas.

El coeficiente de variación del tamaño de la IR se calcula, para cada función, sobre los dos únicos valores de tamaño de IR disponibles (uno por arquitectura del par). Con solo dos observaciones a y b , la desviación estándar muestral se reduce, por pura aritmética, a una expresión fija: la media es $(a + b)/2$, ambos valores se sitúan a la misma distancia de ella ($(a - b)/2$ en valor absoluto), y al aplicar la fórmula habitual de desviación estándar muestral el resultado es siempre $|a - b|/\sqrt{2}$, sin depender de ningún supuesto adicional sobre los datos. La métrica de CV es, por tanto, algebraicamente equivalente a

$$CV_2(a, b) = \frac{|a - b|}{\sqrt{2} \cdot \text{media}(a, b)},$$

es decir, una diferencia relativa simétrica normalizada entre ambos tamaños, y no un coeficiente de variación en el sentido distribucional habitual, que solo aporta información útil sobre la dispersión de una muestra a partir de tres o más observaciones. El delta de complejidad ciclomática se calcula, para cada función, sobre el grafo de flujo de control propio de la representación intermedia de cada *backend* (no sobre un grafo de control uniforme extraído directamente del binario nativo), mediante la

fórmula de McCabe para un único componente conexo, $M = E - N + 2$, donde E es el número de aristas y N el número de nodos de dicho grafo [74], y se define como la diferencia absoluta de dicha complejidad entre ambas arquitecturas del par. La fuente exacta del grafo difiere entre *backends*: el grafo de bloques propio de cada capa de BNIL, el grafo de transiciones de CFGFast en angr (que excluye aristas de llamada interprocedural pero incluye aristas de retorno simulado), el grafo de flujo de IDA Pro, el grafo de bloques del decompilador de Ghidra, el grafo de bloques ESIL de radare2, y una reconstrucción a partir del texto de LLVM IR en RetDec. De modo que, si bien la definición es consistente dentro de cada *backend*, la construcción del grafo de flujo de control no es idéntica entre los ocho: cada uno emplea su propia noción de nodo y arista para representarlo.

Capítulo 6

Experimentos y evaluación

Este capítulo presenta los resultados obtenidos al ejecutar el marco de evaluación cuantitativa descrito sobre el corpus completo de 960 binarios. Primero se describe el entorno de ejecución empleado. A continuación, se presentan los resultados de cada uno de los cuatro bloques de métricas, se discuten de forma conjunta integrando las perspectivas cualitativa y cuantitativa y, finalmente, se recogen las limitaciones identificadas a lo largo del proceso experimental.

6.1. Entorno de ejecución

El corpus completo se procesó en una única máquina virtual, bajo la misma configuración de software y de recursos para todos los *backends*. La configuración de la máquina virtual no se modificó durante la ejecución del experimento, salvo por la variación temporal inherente a la carga del sistema anfitrión.

Hardware. El experimento se ejecutó en una máquina virtual gestionada mediante Oracle VirtualBox, con 4 vCPU y 7,8 GiB de memoria asignados sobre un sistema anfitrión con procesador Intel Core i5-1135G7. Se estableció un límite de memoria de 5000 MiB (5 GiB) para el orquestador, seleccionado de forma conservadora para reservar recursos al sistema anfitrión y permitir la coejecución limitada de los *backends* más exigentes. Esta configuración se utilizó para calibrar los límites de concurrencia y memoria del *pipeline*, manteniéndolos acordes con los recursos disponibles. No obstante, los recursos asignados a la máquina virtual no caracterizan por completo la disponibilidad efectiva durante toda la ejecución. La carga variable del sistema anfitrión y los mecanismos de control del hipervisor, como la limitación

de la ejecución de CPU o el *ballooning* de memoria, pueden modificar los recursos efectivamente disponibles. Estos factores no se controlaron de forma independiente durante los varios días de procesamiento del corpus, por lo que su influencia sobre las cifras del bloque de coste no puede descartarse por completo.

Sistema operativo. Ubuntu 24.04.3 LTS (“Noble Numbat”), con núcleo Linux 7.0.0-28-generic para arquitectura x86_64.

Versiones del software. La Tabla 6.1 recoge las versiones exactas de las herramientas empleadas en el *pipeline*.

Tabla 6.1: Versiones del software empleado en el entorno de ejecución.

Componente	Versión
Python (entorno del <i>pipeline</i>)	3.12.3
angr / PyVEX / archinfo / cle / claripy	9.2.223
Capstone	5.0.6
pyghidra	3.1.0
Ghidra	12.1 PUBLIC (compilación 2026-05-13)
Java (para Ghidra)	OpenJDK Temurin 21.0.11+10
radare2	6.1.9 (+36540)
r2pipe	1.9.8
RetDec	v5.0
Binary Ninja	5.3.9757 (API comercial, modo <i>headless</i>)
IDA Pro	9.2
pyelftools	0.33

Límites de recursos. El *pipeline* limita el número de *workers* concurrentes a cuatro mediante el parámetro `--max-parallel`, de acuerdo con las cuatro CPU virtuales asignadas a la máquina virtual. Además, emplea un presupuesto de memoria de 5000 MiB mediante `--max-mem-mb` como criterio orientativo de planificación: el orquestador estima de antemano el consumo de memoria esperado para cada *backend*, a partir de una tabla de valores de referencia, y determina cuántas ejecuciones puede iniciar en paralelo sin superar dicho presupuesto. Este presupuesto no constituye un límite estricto impuesto por el sistema operativo. El *pipeline* no emplea `setrlimit` ni `cgroups` para restringir la memoria de cada subproceso. El pico de memoria de cada ejecución se registra posteriormente mediante `ru_maxrss`. Cada combinación (binario, *backend*) dispone de un tiempo límite de 1800 segundos (30

minutos). Este valor no se sobrescribió en ninguna invocación del *pipeline* durante el procesamiento del corpus completo. Binary Ninja incorpora, además, un límite de concurrencia específico e independiente del límite general de paralelización, con el fin de respetar las condiciones de su licencia.

6.2. Resultados por bloque de métricas

Los resultados se organizan en los cuatro bloques de métricas definidos en dicho capítulo: verbosidad y estructura, coste, robustez y agnosticismo interarquitectura. Cada tabla identifica el estadístico empleado para resumir los datos. La mayoría de las métricas se presentan mediante mediana e IQR, agregadas por combinación de arquitectura y *backend* en los tres primeros bloques, o por par de arquitecturas en el bloque de agnosticismo. Se emplea la media en su lugar allí donde la distribución de valores está fuertemente concentrada en cero, un escenario en el que la mediana no permite apreciar la señal real: la tasa de error por categoría, la fracción de escape y el delta de complejidad ciclomática dentro del bloque de agnosticismo. Las celdas correspondientes a combinaciones sin datos, ya sea por exclusión metodológica deliberada o por ausencia de resultados en el corpus procesado, se marcan explícitamente como no disponibles (N/D), distinguiendo ambos casos en el texto que acompaña a cada tabla.

Los resultados del primer bloque de métricas (verbosidad y estructura) confirman, con datos reales sobre el corpus completo de 960 binarios, la disociación entre nivel de abstracción declarado y verbosidad medida que ya anticipaba la comparativa cualitativa del Capítulo 4: las representaciones de mayor nivel de abstracción no son necesariamente las que producen más operaciones por instrucción nativa, sino frecuentemente las que menos.

El ratio de expansión en su variante calculada a partir de las operaciones, recogido en las Tablas 6.2, 6.3, 6.4 y 6.5, confirma con datos reales sobre el corpus completo la disociación entre nivel de abstracción declarado y verbosidad medida que ya anticipaba la comparativa cualitativa: HLIL y Hex-Rays Microcode, que corresponden a los niveles de representación más elevados analizados en este bloque, son también los *backends* que exhiben los menores ratios de expansión total de forma consistente (0,267–0,439 y 0,255–0,373, respectivamente, en las ocho arquitecturas), mientras que VEX IR y ESIL presentan, en general, los valores más altos. ESIL alcanza el máximo global en x86-32, con una mediana de 5,265, mientras que VEX IR llega a 4,727 en ARM-32. En varias arquitecturas, ambas representaciones superan una razón de 4:1.

El desglose por categoría (Tablas 6.3, 6.4 y 6.5) matiza esta lectura general. En la categoría aritmética, HLIL y Hex-Rays Microcode vuelven a situarse como los *backends* menos verbosos, con medianas cercanas a cero en la práctica totalidad de las arquitecturas, un resultado que, como se discute más adelante, exige matizarse al comparar con la variante *ast*. En control de flujo, ESIL rompe el patrón esperado: alcanza medianas de aproximadamente 5,6 en las arquitecturas MIPS y MIPSEB, muy por encima tanto del resto de *backends* como de sus propios valores en x86 y ARM, que se sitúan entre 1,15 y 1,80. Esto apunta a una traducción de las instrucciones de salto condicional de MIPS notablemente más verbosa para este *backend*, un patrón que la tabla total, al promediar entre categorías, no deja ver. En acceso a memoria, HLIL vuelve a ser el *backend* más compacto, con medianas iguales o próximas a 0 en todas las arquitecturas. LLIL y VEX IR presentan, en general, valores elevados y una dispersión considerable, aunque ESIL alcanza valores superiores en las arquitecturas x86. Sus IQR abarcan en varios casos desde 0 hasta valores superiores a 1, lo que refleja una traducción de los accesos a memoria menos uniforme entre funciones.

Un segundo patrón, no anticipado en la comparativa cualitativa, es la práctica invariancia del ratio de expansión de cada *backend* entre arquitecturas. Los valores de HLIL, por ejemplo, oscilan entre 0,267 y 0,439 en la variante total a lo largo de las ocho combinaciones de arquitectura y anchura, lo que sugiere que la verbosidad de cada representación depende más de su propio diseño que de la arquitectura de origen del binario.

El ratio de expansión en su variante calculada a partir del árbol de sintaxis abstracta (Tablas 6.6, 6.7, 6.8 y 6.9) es, por definición, idéntico a la variante calculada a partir de las operaciones para P-code, ESIL y LLVM IR, al tratarse de representaciones de estructura plana sin una noción de nodo de árbol distinta de la propia operación. Para VEX IR y las tres capas de BNIL, en cambio, esta variante es sistemáticamente igual o mayor que la calculada a partir de las operaciones en todas las celdas. VEX IR pasa de una mediana de 4,727 operaciones por instrucción en ARM-32 a 10,500 nodos por instrucción en la variante AST, más del doble: aunque VEX IR se organiza como una lista plana de sentencias a nivel de superbloque, cada una de estas sentencias puede contener expresiones anidadas recursivamente, que solo esta variante computa. Hex-Rays Microcode presenta el mismo patrón de forma más moderada, sin llegar a duplicar en ningún caso el valor de la variante basada en operaciones, lo que indica que sus expresiones contienen menos nodos adicionales por operación que las de VEX IR y BNIL, aunque la razón exacta depende también de la estructura concreta de las microinstrucciones.

El contraste más notable de esta comparación es HLIL en la categoría aritmética. Su mediana calculada a partir de las operaciones es prácticamente nula (0,000 en seis de las ocho arquitecturas; Tabla 6.3), mientras que su mediana calculada a partir del árbol de sintaxis abstracta toma valores sustancialmente mayores (0,538–1,000; Tabla 6.7). Esto no es una anomalía de los datos, sino una consecuencia directa del diseño de HLIL: a nivel de operación de primer orden, una expresión aritmética suele quedar plegada dentro de una instrucción de nivel superior clasificada en otra categoría, típicamente una asignación, y solo emerge como nodo aritmético propio al considerar la totalidad del árbol de sintaxis abstracta.

En control de flujo, la variante AST apenas se separa de la variante basada en operaciones para la mayoría de los *backends*. La excepción principal es Hex-Rays Microcode, cuya mediana pasa aproximadamente de 0,625–0,667 a 0,845–0,933. Esto sugiere que parte de la estructura de control implícita en el flujo de las microinstrucciones aparece como nodo propio al considerar el árbol completo. En acceso a memoria (Tabla 6.9), LLIL y Hex-Rays Microcode muestran el mayor salto entre ambas variantes: LLIL pasa de una mediana en torno a 1,0 a 1,13–1,18 en ARM, y Hex-Rays Microcode multiplica su valor por un factor de entre 3 y 6 según la arquitectura (por ejemplo, de 0,086 a 0,271 en x86-32), reflejando que buena parte de la complejidad de un acceso a memoria en estas dos representaciones se articula en subexpresiones anidadas, como el cálculo de direcciones y las desreferencias compuestas, que la variante calculada a partir de las operaciones no computa como nodos propios.

Este conjunto de resultados ilustra por qué el bloque de verbosidad reporta ambas variantes por separado en lugar de una única cifra: la elección de granularidad de conteo (operación de primer orden frente a árbol completo) puede alterar sustancialmente la interpretación de dónde se concentra la expansión de una IR, y esa alteración no es uniforme entre categorías ni entre *backends*.

La Tabla 6.10 recoge el número de variables temporales introducidas por instrucción nativa, con el denominador establecido en la Sección 5.3.1, correspondiente al número de instrucciones nativas y no al número de operaciones de la IR resultante. VEX IR introduce, con diferencia, el mayor número de temporales por instrucción del conjunto, con medianas de 3,0 a 5,4 según la arquitectura y un tercer cuartil de 15,0 en ARM-32, coherente con su forma SSA estricta a nivel de temporales dentro de cada superbloque, que exige nombrar explícitamente cada valor intermedio en lugar de reutilizar variables. LLVM IR y P-code le siguen a considerable distancia, con medianas entre 0,25 y 0,63, y Hex-Rays Microcode se sitúa en el extremo opuesto entre los *backends* con temporales medibles, con valores en torno a 0,07–0,11.

ESIL muestra un valor estructuralmente nulo en las ocho arquitecturas, consistente con su evaluación basada en pila, que no introduce temporales con nombre propio. Las tres capas de BNIL presentan un patrón distinto que conviene no confundir con el de ESIL pese a mostrar la misma mediana: MLIL y HLIL sí introducen temporales, pero en una proporción tan reducida que sus medianas y rangos intercuartílicos se redondean a 0,000 en todas las arquitecturas. Sus medias, sin embargo, son positivas y no nulas (0,0073 en MLIL, 0,0011 en HLIL para ARM-32, con valores máximos de hasta 1,0 y 0,5 respectivamente), reflejando que el proceso de simplificación de expresiones de estas dos capas elimina la práctica totalidad de los temporales heredados de niveles inferiores, sin eliminarlos por completo. LLIL, la capa menos abstracta de BNIL, conserva una proporción mayor de temporales (media de 0,018 en ARM-32) y, a diferencia de MLIL y HLIL, presenta variación real entre arquitecturas: su rango intercuartílico superior alcanza 0,022 en ARM-32 y 0,011 en x86-32, pero es exactamente 0,000–0,000 en MIPS y MIPSEB, sugiriendo una traducción con menos temporales de bajo nivel para estas dos arquitecturas concretas.

Las Tablas 6.11 y 6.12 recogen, respectivamente, la profundidad media y máxima de anidamiento por función. Estas métricas se aplican a los *backends* cuya IR admite la composición recursiva de operandos: VEX IR, las tres capas de BNIL (LLIL, MLIL y HLIL) y Hex-Rays Microcode. HLIL presenta la mayor profundidad media, tanto entre las capas de BNIL como en el conjunto completo, con medianas de entre 2,857 y 3,053 según la arquitectura. Le siguen LLIL, con valores entre 2,619 y 2,971, y MLIL, con la menor profundidad de las tres (entre 2,250 y 2,333). MLIL queda por debajo de LLIL en las ocho arquitecturas del corpus. Este resultado es compatible con que MLIL, al abstraer registros y accesos a pila mediante variables, sustituya las cadenas de temporales y banderas que LLIL representa mediante composición anidada por estructuras más anchas, pero menos profundas. VEX IR y Hex-Rays Microcode ocupan las posiciones siguientes, aunque su orden varía según la arquitectura: VEX IR supera a Hex-Rays Microcode en ARM y MIPS, mientras que ocurre lo contrario en x86.

La profundidad máxima muestra un patrón distinto. VEX IR se mantiene en torno a 3 en ARM y MIPS, pero presenta valores inferiores en x86, especialmente en x86-64, cuya mediana es 1,0. LLIL y MLIL ocupan un rango intermedio, con medianas de 4,0 y entre 4,0 y 5,0, respectivamente. HLIL alcanza medianas de hasta 6,0 en MIPS y MIPSEB de 64 bits. Además, en la totalidad de las funciones elevadas por su *backend*, VEX IR no supera una profundidad máxima de 3. Esto se observa en más de 1,8 millones de funciones analizadas únicamente con **angr**. El resultado es compatible con la representación de VEX IR como una lista de sentencias dentro de

superbloques, aunque los datos no permiten afirmar que 3 sea un límite teórico de la IR.

El valor máximo de profundidad registrado para HLIL es 130, debido a un artefacto aislado de la decompilación de **Binary Ninja** que afecta a varias macros sencillas de una línea de *gnulib*: `c_isascii`, `c_isprint`, `c_isgraph`, `c_isalnum` y `c_isalpha`. Este comportamiento solo aparece en binarios compilados con Clang 6.0/7.0; esas mismas funciones presentan profundidades de entre 3 y 5 con otros compiladores, arquitecturas o niveles de optimización. Además, VEX IR y Hex-Rays Microcode no muestran una anomalía equivalente al calcular la métrica de forma independiente.

La Tabla 6.13 recoge la métrica de SSA de BNIL y VEX IR, agrupadas visualmente en columnas distintas dentro de la misma tabla dado que ambas magnitudes no son comparables entre sí: la primera es una razón de expansión entre la vista SSA y la vista base de cada capa de BNIL, mientras que la segunda es una densidad de declaraciones de temporales por sentencia de VEX IR.

Para BNIL, el mínimo observado en las tres capas es exactamente 1,0 en la totalidad del corpus, confirmando que la forma SSA nunca contiene menos instrucciones que la vista base correspondiente. Sin embargo, esta igualdad no es un caso marginal: el 28,2 % de las funciones de LLIL, el 34,8 % de MLIL y el 46,4 % de HLIL presentan una razón exactamente igual a 1,0, es decir, sin expansión alguna al construir su forma SSA. Esta proporción crece con el nivel de abstracción de la capa. Este patrón es compatible con que HLIL haya eliminado buena parte de las asignaciones múltiples a una misma variable antes de calcular su forma SSA, dejando menos margen para que la colocación de nodos ϕ introduzca instrucciones adicionales.

Para VEX IR, la métrica definida para esta representación no muestra una tendencia dominante en el conjunto completo del corpus: de las 1,8 millones de funciones analizadas, el 43,6 % presenta valores inferiores a 1, el 9,1 % valores exactamente iguales a 1 y el 47,3 % valores superiores a 1. En conjunto, los resultados muestran una distribución sin una concentración clara de los valores a ninguno de los dos lados de 1. La Tabla 6.13 revela que este comportamiento depende de la arquitectura: ARM se sitúa sistemáticamente en o por encima de 1 (mediana de 1,03 a 1,23), MIPS y MIPSEB se sitúan por debajo (mediana de 0,91 a 0,95), y x86-64 constituye un caso atípico con una dispersión considerablemente mayor que el resto (mediana 2,0, IQR hasta 7,0, y un máximo observado en todo el corpus de 21,6). Una posible explicación es que las instrucciones CISC de x86, cuya semántica puede ser más compleja, se descompongan en un número proporcionalmente mayor de temporales de VEX IR por sentencia que las instrucciones de las ISA RISC consideradas en el corpus. Esta hipótesis no se verifica de forma independiente en este trabajo.

El bloque de coste mide el tiempo y la memoria que exige obtener cada representación intermedia. El tiempo reportado corresponde a la medición interna del propio *script* de cada *backend*, en lugar de al tiempo de reloj del subproceso completo.

El tiempo de *lifting*, mostrado en la Tabla 6.14 sitúa a ESIL y LLVM IR como los *backends* más rápidos del conjunto. ESIL presenta medianas entre 22,3 y 42,7 segundos, mientras que LLVM IR se sitúa entre 18,3 y 61,7 segundos en las arquitecturas con datos disponibles. P-code es el más lento, con medianas entre 191,9 y 309,5 segundos. En algunas arquitecturas, P-code multiplica varias veces el tiempo de ESIL o LLVM IR, aunque la magnitud exacta depende de la arquitectura y del *backend* utilizado como referencia.

Las tres capas de BNIL muestran un patrón que la estabilidad arquitectónica observada en el bloque de verbosidad no anticipaba: su coste temporal depende de la arquitectura de origen. En x86, sus medianas se sitúan aproximadamente entre 62 y 97 segundos; en ARM, entre 81,5 y 167,8 segundos; y en MIPS y MIPSEB, aproximadamente entre 88 y 131 segundos. HLIL alcanza 167,8 segundos en ARM-32, más del doble que su valor en x86-64. Esta asimetría no se observa en ningún otro *backend* del conjunto en la misma magnitud y sugiere que el proceso de análisis y decompilación de **Binary Ninja** resulta considerablemente más costoso sobre binarios RISC que sobre x86 para este corpus concreto.

El consumo de memoria (Tabla 6.15) presenta un ordenamiento distinto al del tiempo. ESIL es el *backend* más ligero, con medianas entre aproximadamente 72,9 y 123,5 MiB. P-code y las capas de BNIL se encuentran entre los *backends* más exigentes, con medianas que superan con frecuencia 1 230 MiB. LLVM IR ocupa una posición intermedia en las arquitecturas disponibles, pero presenta la mayor dispersión relativa, con IQR que en varios casos abarcan desde aproximadamente 253 hasta más de 1 569 MiB, reflejando una sensibilidad al tamaño y la complejidad del binario considerablemente mayor que la del resto de *backends*.

El bloque de robustez mide dos aspectos distintos de la fiabilidad de cada *backend*: la proporción de funciones que fallan al elevarse dentro de ejecuciones completadas con éxito, desglosada por el tipo de instrucción nativa predominante en cada función, y la proporción de operaciones que recurren a mecanismos de escape en lugar de a construcciones propias de la representación intermedia, calculada también a nivel de función.

La tasa de error total presenta una mediana igual a 0,000 en la práctica totalidad de las celdas. Las tres capas de BNIL constituyen la única excepción sistemática, con una mediana no nula en ARM-32, donde alcanza 0,002 en las tres capas. El resto de los *backends* presenta medianas de 0,000 o valores residuales en las celdas

correspondientes. Dado que la mediana tiende a ocultar errores concentrados en un subconjunto reducido de funciones o ejecuciones, este resultado debe interpretarse junto con la media reportada por celda: la de BNIL en ARM-32 alcanza 0,071–0,074 según la capa, frente a valores de 0,000–0,010 en el resto de *backends* para esa misma arquitectura, y las tres capas mantienen una media sistemáticamente superior al resto en todas las combinaciones arquitectura/anchura, no solo en ARM-32.

El desglose por categoría (Tablas 6.16, 6.17, 6.18 y 6.19) reclasifica cada función según el tipo de instrucción nativa predominante en su rango de bytes y confirma que la tasa de error de las tres capas de BNIL no se concentra en ninguna de estas categorías en particular, sino que se distribuye de forma prácticamente uniforme entre las cuatro: en ARM-32, la media oscila entre 0,066 y 0,078 en las cuatro categorías para cada una de las tres capas, sin que ninguna destaque sistemáticamente sobre las demás. Este resultado no respalda la hipótesis de que los errores de BNIL se concentren exclusivamente en un tipo concreto de función, como las dominadas por aritmética compleja o por accesos indirectos a memoria. Apunta, en cambio, a una causa más general del proceso de análisis de Binary Ninja sobre binarios ARM, coherente con el patrón de coste temporal elevado ya observado para esta misma combinación anteriormente. Cabe señalar que esta granularidad no permite distinguir si el fallo de una función se debe a una única instrucción problemática o a la función en su conjunto, una limitación que se retoma más adelante.

La fracción de escape (Tabla 6.20) mide, a diferencia de la tasa de error anterior, la proporción de funciones que recurren a algún mecanismo de escape frente al total de funciones elevadas con éxito, calculada función a función y agregada directamente sobre el conjunto de todas las funciones del corpus para cada celda (*backend*, arquitectura), sin un paso intermedio de agregación por ejecución: una función completada pesa por igual en la distribución con independencia del binario al que pertenezca, de modo que un binario con más funciones contribuye proporcionalmente más datos que uno con menos.

Cada *backend* define su propio mecanismo de escape mediante un conjunto cerrado y fijo de construcciones, no una heurística de puntuación: P-code lo identifica mediante el mnemónico `CALLOTHER`; VEX IR mediante sentencias de tipo `Ist_Dirty`; BNIL mediante el sufijo de operación `INTRINSIC` en sus distintas variantes, con una salvedad propia de este *backend* frente al resto: cuando la instrucción de nivel superior es una asignación pura, el mecanismo de escape se busca en la expresión de origen envuelta por dicha asignación en lugar de en la propia asignación, de modo que una construcción como `var = intrinsic(...)` se identifique correctamente como escape a través de la expresión anidada; y Hex-Rays Microcode mediante ope-

randos de tipo `mop_h` (*helper function*), que representan una llamada a una función auxiliar del tiempo de ejecución en lugar de una dirección o una operación primitiva de *microcode*. ESIL constituye un caso distinto, porque su valor no representa la proporción de operaciones de escape, sino la proporción de instrucciones nativas sin traducción válida. Por ello, sus resultados deben interpretarse como una medida de cobertura y no como una fracción de escape estrictamente comparable con la de los demás *backends*. ESIL presenta, en consecuencia, la media más alta del conjunto en la mayoría de arquitecturas, con su valor máximo en ARM-32 (0,030) y valores igualmente elevados en MIPS y MIPSEB (0,022–0,025), consistente con que su métrica no mide una construcción de escape propiamente dicha, sino una cobertura de traducción distinta en naturaleza al resto de *backends*. Hex-Rays Microcode presenta, en cambio, una media moderada pero sistemáticamente superior a P-code, VEX IR y BNIL en todas las arquitecturas (0,0008–0,0029 frente a valores mayoritariamente inferiores a 0,0009 en el resto), sin llegar a la magnitud de ESIL. LLVM IR se excluye de esta tabla en su totalidad, marcado como no disponible por diseño y no por ausencia de datos, dado que no se pudo confirmar durante el desarrollo del *pipeline* ninguna construcción de escape única y consistente en su salida.

El bloque de agnosticismo mide en qué medida cada representación intermedia consigue abstraer las diferencias entre arquitecturas de origen. Los resultados sobre los doce pares de arquitectura pura (Tablas 6.21–6.24) confirman, de forma consistente en las cuatro métricas y en los ocho *backends*, la hipótesis de control planteada en la Sección 4.1: el par MIPS-32 versus MIPSEB-32 y su equivalente de 64 bits, que difieren únicamente en el orden de bytes, obtienen una similitud de Jaccard y de Jensen-Shannon de 1,000 en la totalidad de los *backends* sin una sola excepción, un coeficiente de variación del tamaño de 0,000 y deltas de complejidad ciclomática muy bajos. En los pares MIPS/MIPSEB, estos deltas se sitúan entre 0,010 y 0,423 según el *backend*, mientras que en los demás pares alcanzan valores entre 0,55 y 10,52. Estos resultados respaldan la interpretación de que las cuatro métricas capturan diferencias reales de *lifting* entre arquitecturas genuinamente distintas y no únicamente ruido de bajo nivel.

ESIL constituye el caso atípico más marcado del conjunto en las comparaciones cruzadas de arquitectura. Su similitud de Jaccard desciende hasta valores entre 0,191 y 0,407, inferiores a los observados en los demás *backends*, y su delta de complejidad ciclomática alcanza valores de aproximadamente 10,3–10,5 en los pares que involucran x86-32, entre tres y seis veces superior al del resto. Hex-Rays Microcode y P-code se sitúan, de forma consistente, entre los *backends* con mayor invariancia interarquitectura del conjunto: presentan similitudes elevadas y, en general, coefi-

cientes de variación del tamaño y deltas de complejidad ciclomática reducidos.

Las celdas de LLVM IR correspondientes a los pares que involucran MIPS-64 o MIPSEB-64 se marcan como no disponibles. Esta ausencia es coherente con el fallo sistemático de **RetDec** en esas dos combinaciones, dado que dichas arquitecturas no están soportadas por la herramienta.

Las Tablas 6.25–6.28 recogen los cuatro pares de anchura pura, cada uno correspondiente a la misma arquitectura en sus dos anchuras de registro (32 y 64 bits). Comparando estos resultados agregados con los doce pares de arquitectura pura anteriores, se observa que, para las métricas basadas en histogramas de operaciones y tamaño de IR, la variación de anchura de registro produce, en general, una divergencia menor que la variación de arquitectura. El efecto no es uniforme entre *backends* y no se reproduce de forma sistemática en el delta de complejidad ciclomática.

ESIL constituye el caso más destacado: su similitud ponderada de Jaccard pasa de 0,297 en los pares de arquitectura pura a 0,644 en los pares de anchura pura, el mayor incremento observado entre las dos categorías de pares, confirmado de forma independiente tanto por la similitud de Jensen-Shannon (0,374 a 0,646) como por el coeficiente de variación del tamaño de la IR (0,173 a 0,097). Las tres capas de BNIL muestran el mismo patrón de forma consistente en las tres métricas, aunque de magnitud más moderada: LLIL, MLIL y HLIL incrementan su similitud ponderada de Jaccard de 0,681, 0,834 y 0,846 a 0,778, 0,875 y 0,861, respectivamente. P-code, VEX IR, LLVM IR y Hex-Rays Microcode se mueven, en cambio, en la dirección opuesta, con una divergencia ligeramente mayor en los pares de anchura pura, aunque en P-code, LLVM IR y Hex-Rays Microcode la magnitud del cambio es reducida (inferior a 0,03 en similitud ponderada de Jaccard). VEX IR presenta el cambio más pronunciado de los cuatro (0,753 a 0,710), aunque sigue siendo un orden de magnitud menor que el incremento observado en ESIL o BNIL. El patrón no es, por tanto, mayoritario en sentido estricto entre los ocho *backends*, cuatro incrementan su similitud y cuatro la reducen, pero el efecto de mayor magnitud del conjunto, el de ESIL, junto con la mejora consistente de las tres capas de BNIL, apunta a que la variación de anchura de registro puede resultar, para varios *backends*, menos disruptiva para el *lifting* que la variación de arquitectura. El coeficiente de variación del tamaño de la IR sí sigue una tendencia mayoritaria más clara, con una media que se reduce de 0,067 en pares de arquitectura pura a 0,055 en pares de anchura pura, con solo tres excepciones (P-code, HLIL y LLVM IR) que aumentan ligeramente en sentido contrario.

El delta de complejidad ciclomática rompe este patrón. A diferencia de las tres métricas anteriores, construidas a partir de histogramas de operaciones o del tamaño

de la IR por función, esta métrica no comparte de forma consistente la misma dirección: ESIL mejora sustancialmente en los pares de anchura pura (de una media de 5,459 a 2,141, coherente con su comportamiento en las demás métricas), pero VEX (de 1,990 a 2,382) y Hex-Rays Microcode (de 1,488 a 1,946) empeoran, mientras que P-code, BNIL y LLVM IR se mantienen aproximadamente estables en ambos tipos de par.

Esta divergencia entre familias de métricas resulta significativa. Las métricas basadas en la distribución de operaciones y el tamaño de la IR no se comportan necesariamente como la métrica basada en la forma del grafo de flujo de control.

Una explicación plausible, aunque no verificada en este trabajo, es que las diferencias de convención de llamada y asignación de registros entre variantes de 32 y 64 bits de una misma arquitectura puedan reestructurar la disposición de los bloques básicos. Esto puede ocurrir, por ejemplo, mediante bloques adicionales de guardado y recuperación de registros o prólogos de función distintos, sin alterar sustancialmente la combinación de operaciones utilizada. El comportamiento de división de bloques de VEX y Hex-Rays Microcode podría ser más sensible a este efecto que el de P-code o BNIL.

Tabla 6.2: Ratio de expansión (ops, total), mediana (IQR), por arquitectura y *backend*.

Arquitectura	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
x86-32	1,000 (0,682–1,462)	1,000 (0,956–1,000)	0,754 (0,625–0,880)	0,404 (0,262–0,523)	4,000 (2,000–5,120)	5,265 (4,128–5,979)	1,595 (1,057–2,250)	0,327 (0,222–0,456)
x86-64	1,132 (0,783–1,692)	1,000 (0,941–1,000)	0,835 (0,730–1,000)	0,439 (0,276–0,667)	2,000 (2,000–5,238)	4,310 (3,351–5,106)	1,673 (1,143–2,420)	0,373 (0,257–0,500)
ARM-32	1,000 (0,667–1,395)	1,111 (1,000–1,333)	0,857 (0,741–1,000)	0,421 (0,290–0,585)	4,727 (4,000–7,000)	3,844 (3,320–4,380)	1,667 (1,023–2,127)	0,339 (0,244–0,500)
ARM-64	1,000 (0,647–1,441)	1,070 (1,000–1,194)	0,765 (0,625–0,875)	0,321 (0,227–0,500)	4,286 (2,500–4,892)	3,385 (2,778–3,799)	1,235 (0,800–1,721)	0,323 (0,240–0,483)
MIPS-32	0,833 (0,500–1,237)	0,974 (0,900–1,000)	0,662 (0,519–0,767)	0,299 (0,204–0,447)	3,425 (3,097–3,745)	3,487 (3,052–3,875)	1,061 (0,657–1,648)	0,265 (0,176–0,403)
MIPS-64	0,800 (0,492–1,237)	0,989 (0,912–1,021)	0,593 (0,500–0,719)	0,267 (0,166–0,427)	3,625 (3,333–3,849)	3,426 (3,070–3,893)	N/D	0,255 (0,150–0,378)
MIPSEB-32	0,840 (0,506–1,237)	0,974 (0,900–1,000)	0,660 (0,517–0,766)	0,300 (0,204–0,447)	3,423 (3,099–3,747)	3,481 (3,045–3,870)	1,053 (0,648–1,648)	0,266 (0,178–0,403)
MIPSEB-64	0,799 (0,490–1,234)	0,989 (0,912–1,021)	0,592 (0,500–0,717)	0,267 (0,165–0,428)	3,622 (3,333–3,846)	3,419 (3,067–3,887)	N/D	0,255 (0,149–0,377)

Tabla 6.3: Ratio de expansión (ops, aritmética), mediana (IQR), por arquitectura y *backend*.

Arquitectura	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
x86-32	1,200 (0,667–1,824)	0,750 (0,583–1,000)	0,277 (0,000–0,500)	0,024 (0,000–0,129)	3,250 (2,583–4,444)	5,429 (4,500–6,857)	1,116 (0,600–1,571)	0,022 (0,000–0,158)
x86-64	1,600 (1,000–2,375)	0,545 (0,333–0,800)	0,333 (0,056–0,576)	0,029 (0,000–0,149)	3,768 (2,800–5,500)	5,500 (4,500–7,250)	1,417 (1,000–1,933)	0,048 (0,000–0,200)
ARM-32	1,378 (0,736–2,010)	0,750 (0,500–1,000)	0,423 (0,000–0,708)	0,000 (0,000–0,123)	3,000 (2,167–4,747)	7,833 (5,500–10,375)	1,095 (0,500–1,625)	0,015 (0,000–0,222)
ARM-64	1,596 (0,778–2,778)	1,000 (0,800–1,125)	0,333 (0,000–0,588)	0,000 (0,000–0,118)	3,204 (1,801–5,000)	4,500 (3,000–6,500)	1,000 (0,333–1,800)	0,000 (0,000–0,226)
MIPS-32	1,167 (0,571–2,000)	0,688 (0,538–0,889)	0,294 (0,000–0,500)	0,000 (0,000–0,083)	2,455 (1,444–3,806)	5,000 (4,000–7,000)	0,795 (0,286–1,250)	0,000 (0,000–0,152)
MIPS-64	1,015 (0,500–1,923)	0,693 (0,535–0,840)	0,200 (0,000–0,407)	0,000 (0,000–0,061)	2,154 (1,857–3,321)	5,000 (3,647–6,194)	N/D	0,000 (0,000–0,127)
MIPSEB-32	1,167 (0,571–2,000)	0,692 (0,538–0,891)	0,297 (0,000–0,500)	0,000 (0,000–0,083)	2,458 (1,437–3,833)	5,000 (4,000–7,000)	0,800 (0,259–1,250)	0,000 (0,000–0,154)
MIPSEB-64	1,003 (0,500–1,909)	0,696 (0,537–0,842)	0,200 (0,000–0,400)	0,000 (0,000–0,061)	2,143 (1,854–3,308)	5,000 (3,636–6,185)	N/D	0,000 (0,000–0,125)

Tabla 6.4: Ratio de expansión (ops, control de flujo), mediana (IQR), por arquitectura y *backend*.

Arquitectura	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
x86-32	1,000 (1,000–1,100)	1,114 (1,000–1,250)	1,115 (1,000–1,250)	1,000 (0,712–1,000)	1,467 (1,000–1,667)	1,743 (1,000–2,093)	1,000 (0,929–1,190)	0,667 (0,333–0,833)
x86-64	1,000 (1,000–1,125)	1,111 (1,000–1,250)	1,115 (1,000–1,250)	1,000 (0,729–1,000)	1,458 (1,000–1,667)	1,727 (1,000–2,084)	1,000 (0,909–1,214)	0,667 (0,333–0,833)
ARM-32	1,000 (0,962–1,100)	1,111 (1,000–1,371)	1,111 (1,000–1,375)	0,923 (0,600–1,000)	1,250 (1,000–1,583)	1,667 (1,000–2,300)	1,000 (0,851–1,111)	0,625 (0,000–0,869)
ARM-64	1,000 (1,000–1,125)	1,125 (1,000–1,250)	1,132 (1,000–1,250)	0,949 (0,692–1,000)	1,400 (1,000–1,625)	1,800 (1,000–2,182)	1,000 (0,833–1,111)	0,667 (0,333–0,857)
MIPS-32	1,000 (0,967–1,053)	1,125 (1,000–1,253)	1,125 (1,000–1,259)	0,875 (0,652–1,000)	1,333 (1,000–1,631)	5,667 (5,000–6,000)	0,977 (0,640–1,000)	0,667 (0,383–0,857)
MIPS-64	1,000 (0,978–1,045)	1,143 (1,000–1,300)	1,150 (1,000–1,304)	0,863 (0,625–1,000)	1,222 (1,000–1,562)	5,581 (5,000–6,000)	N/D	0,667 (0,333–0,857)
MIPSEB-32	1,000 (0,976–1,059)	1,125 (1,000–1,255)	1,125 (1,000–1,258)	0,882 (0,667–1,000)	1,333 (1,000–1,632)	5,667 (5,000–6,000)	0,980 (0,636–1,000)	0,667 (0,400–0,857)
MIPSEB-64	1,000 (0,976–1,043)	1,143 (1,000–1,300)	1,150 (1,000–1,308)	0,864 (0,625–1,000)	1,222 (1,000–1,562)	5,583 (5,000–6,000)	N/D	0,667 (0,333–0,857)

Tabla 6.5: Ratio de expansión (ops, acceso a memoria), mediana (IQR), por arquitectura y *backend*.

Arquitectura	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
x86-32	0,286 (0,069–0,500)	1,000 (0,989–1,000)	0,222 (0,000–0,400)	0,000 (0,000–0,018)	1,200 (0,000–1,410)	1,333 (1,205–1,500)	0,696 (0,321–1,286)	0,086 (0,000–0,219)
x86-64	0,403 (0,154–0,800)	1,000 (0,965–1,000)	0,321 (0,059–0,571)	0,000 (0,000–0,026)	0,000 (0,000–1,300)	1,378 (1,228–1,667)	0,943 (0,444–1,636)	0,133 (0,000–0,333)
ARM-32	0,333 (0,063–0,706)	1,133 (1,000–1,640)	0,457 (0,143–0,769)	0,000 (0,000–0,043)	1,200 (1,000–1,684)	1,164 (1,030–1,438)	0,798 (0,333–1,571)	0,095 (0,000–0,333)
ARM-64	0,329 (0,000–0,667)	1,080 (1,000–1,320)	0,333 (0,045–0,600)	0,000 (0,000–0,013)	1,104 (1,000–1,333)	1,152 (1,000–1,375)	0,455 (0,108–1,000)	0,089 (0,000–0,289)
MIPS-32	0,286 (0,056–0,556)	1,000 (0,964–1,000)	0,257 (0,028–0,500)	0,000 (0,000–0,015)	1,000 (1,000–1,000)	1,000 (1,000–1,000)	0,500 (0,176–1,000)	0,074 (0,000–0,224)
MIPS-64	0,270 (0,073–0,543)	0,953 (0,872–1,000)	0,176 (0,000–0,357)	0,000 (0,000–0,008)	1,000 (1,000–1,000)	1,000 (1,000–1,000)	N/D	0,040 (0,000–0,161)
MIPSEB-32	0,288 (0,057–0,565)	1,000 (0,965–1,000)	0,252 (0,026–0,488)	0,000 (0,000–0,015)	1,000 (1,000–1,000)	1,000 (1,000–1,000)	0,500 (0,170–1,000)	0,075 (0,000–0,226)
MIPSEB-64	0,269 (0,071–0,540)	0,953 (0,873–1,000)	0,176 (0,000–0,357)	0,000 (0,000–0,008)	1,000 (1,000–1,000)	1,000 (1,000–1,000)	N/D	0,040 (0,000–0,161)

Tabla 6.6: Ratio de expansión (ast, total), mediana (IQR), por arquitectura y *backend*.

Arquitectura	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
x86-32	1,000 (0,682–1,462)	3,259 (2,909–3,700)	2,127 (1,698–2,577)	1,643 (1,263–2,323)	6,000 (2,000–11,519)	5,265 (4,128–5,979)	1,595 (1,057–2,250)	0,612 (0,419–0,859)
x86-64	1,132 (0,783–1,692)	3,000 (2,556–3,512)	2,309 (2,000–2,743)	2,000 (1,409–2,578)	2,000 (2,000–11,875)	4,310 (3,351–5,106)	1,673 (1,143–2,420)	0,714 (0,500–1,000)
ARM-32	1,000 (0,667–1,395)	3,622 (3,242–4,067)	2,556 (2,000–3,210)	2,047 (1,282–2,779)	10,500 (8,552–17,000)	3,844 (3,320–4,380)	1,667 (1,023–2,127)	0,667 (0,447–0,917)
ARM-64	1,000 (0,647–1,441)	3,903 (3,364–4,328)	2,192 (1,750–2,615)	1,533 (1,017–2,333)	9,474 (3,500–11,080)	3,385 (2,778–3,799)	1,235 (0,800–1,721)	0,625 (0,471–0,918)
MIPS-32	0,833 (0,500–1,237)	3,406 (3,167–3,719)	1,857 (1,453–2,349)	1,416 (0,912–1,981)	7,271 (6,000–8,135)	3,487 (3,052–3,875)	1,061 (0,657–1,648)	0,500 (0,349–0,727)
MIPS-64	0,800 (0,492–1,237)	3,553 (3,250–3,827)	1,682 (1,250–2,200)	1,321 (0,857–1,783)	7,667 (7,000–8,368)	3,426 (3,070–3,893)	N/D	0,500 (0,310–0,722)
MIPSEB-32	0,840 (0,506–1,237)	3,405 (3,111–3,722)	1,857 (1,419–2,343)	1,429 (0,952–2,000)	7,268 (6,016–8,136)	3,481 (3,045–3,870)	1,053 (0,648–1,648)	0,500 (0,350–0,727)
MIPSEB-64	0,799 (0,490–1,234)	3,554 (3,250–3,828)	1,679 (1,250–2,195)	1,319 (0,856–1,784)	7,667 (7,000–8,367)	3,419 (3,067–3,887)	N/D	0,500 (0,309–0,722)

Tabla 6.7: Ratio de expansión (ast, aritmética), mediana (IQR), por arquitectura y *backend*.

Arquitectura	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
x86-32	1,200 (0,667–1,824)	2,211 (1,667–3,167)	0,892 (0,500–1,286)	0,750 (0,391–1,111)	3,250 (2,583–4,444)	5,429 (4,500–6,857)	1,116 (0,600–1,571)	0,682 (0,267–1,162)
x86-64	1,600 (1,000–2,375)	2,250 (1,500–3,585)	1,083 (0,800–1,500)	1,000 (0,667–1,333)	3,768 (2,800–5,500)	5,500 (4,500–7,250)	1,417 (1,000–1,933)	1,000 (0,467–1,500)
ARM-32	1,378 (0,736–2,010)	2,188 (1,754–3,167)	1,176 (0,727–1,764)	1,000 (0,500–1,500)	3,000 (2,167–4,747)	7,833 (5,500–10,375)	1,095 (0,500–1,625)	0,826 (0,216–1,364)
ARM-64	1,596 (0,778–2,778)	3,400 (2,103–5,000)	1,111 (0,556–1,757)	1,000 (0,375–1,571)	3,204 (1,801–5,000)	4,500 (3,000–6,500)	1,000 (0,333–1,800)	1,000 (0,357–1,886)
MIPS-32	1,167 (0,571–2,000)	2,714 (1,987–4,111)	0,916 (0,333–1,278)	0,706 (0,222–1,111)	2,455 (1,444–3,806)	5,000 (4,000–7,000)	0,795 (0,286–1,250)	0,688 (0,111–1,216)
MIPS-64	1,015 (0,500–1,923)	2,560 (1,750–3,570)	0,700 (0,214–1,097)	0,538 (0,143–0,933)	2,154 (1,857–3,321)	5,000 (3,647–6,194)	N/D	0,550 (0,091–1,118)
MIPSEB-32	1,167 (0,571–2,000)	2,733 (2,000–4,143)	0,917 (0,333–1,286)	0,700 (0,222–1,111)	2,458 (1,437–3,833)	5,000 (4,000–7,000)	0,800 (0,259–1,250)	0,692 (0,111–1,227)
MIPSEB-64	1,003 (0,500–1,909)	2,552 (1,750–3,560)	0,700 (0,200–1,091)	0,538 (0,143–0,929)	2,143 (1,854–3,308)	5,000 (3,636–6,185)	N/D	0,554 (0,091–1,111)

Tabla 6.8: Ratio de expansión (ast, control de flujo), mediana (IQR), por arquitectura y *backend*.

Arquitectura	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
x86-32	1,000 (1,000–1,100)	1,114 (1,000–1,250)	1,115 (1,000–1,250)	1,000 (0,951–1,000)	1,467 (1,000–1,667)	1,743 (1,000–2,093)	1,000 (0,929–1,190)	0,909 (0,750–1,000)
x86-64	1,000 (1,000–1,125)	1,111 (1,000–1,250)	1,115 (1,000–1,250)	1,000 (0,951–1,009)	1,458 (1,000–1,667)	1,727 (1,000–2,084)	1,000 (0,909–1,214)	0,933 (0,750–1,000)
ARM-32	1,000 (0,962–1,100)	1,111 (1,000–1,371)	1,111 (1,000–1,375)	1,000 (0,824–1,143)	1,250 (1,000–1,583)	1,667 (1,000–2,300)	1,000 (0,851–1,111)	0,845 (0,500–1,000)
ARM-64	1,000 (1,000–1,125)	1,125 (1,000–1,250)	1,132 (1,000–1,250)	1,000 (0,911–1,036)	1,400 (1,000–1,625)	1,800 (1,000–2,182)	1,000 (0,833–1,111)	0,929 (0,720–1,000)
MIPS-32	1,000 (0,967–1,053)	1,125 (1,000–1,253)	1,125 (1,000–1,259)	1,000 (0,875–1,089)	1,333 (1,000–1,631)	5,667 (5,000–6,000)	0,977 (0,640–1,000)	0,889 (0,640–1,000)
MIPS-64	1,000 (0,978–1,045)	1,143 (1,000–1,300)	1,150 (1,000–1,304)	1,000 (0,889–1,143)	1,222 (1,000–1,562)	5,581 (5,000–6,000)	N/D	0,857 (0,500–1,000)
MIPSEB-32	1,000 (0,976–1,059)	1,125 (1,000–1,255)	1,125 (1,000–1,258)	1,000 (0,880–1,091)	1,333 (1,000–1,632)	5,667 (5,000–6,000)	0,980 (0,636–1,000)	0,889 (0,636–1,000)
MIPSEB-64	1,000 (0,976–1,043)	1,143 (1,000–1,300)	1,150 (1,000–1,308)	1,000 (0,889–1,143)	1,222 (1,000–1,562)	5,583 (5,000–6,000)	N/D	0,857 (0,500–1,000)

Tabla 6.9: Ratio de expansión (ast, acceso a memoria), mediana (IQR), por arquitectura y *backend*.

Arquitectura	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
x86-32	0,286 (0,069–0,500)	1,171 (1,077–1,333)	0,282 (0,036–0,500)	0,239 (0,000–0,450)	1,200 (0,000–1,410)	1,333 (1,205–1,500)	0,696 (0,321–1,286)	0,271 (0,059–0,500)
x86-64	0,403 (0,154–0,800)	1,167 (1,077–1,333)	0,400 (0,125–0,792)	0,367 (0,091–0,750)	0,000 (0,000–1,300)	1,378 (1,228–1,667)	0,943 (0,444–1,636)	0,406 (0,157–0,789)
ARM-32	0,333 (0,063–0,706)	1,182 (1,000–1,667)	0,500 (0,191–0,833)	0,478 (0,158–0,759)	1,200 (1,000–1,684)	1,164 (1,030–1,438)	0,798 (0,333–1,571)	0,333 (0,021–0,714)
ARM-64	0,329 (0,000–0,667)	1,133 (1,000–1,348)	0,364 (0,100–0,700)	0,333 (0,058–0,647)	1,104 (1,000–1,333)	1,152 (1,000–1,375)	0,455 (0,108–1,000)	0,333 (0,000–0,673)
MIPS-32	0,286 (0,056–0,556)	1,000 (1,000–1,000)	0,294 (0,071–0,534)	0,273 (0,062–0,500)	1,000 (1,000–1,000)	1,000 (1,000–1,000)	0,500 (0,176–1,000)	0,250 (0,000–0,500)
MIPS-64	0,270 (0,073–0,543)	1,000 (1,000–1,000)	0,231 (0,039–0,438)	0,215 (0,030–0,417)	1,000 (1,000–1,000)	1,000 (1,000–1,000)	N/D	0,222 (0,000–0,481)
MIPSEB-32	0,288 (0,057–0,565)	1,000 (1,000–1,000)	0,289 (0,067–0,527)	0,267 (0,061–0,500)	1,000 (1,000–1,000)	1,000 (1,000–1,000)	0,500 (0,170–1,000)	0,250 (0,000–0,500)
MIPSEB-64	0,269 (0,071–0,540)	1,000 (1,000–1,000)	0,231 (0,039–0,437)	0,216 (0,031–0,417)	1,000 (1,000–1,000)	1,000 (1,000–1,000)	N/D	0,222 (0,000–0,481)

Tabla 6.10: Temporales por instrucción, mediana (IQR), por arquitectura y *backend*.

Arquitectura	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
x86-32	0,320 (0,086–0,574)	0,000 (0,000–0,011)	0,000 (0,000–0,000)	0,000 (0,000–0,000)	3,949 (2,000–4,100)	0,000 (0,000–0,000)	0,595 (0,351–0,900)	0,083 (0,033–0,135)
x86-64	0,353 (0,108–0,630)	0,000 (0,000–0,004)	0,000 (0,000–0,000)	0,000 (0,000–0,000)	4,622 (4,000–7,000)	0,000 (0,000–0,000)	0,625 (0,391–0,963)	0,095 (0,038–0,154)
ARM-32	0,250 (0,077–0,462)	0,000 (0,000–0,022)	0,000 (0,000–0,000)	0,000 (0,000–0,000)	5,407 (4,333–15,000)	0,000 (0,000–0,000)	0,477 (0,313–0,784)	0,111 (0,048–0,190)
ARM-64	0,363 (0,148–0,643)	0,000 (0,000–0,000)	0,000 (0,000–0,000)	0,000 (0,000–0,000)	4,097 (2,250–5,000)	0,000 (0,000–0,000)	0,410 (0,246–0,689)	0,100 (0,044–0,167)
MIPS-32	0,298 (0,132–0,519)	0,000 (0,000–0,000)	0,000 (0,000–0,000)	0,000 (0,000–0,000)	3,017 (2,607–3,253)	0,000 (0,000–0,000)	0,350 (0,188–0,588)	0,077 (0,034–0,132)
MIPS-64	0,284 (0,133–0,510)	0,000 (0,000–0,000)	0,000 (0,000–0,000)	0,000 (0,000–0,000)	3,351 (2,671–3,667)	0,000 (0,000–0,000)	N/D	0,069 (0,032–0,124)
MIPSEB-32	0,300 (0,133–0,514)	0,000 (0,000–0,000)	0,000 (0,000–0,000)	0,000 (0,000–0,000)	3,014 (2,600–3,250)	0,000 (0,000–0,000)	0,348 (0,185–0,583)	0,080 (0,037–0,134)
MIPSEB-64	0,283 (0,133–0,510)	0,000 (0,000–0,000)	0,000 (0,000–0,000)	0,000 (0,000–0,000)	3,348 (2,667–3,667)	0,000 (0,000–0,000)	N/D	0,069 (0,032–0,124)

Tabla 6.11: Profundidad de anidamiento media, mediana (IQR), por arquitectura y *backend*.

Arquitectura	VEX IR	LLIL	MLIL	HLIL	H-R Microcode
x86-32	1,667 (1,000–2,049)	2,778 (2,500–3,000)	2,250 (2,000–2,478)	2,857 (2,237–3,235)	1,826 (1,451–2,000)
x86-64	1,000 (1,000–2,079)	2,650 (2,430–3,000)	2,280 (2,000–2,521)	3,000 (2,333–3,395)	1,869 (1,496–2,091)
ARM-32	2,055 (1,986–2,133)	2,619 (2,383–2,780)	2,318 (2,000–2,574)	2,912 (2,400–3,286)	1,750 (1,394–2,000)
ARM-64	2,036 (1,625–2,113)	2,773 (2,633–2,969)	2,321 (2,056–2,538)	3,000 (2,547–3,468)	1,852 (1,500–2,086)
MIPS-32	1,944 (1,852–2,000)	2,833 (2,686–3,000)	2,318 (2,056–2,553)	3,000 (2,571–3,361)	1,769 (1,400–2,075)
MIPS-64	2,000 (1,943–2,023)	2,971 (2,838–3,083)	2,333 (2,068–2,588)	3,053 (2,667–3,556)	1,833 (1,421–2,353)
MIPSEB-32	1,943 (1,854–2,000)	2,833 (2,686–3,000)	2,316 (2,053–2,552)	3,000 (2,571–3,353)	1,768 (1,400–2,071)
MIPSEB-64	2,000 (1,943–2,023)	2,971 (2,838–3,083)	2,333 (2,068–2,589)	3,053 (2,667–3,556)	1,835 (1,421–2,357)

Tabla 6.12: Profundidad de anidamiento máxima, mediana (IQR), por arquitectura y *backend*.

Arquitectura	VEX IR	LLIL	MLIL	HLIL	H-R Microcode
x86-32	2,000 (1,000–3,000)	5,000 (4,000–5,000)	4,000 (2,000–5,000)	5,000 (3,000–7,000)	4,000 (2,000–6,000)
x86-64	1,000 (1,000–3,000)	5,000 (3,000–5,000)	4,000 (2,000–5,000)	5,000 (3,000–7,000)	4,000 (2,000–6,000)
ARM-32	3,000 (3,000–3,000)	4,000 (3,000–5,000)	4,000 (3,000–4,000)	5,000 (3,000–7,000)	3,000 (2,000–5,000)
ARM-64	3,000 (3,000–3,000)	4,000 (4,000–5,000)	4,000 (3,000–5,000)	5,000 (3,000–7,000)	4,000 (2,000–6,000)
MIPS-32	3,000 (3,000–3,000)	4,000 (4,000–5,000)	4,000 (3,000–4,000)	5,000 (4,000–7,000)	4,000 (2,000–6,000)
MIPS-64	3,000 (3,000–3,000)	5,000 (4,000–5,000)	4,000 (3,000–5,000)	6,000 (4,000–8,000)	4,000 (2,000–6,000)
MIPSEB-32	3,000 (3,000–3,000)	4,000 (4,000–5,000)	4,000 (3,000–4,000)	5,000 (4,000–7,000)	4,000 (2,000–6,000)
MIPSEB-64	3,000 (3,000–3,000)	5,000 (4,000–5,000)	4,000 (3,000–5,000)	6,000 (4,000–8,000)	4,000 (2,000–6,000)

Tabla 6.13: Operaciones SSA, mediana (IQR), por arquitectura y *backend*.

Arquitectura	BNIL (razón de expansión)			VEX IR (densidad)
	LLIL	MLIL	HLIL	VEX IR
x86-32	1,358 (1,171–1,500)	1,250 (1,000–1,525)	1,056 (1,000–1,333)	1,000 (0,820–1,333)
x86-64	1,500 (1,125–2,106)	1,245 (1,000–1,507)	1,062 (1,000–1,320)	2,000 (0,946–7,000)
ARM-32	1,271 (1,000–1,497)	1,178 (1,000–1,412)	1,000 (1,000–1,296)	1,230 (1,092–1,625)
ARM-64	1,285 (1,000–2,504)	1,247 (1,000–1,483)	1,083 (1,000–1,329)	1,031 (1,000–1,125)
MIPS-32	1,267 (1,000–1,655)	1,270 (1,000–1,500)	1,103 (1,000–1,353)	0,907 (0,844–1,000)
MIPS-64	1,404 (1,000–2,292)	1,267 (1,000–1,500)	1,100 (1,000–1,333)	0,951 (0,857–1,085)
MIPSEB-32	1,268 (1,000–1,655)	1,269 (1,000–1,500)	1,100 (1,000–1,353)	0,906 (0,844–1,000)
MIPSEB-64	1,403 (1,000–2,287)	1,267 (1,000–1,500)	1,100 (1,000–1,333)	0,952 (0,857–1,086)

Tabla 6.14: Tiempo de lifting (interno), mediana (IQR) en segundos, por arquitectura y *backend*.

Arquitectura	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
x86-32	245,684 (107,790–400,322)	83,381 (32,167–170,835)	97,082 (32,691–168,715)	90,555 (54,110–175,184)	73,470 (25,717–98,082)	42,688 (15,427–67,895)	27,865 (9,684–59,646)	76,162 (24,309–167,777)
x86-64	191,935 (84,433–420,483)	62,218 (21,241–170,194)	84,965 (21,803–169,221)	67,140 (41,204–169,055)	63,432 (15,178–104,652)	23,836 (9,297–63,236)	22,513 (6,195–74,822)	79,727 (16,281–182,071)
ARM-32	263,926 (127,986–382,386)	153,400 (32,332–514,249)	161,695 (44,729–513,095)	167,798 (54,768–565,138)	40,489 (25,570–92,580)	22,294 (11,817–58,343)	61,689 (13,360–96,124)	75,907 (40,939–227,908)
ARM-64	222,724 (87,078–393,334)	81,521 (36,003–192,298)	91,009 (51,174–189,935)	113,810 (59,074–206,148)	35,512 (20,041–79,443)	23,346 (11,792–56,084)	18,252 (10,678–46,934)	78,729 (39,168–204,628)
MIPS-32	309,454 (107,087–520,695)	90,685 (37,214–210,837)	88,144 (46,096–200,470)	120,248 (52,997–196,522)	48,127 (19,855–107,442)	24,938 (13,153–76,049)	23,853 (8,938–62,892)	106,082 (52,338–278,212)
MIPS-64	301,660 (121,659–519,040)	113,478 (48,380–230,293)	106,057 (57,491–237,910)	111,629 (66,392–216,685)	86,368 (39,660–138,185)	23,725 (13,801–55,687)	N/D	93,525 (44,815–204,304)
MIPSEB-32	294,583 (107,583–533,695)	91,740 (43,105–222,834)	91,070 (38,249–210,725)	101,409 (62,545–207,941)	48,730 (19,839–107,147)	23,708 (11,798–72,327)	24,075 (8,233–62,677)	107,228 (51,065–290,313)
MIPSEB-64	298,321 (123,928–586,806)	110,046 (48,262–242,018)	130,595 (61,474–254,703)	107,792 (73,332–228,870)	99,736 (39,117–142,364)	24,132 (16,131–52,827)	N/D	93,695 (34,500–210,536)

Tabla 6.15: Memoria pico en MiB, mediana (IQR), por arquitectura y *backend*.

Arquitectura	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
x86-32	1320,5 (1136,5–1454,0)	1616,2 (1368,2–1778,7)	1511,0 (1346,0–1758,8)	1352,5 (1247,6–1478,3)	371,5 (258,9–598,7)	123,5 (54,7–167,7)	749,0 (384,0–1569,3)	250,4 (186,1–486,4)
x86-64	1337,4 (1097,4–1458,7)	1575,6 (1337,9–1790,1)	1561,8 (1324,5–1800,9)	1324,8 (1258,7–1456,4)	350,0 (229,2–644,8)	72,9 (47,7–148,7)	634,0 (253,3–1490,7)	245,9 (160,0–531,0)
ARM-32	1350,7 (1040,5–1450,7)	1529,1 (1251,2–1891,0)	1539,3 (1233,7–1946,9)	1273,4 (1160,2–1841,8)	367,3 (258,5–560,5)	85,6 (60,3–150,2)	610,6 (359,6–1168,6)	246,8 (186,6–437,1)
ARM-64	1410,5 (1209,4–1515,8)	1467,0 (1399,2–1649,6)	1458,7 (1363,9–1645,6)	1229,5 (862,3–1969,7)	348,4 (245,1–566,3)	74,3 (53,1–141,3)	523,6 (271,1–1127,1)	272,7 (203,5–544,6)
MIPS-32	1349,3 (1176,0–1490,0)	1739,5 (1359,8–1937,7)	1375,8 (1269,4–1791,1)	1778,2 (1609,1–1812,5)	367,6 (255,6–595,2)	84,8 (59,5–173,1)	624,7 (323,7–1385,8)	259,3 (193,6–552,1)
MIPS-64	1401,7 (1328,9–1475,3)	1827,9 (1607,1–1964,3)	1598,2 (1467,7–1832,2)	1602,4 (1478,6–1796,5)	414,8 (331,0–644,0)	76,9 (52,1–132,5)	N/D	240,1 (186,8–452,9)
MIPSEB-32	1379,9 (1248,0–1486,3)	1754,2 (1398,6–1886,9)	1391,0 (1248,0–1752,7)	1409,3 (1273,1–1811,9)	365,8 (256,1–588,8)	84,9 (59,0–172,3)	624,2 (323,3–1389,3)	260,6 (194,2–557,4)
MIPSEB-64	1403,9 (1326,2–1483,2)	1784,1 (1525,6–1927,6)	1724,5 (1530,9–1858,4)	1532,5 (1478,1–1826,0)	412,9 (330,9–627,4)	74,7 (52,0–133,1)	N/D	242,8 (188,4–459,1)

Tabla 6.16: Tasa de error, categoría aritmética, media, por arquitectura y *backend*.

Arquitectura	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
x86-32	0,0056	0,0066	0,0075	0,0078	0,0000	0,0000	0,0000	0,0001
x86-64	0,0141	0,0033	0,0033	0,0034	0,0000	0,0000	0,0000	0,0000
ARM-32	0,0022	0,0683	0,0663	0,0712	0,0018	0,0000	0,0004	0,0011
ARM-64	0,0068	0,0056	0,0058	0,0051	0,0000	0,0000	0,0000	0,0014
MIPS-32	0,0059	0,0024	0,0029	0,0050	0,0000	0,0000	0,0000	0,0028
MIPS-64	0,0064	0,0097	0,0094	0,0097	0,0000	0,0000	N/D	0,0000
MIPSEB-32	0,0042	0,0049	0,0058	0,0056	0,0000	0,0000	0,0000	0,0028
MIPSEB-64	0,0062	0,0109	0,0125	0,0094	0,0000	0,0000	N/D	0,0000

Tabla 6.17: Tasa de error, categoría de control de flujo, media, por arquitectura y *backend*.

Arquitectura	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
x86-32	0,0053	0,0063	0,0071	0,0073	0,0000	0,0000	0,0000	0,0005
x86-64	0,0089	0,0034	0,0034	0,0035	0,0000	0,0000	0,0000	0,0000
ARM-32	0,0023	0,0683	0,0661	0,0711	0,0023	0,0000	0,0003	0,0009
ARM-64	0,0057	0,0050	0,0052	0,0049	0,0000	0,0000	0,0000	0,0016
MIPS-32	0,0064	0,0022	0,0026	0,0050	0,0000	0,0000	0,0000	0,0036
MIPS-64	0,0080	0,0098	0,0098	0,0099	0,0000	0,0000	N/D	0,0000
MIPSEB-32	0,0048	0,0044	0,0054	0,0052	0,0000	0,0000	0,0000	0,0036
MIPSEB-64	0,0076	0,0110	0,0133	0,0094	0,0000	0,0000	N/D	0,0000

Tabla 6.18: Tasa de error, categoría de acceso a memoria, media, por arquitectura y *backend*.

Arquitectura	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
x86-32	0,0054	0,0068	0,0078	0,0080	0,0000	0,0000	0,0000	0,0002
x86-64	0,0064	0,0047	0,0047	0,0048	0,0000	0,0000	0,0000	0,0000
ARM-32	0,0022	0,0734	0,0707	0,0763	0,0027	0,0000	0,0003	0,0009
ARM-64	0,0045	0,0068	0,0069	0,0059	0,0000	0,0000	0,0000	0,0012
MIPS-32	0,0066	0,0030	0,0035	0,0060	0,0000	0,0000	0,0000	0,0028
MIPS-64	0,0062	0,0098	0,0096	0,0098	0,0000	0,0000	N/D	0,0000
MIPSEB-32	0,0050	0,0056	0,0064	0,0065	0,0000	0,0000	0,0000	0,0028
MIPSEB-64	0,0060	0,0108	0,0129	0,0089	0,0000	0,0000	N/D	0,0000

Tabla 6.19: Tasa de error, categoría otras, media, por arquitectura y *backend*.

Arquitectura	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
x86-32	0,0034	0,0041	0,0048	0,0051	0,0000	0,0000	0,0000	0,0002
x86-64	0,0109	0,0034	0,0034	0,0035	0,0000	0,0000	0,0000	0,0000
ARM-32	0,0020	0,0742	0,0721	0,0779	0,0019	0,0000	0,0001	0,0009
ARM-64	0,0069	0,0051	0,0055	0,0050	0,0000	0,0000	0,0000	0,0014
MIPS-32	0,0067	0,0019	0,0023	0,0047	0,0000	0,0000	0,0000	0,0033
MIPS-64	0,0073	0,0069	0,0071	0,0071	0,0000	0,0000	N/D	0,0000
MIPSEB-32	0,0049	0,0040	0,0050	0,0049	0,0000	0,0000	0,0000	0,0033
MIPSEB-64	0,0075	0,0080	0,0105	0,0066	0,0000	0,0000	N/D	0,0000

Tabla 6.20: Fracción de escape, media, por arquitectura y *backend*.

Arquitectura	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	H-R Microcode
x86-32	0,00038	0,00050	0,00036	0,00016	0,00000	0,00486	0,00157
x86-64	0,00037	0,00052	0,00029	0,00012	0,00000	0,00056	0,00292
ARM-32	0,00007	0,00000	0,00000	0,00000	0,00000	0,02968	0,00078
ARM-64	0,00034	0,00035	0,00034	0,00012	0,00000	0,00333	0,00259
MIPS-32	0,00075	0,00183	0,00225	0,00293	0,00004	0,02467	0,00098
MIPS-64	0,00091	0,00143	0,00191	0,00211	0,00003	0,02225	0,00084
MIPSEB-32	0,00075	0,00183	0,00224	0,00299	0,00004	0,02503	0,00103
MIPSEB-64	0,00095	0,00146	0,00199	0,00220	0,00003	0,02309	0,00088

Tabla 6.21: Similitud ponderada de Jaccard, mediana, por par de arquitectura pura y *backend*.

Par	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
ARM-32 vs MIPS-32	0,862	0,621	0,805	0,789	0,763	0,263	0,853	0,902
ARM-32 vs MIPSEB-32	0,852	0,600	0,806	0,805	0,762	0,263	0,844	0,897
ARM-32 vs x86-32	0,915	0,667	0,829	0,835	0,692	0,359	0,824	0,931
MIPS-32 vs MIPSEB-32	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
MIPS-32 vs x86-32	0,803	0,625	0,774	0,762	0,661	0,191	0,798	0,889
MIPSEB-32 vs x86-32	0,800	0,624	0,775	0,778	0,661	0,191	0,791	0,881
ARM-64 vs MIPS-64	0,874	0,701	0,795	0,792	0,734	0,354	N/D	0,845
ARM-64 vs MIPSEB-64	0,867	0,698	0,795	0,791	0,734	0,354	N/D	0,840
ARM-64 vs x86-64	0,960	0,659	0,873	0,909	0,744	0,407	0,916	0,971
MIPS-64 vs MIPSEB-64	1,000	1,000	1,000	1,000	1,000	1,000	N/D	1,000
MIPS-64 vs x86-64	0,866	0,488	0,774	0,780	0,703	0,218	N/D	0,846
MIPSEB-64 vs x86-64	0,857	0,487	0,773	0,779	0,703	0,218	N/D	0,841

Tabla 6.22: Similitud de Jensen-Shannon, mediana, por par de arquitectura pura y *backend*.

Par	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
ARM-32 vs MIPS-32	0,850	0,694	0,821	0,791	0,822	0,340	0,874	0,873
ARM-32 vs MIPSEB-32	0,840	0,694	0,821	0,800	0,821	0,340	0,868	0,866
ARM-32 vs x86-32	0,894	0,737	0,859	0,831	0,859	0,518	0,862	0,919
MIPS-32 vs MIPSEB-32	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
MIPS-32 vs x86-32	0,808	0,727	0,813	0,777	0,805	0,266	0,847	0,864
MIPSEB-32 vs x86-32	0,802	0,727	0,813	0,788	0,804	0,266	0,843	0,856
ARM-64 vs MIPS-64	0,838	0,789	0,801	0,775	0,827	0,403	N/D	0,802
ARM-64 vs MIPSEB-64	0,833	0,788	0,800	0,773	0,826	0,404	N/D	0,797
ARM-64 vs x86-64	0,941	0,771	0,877	0,882	0,825	0,540	0,910	0,962
MIPS-64 vs MIPSEB-64	1,000	1,000	1,000	1,000	1,000	1,000	N/D	1,000
MIPS-64 vs x86-64	0,835	0,702	0,795	0,768	0,772	0,304	N/D	0,804
MIPSEB-64 vs x86-64	0,829	0,702	0,794	0,768	0,772	0,305	N/D	0,801

Tabla 6.23: Coeficiente de variación del tamaño de la IR, mediana, por par de arquitectura pura y *backend*.

Par	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
ARM-32 vs MIPS-32	0,039	0,109	0,083	0,057	0,102	0,178	0,047	0,012
ARM-32 vs MIPSEB-32	0,041	0,109	0,083	0,060	0,102	0,178	0,050	0,015
ARM-32 vs x86-32	0,026	0,116	0,079	0,046	0,218	0,361	0,052	0,014
MIPS-32 vs MIPSEB-32	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000
MIPS-32 vs x86-32	0,067	0,149	0,119	0,088	0,191	0,223	0,067	0,020
MIPSEB-32 vs x86-32	0,068	0,150	0,120	0,091	0,191	0,222	0,071	0,023
ARM-64 vs MIPS-64	0,040	0,157	0,071	0,052	0,152	0,296	N/D	0,022
ARM-64 vs MIPSEB-64	0,043	0,157	0,071	0,052	0,153	0,296	N/D	0,023
ARM-64 vs x86-64	0,007	0,143	0,067	0,010	0,151	0,240	0,027	0,000
MIPS-64 vs MIPSEB-64	0,000	0,000	0,000	0,000	0,000	0,000	N/D	0,000
MIPS-64 vs x86-64	0,048	0,278	0,114	0,067	0,130	0,209	N/D	0,020
MIPSEB-64 vs x86-64	0,051	0,280	0,115	0,067	0,129	0,209	N/D	0,021

Tabla 6.24: Delta de complejidad ciclomática, media, por par de arquitectura pura y *backend*.

Par	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
ARM-32 vs MIPS-32	1,587	2,646	2,637	2,326	2,673	4,443	2,852	1,344
ARM-32 vs MIPSEB-32	1,601	2,638	2,614	2,348	2,674	4,435	2,885	1,349
ARM-32 vs x86-32	1,158	2,208	2,212	2,100	1,774	10,322	2,089	0,564
MIPS-32 vs MIPSEB-32	0,094	0,066	0,066	0,103	0,061	0,096	0,423	0,086
MIPS-32 vs x86-32	1,867	1,729	1,723	1,661	1,850	10,520	2,951	1,371
MIPSEB-32 vs x86-32	1,864	1,734	1,727	1,664	1,833	10,515	2,952	1,337
ARM-64 vs MIPS-64	1,486	1,352	1,352	1,425	3,045	2,006	N/D	2,951
ARM-64 vs MIPSEB-64	1,465	1,366	1,362	1,434	3,042	2,006	N/D	2,957
ARM-64 vs x86-64	1,161	1,262	1,262	1,247	1,683	7,249	3,721	0,547
MIPS-64 vs MIPSEB-64	0,023	0,047	0,045	0,046	0,039	0,010	N/D	0,016
MIPS-64 vs x86-64	1,528	1,315	1,317	1,401	3,442	8,044	N/D	2,839
MIPSEB-64 vs x86-64	1,508	1,334	1,331	1,411	3,438	8,045	N/D	2,844

Tabla 6.25: Similitud ponderada de Jaccard, mediana, por par de anchura pura y *backend*.

Par	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
ARM-32 vs ARM-64	0,895	0,681	0,833	0,856	0,717	0,381	0,878	0,897
MIPS-32 vs MIPS-64	0,875	0,800	0,880	0,833	0,713	0,740	N/D	0,912
MIPSEB-32 vs MIPSEB-64	0,875	0,811	0,889	0,847	0,713	0,740	N/D	0,909
x86-32 vs x86-64	0,957	0,793	0,885	0,908	0,702	0,560	0,875	0,978

Tabla 6.26: Similitud de Jensen-Shannon, mediana, por par de anchura pura y *backend*.

Par	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
ARM-32 vs ARM-64	0,856	0,725	0,848	0,843	0,703	0,456	0,855	0,864
MIPS-32 vs MIPS-64	0,836	0,838	0,868	0,816	0,710	0,686	N/D	0,865
MIPSEB-32 vs MIPSEB-64	0,836	0,842	0,874	0,823	0,710	0,686	N/D	0,860
x86-32 vs x86-64	0,930	0,863	0,906	0,885	0,693	0,642	0,863	0,967

Tabla 6.27: Coeficiente de variación del tamaño de la IR, mediana, por par de anchura pura y *backend*.

Par	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
ARM-32 vs ARM-64	0,030	0,073	0,078	0,040	0,109	0,145	0,038	0,016
MIPS-32 vs MIPS-64	0,048	0,083	0,035	0,038	0,115	0,056	N/D	0,000
MIPSEB-32 vs MIPSEB-64	0,047	0,083	0,034	0,040	0,115	0,056	N/D	0,000
x86-32 vs x86-64	0,014	0,084	0,061	0,029	0,083	0,207	0,034	0,000

Tabla 6.28: Delta de complejidad ciclomática, media, por par de anchura pura y *backend*.

Par	P-code	LLIL	MLIL	HLIL	VEX IR	ESIL	LLVM IR	H-R Microcode
ARM-32 vs ARM-64	1,220	2,565	2,577	2,289	2,255	4,269	3,944	0,890
MIPS-32 vs MIPS-64	1,503	1,490	1,488	1,262	3,390	0,740	N/D	3,121
MIPSEB-32 vs MIPSEB-64	1,503	1,491	1,472	1,252	3,384	0,744	N/D	3,129
x86-32 vs x86-64	0,857	0,660	0,660	0,756	0,782	2,961	1,053	0,458

6.3. Discusión comparativa de las IR

La discusión se organiza en torno a los patrones transversales que emergen al cruzar los cuatro bloques de métricas entre sí, en lugar de repetir bloque a bloque los resultados ya presentados.

El nivel de abstracción no predice la verbosidad, pero dentro de la familia BNIL sí se asocia con un mayor agnosticismo. Las métricas de verbosidad mostraron que HLIL y Hex-Rays Microcode, las dos representaciones de mayor nivel de abstracción asignado, son también las de menor ratio de expansión, mientras que VEX IR y ESIL, de nivel medio y bajo-medio, producen la IR más voluminosa. Esta disociación entre abstracción y verbosidad no se traslada, sin embargo, al agnosticismo interarquitectura dentro de la propia familia de BNIL. Sus tres capas comparten herramienta, proceso de *lifting*, corpus y arquitecturas, y constituyen una progresión controlada de niveles de abstracción. En prácticamente todos los pares de arquitectura pura, LLIL, la capa menos abstracta obtiene la similitud de Jaccard y de Jensen-Shannon más baja de las tres, seguida de MLIL, y HLIL alcanza la más alta en la práctica totalidad de los pares, con excepciones puntuales. Esta progresión, aislada de cualquier diferencia de *lifter* o de corpus, respalda la hipótesis cualitativa de que una mayor abstracción facilita la portabilidad del análisis entre arquitecturas, incluso cuando esa misma abstracción no se traduce en una IR más compacta. Esta conclusión, no obstante, debe restringirse a la comparación interna de BNIL: entre las representaciones intermedias del conjunto cambian simultáneamente el nivel de abstracción, el *lifter*, el modelo de memoria y el *pipeline* de transformaciones, por lo que el nivel de abstracción por sí solo no puede aislarse como predictor general del agnosticismo fuera de esta comparación controlada.

ESIL confirma su perfil cualitativo de baja abstracción, con un peor desempeño cuantitativo en las métricas de agnosticismo, aunque con una salvedad sobre cómo se mide su expresividad. Recuérdese que ESIL es una traducción directa de la semántica nativa, sin abstraer registros ni pila. Esta cercanía al conjunto de instrucciones original se traduce, en el bloque de agnosticismo, en el peor desempeño del conjunto en las cuatro métricas: la similitud de Jaccard más baja de los ocho *backends* en la práctica totalidad de los pares de arquitectura pura, y el delta de complejidad ciclométrica más alto con diferencia. En este caso concreto, la traducción directa y la ausencia de normalización se asocian a un agnosticismo bajo; esto no debe leerse como una imposibilidad general de que una IR combine ambas propiedades, sino como lo observado para esta implementación y este corpus. ESIL no es, en cambio, deficiente en términos absolutos: la sección de coste la sitúa como la representación más

rápida y de menor consumo de memoria del conjunto. Conviene señalar, además, que la fracción de escape de ESIL no mide lo mismo que la del resto de *backends*: al carecer de una construcción de escape genérica, se aproxima mediante la cobertura de instrucciones nativas traducidas, una magnitud distinta que no permite comparar directamente su expresividad frente al resto bajo la misma métrica, y que en ningún caso certifica la fidelidad semántica completa de su traducción. ESIL ilustra así una disyuntiva de diseño real para quien deba elegir una IR: rapidez y ligereza frente a agnosticismo interarquitectura, sin que ningún *backend* del conjunto ofrezca ambas propiedades a la vez en su extremo.

P-code y Hex-Rays Microcode ocupan el extremo opuesto de esa disyuntiva, con matices distintos entre ambos. Los dos *backends* ocupan de forma consistente las primeras posiciones en las cuatro métricas de agnosticismo, coherente con su descripción cualitativa como representaciones de nivel de abstracción alto y medio-alto respectivamente, con un modelo de memoria explícito y una expresividad orientada a preservar la semántica nativa sin renunciar a la normalización. Esta comparación debe leerse con la misma cautela señalada en la Sección 5.3 sobre el delta de complejidad ciclomática: cada *backend* construye su grafo de flujo de control con una noción de bloque y arista propia, no uniforme entre los ocho, por lo que el orden entre P-code y Hex-Rays Microcode en esta métrica concreta refleja la estabilidad de esa construcción particular, no una medida directamente comparable de un CFG nativo común. El coste de este agnosticismo, además, difiere entre ambos *backends*: P-code es, con diferencia, el más lento del conjunto en tiempo de *lifting*, mientras que el *backend* de Hex-Rays Microcode ocupa una posición intermedia. Esta diferencia no estaba anticipada por la comparativa cualitativa, que no establece una relación directa entre el mecanismo de elevación de cada herramienta y su coste computacional, y constituye un hallazgo genuinamente aportado por el marco cuantitativo de este trabajo.

La tasa de error de BNIL, más alta en ARM-32 que en el resto de arquitecturas y *backends*, coincide en este corpus con un coste temporal también más elevado en esa misma combinación, aunque el diseño experimental de este trabajo no permite establecer una relación causal entre ambos fenómenos, sino únicamente su presencia simultánea. El análisis de coste mostró que las tres capas de BNIL incrementan su tiempo de *lifting* en ARM y MIPS frente a x86. Por su parte, las métricas de robustez mostraron que las tasas de error más altas de estas tres capas, dentro del conjunto de valores predominantemente nulos del resto del corpus, se concentran en ARM-32 y se distribuyen de forma similar entre las cuatro categorías de instrucción. Ninguna de estas observaciones se explica directamente por el diseño de BNIL, que

no distingue entre familias de ISA. La coincidencia entre el coste elevado y la tasa de error en ARM-32 apunta a una posible causa compartida en el motor de análisis y decompilación de **Binary Ninja** frente a esta combinación concreta. El experimento no aísla el mecanismo responsable y no permite descartar explicaciones alternativas. Las preocupaciones de expresividad documentadas en la literatura sobre VEX IR no se manifiestan como una tasa de error elevada en este corpus, lo que señala una limitación del propio bloque de robustez más que una ausencia real de problemas. El *backend* de VEX IR basado en **angr** presenta una de las tasas de error más bajas del conjunto en todas las arquitecturas. Esta baja tasa de error operativa no permite concluir que el *lifting* de VEX IR sea semánticamente correcto en todos los casos: la tasa de error de este trabajo mide únicamente fallos de *lifting* a nivel de función (excepciones no controladas, bloqueos), y no captura errores de fidelidad semántica silenciosos, en los que la función se eleva sin excepción alguna pero con un resultado incorrecto. Esta distinción entre robustez operativa y fidelidad semántica es una de las limitaciones de este trabajo.

Esta misma distinción entre robustez operativa y de ejecución completa es necesaria para interpretar correctamente el resto de resultados de este bloque: la tasa de error por categoría de robustez solo se calcula sobre ejecuciones que finalizaron correctamente, y la mayoría de los *backends* muestran, bajo esa definición, tasas de error prácticamente nulas. Esta cifra no debe interpretarse como una robustez elevada a nivel de ejecución completa: **RetDec** no completa el 31,04 % de las ejecuciones del corpus, con un fallo total en MIPS-64 y MIPSEB-64, y **angr** no completa el 4,58 % de las suyas, concentradas en esas mismas dos combinaciones. Ambas cifras conviven con las tasas de error por categoría más bajas del conjunto para ambos *backends*, precisamente porque miden fenómenos distintos: la tasa de error por categoría caracteriza fallos parciales dentro de ejecuciones completadas, mientras que estas cifras caracterizan la propia ejecución no llegando a completarse.

El par de control MIPS/MIPSEB confirma, con la mayor solidez de todo el trabajo, que las cuatro métricas de agnosticismo responden de forma coherente ante un par que difiere únicamente en el orden de bytes. Tal como se estableció en la Sección 4.1, este par se incluyó explícitamente como control metodológico, bajo la hipótesis de que dos binarios que difieren únicamente en *endianness* deberían mostrar una divergencia mínima frente a pares de arquitecturas genuinamente distintas. El resultado, presentado respecto al agnosticismo, confirma esta hipótesis en todos los *backends* con datos disponibles y en las cuatro métricas: una similitud de Jaccard y de Jensen-Shannon de 1,000, un coeficiente de variación de tamaño de 0,000, y el delta de complejidad ciclomática más bajo del conjunto con diferencia. Esta valida-

ción se limita al control de *endianness* concretamente evaluado; no implica, por sí sola, que las cuatro métricas capturen con la misma fidelidad cualquier otro tipo de divergencia semántica entre arquitecturas, dado que, como se ha señalado, el emparejamiento de funciones depende de coincidencia exacta de nombre y la construcción del CFG no es uniforme entre *backends*. Dentro de ese alcance, el resultado respalda de forma directa la decisión metodológica original de BinKit de incluir ambas variantes de *endianness* de MIPS.

La diversidad de diseño CISC/RISC planteada como criterio de selección de arquitecturas se manifiesta de forma recurrente en los resultados, aunque como observación transversal secundaria más que como un patrón único y contundente. La inclusión de x86 junto a ARM y MIPS se justificó precisamente para observar cómo se adapta cada *backend* a instrucciones de complejidad semántica distinta. Esta diversidad aparece de forma recurrente, aunque no unificada, a lo largo de los cuatro bloques: la métrica SSA de VEX IR muestra en x86-64 una dispersión sustancialmente mayor que en el resto de arquitecturas, con un máximo de todo el corpus de 21,6 frente a valores acotados en ARM y MIPS; el coste temporal de BNIL se dispara específicamente en ARM y MIPS frente a x86; y el delta de complejidad ciclomática de VEX IR y Hex-Rays Microcode empeora en los pares de anchura mientras que el resto de *backends* mejora o se mantiene estable, una divergencia posiblemente asociada a diferencias de convención de llamada entre variantes de 32 y 64 bits. Ningún patrón único explica el conjunto completo de estas observaciones, pero su recurrencia respalda la decisión de incluir ambas filosofías de diseño de ISA en el corpus, ya que buena parte de los hallazgos más informativos de este trabajo dependen precisamente de esa diversidad.

En conjunto, estos resultados sugieren que ninguna de las representaciones intermedias estudiadas domina de forma simultánea los cuatro bloques de métricas. Centrando la atención en el agnosticismo interarquitectura, Hex-Rays Microcode se impone como el *backend* más agnóstico del conjunto, liderando tres de las cuatro métricas (similitud de Jaccard, similitud de Jensen-Shannon y coeficiente de variación del tamaño de la IR), con P-code inmediatamente por detrás en esas tres y por delante en la cuarta (delta de complejidad ciclomática), donde ambos ocupan las dos primeras posiciones del conjunto. Esta consistencia entre las cuatro métricas, construidas a partir de fuentes de datos distintas (histogramas de operaciones, tamaño de la IR y forma del grafo de flujo de control), refuerza la fiabilidad de esta conclusión frente a la posibilidad de que fuera un artefacto de una única métrica concreta, si bien las diferencias de construcción del CFG entre *backends* señaladas más arriba recomiendan prudencia específicamente sobre la cuarta.

Ese liderazgo, sin embargo, no es gratuito: P-code, en particular, mantiene un agnosticismo interarquitectura muy elevado a costa de ser el *backend* más lento del conjunto en tiempo de *lifting*, mientras que Hex-Rays Microcode logra un agnosticismo globalmente superior con un coste temporal intermedio. ESIL, en el extremo opuesto de ambas dimensiones, minimiza el coste a costa del agnosticismo, con la peor puntuación del conjunto en las cuatro métricas. BNIL ofrece una validación interna clara del efecto del nivel de abstracción sobre el agnosticismo dentro de su propia familia de capas, pero presenta un coste temporal superior en ARM y MIPS, así como una reducción de robustez especialmente marcada en ARM-32, cuya causa no puede establecerse con certeza a partir de este trabajo. VEX IR, pese a una tasa de error operativa baja en este corpus, arrastra limitaciones de fidelidad semántica documentadas en la literatura que el propio diseño del bloque de robustez de este trabajo no está en condiciones de capturar, del mismo modo que RetDec y angr, presentan tasas de fallo de ejecución completa muy superiores al resto del conjunto pese a sus bajas tasas de error por categoría.

La elección de una representación intermedia para un caso de uso concreto depende, por tanto, de cuál de estas cuatro dimensiones (verbosidad, coste, robustez y agnosticismo) resulte prioritaria para dicho caso; para quien priorice específicamente el agnosticismo interarquitectura, sin embargo, los resultados de este trabajo señalan a Hex-Rays Microcode y P-code como las opciones mejor respaldadas por la evidencia cuantitativa recogida, dentro de las limitaciones de alcance señaladas a lo largo de esta discusión y resumidas a continuación.

6.4. Limitaciones

Las limitaciones se agrupan en tres bloques: las asociadas al bloque de robustez, las derivadas de la cobertura y comparabilidad de las métricas, y las relacionadas con el entorno y la medición de recursos.

El bloque de robustez no captura fallos de ejecución completa: la tasa de error por categoría solo considera funciones de ejecuciones completadas, por lo que un *timeout* o una excepción no controlada excluye la ejecución del cómputo y no se refleja en las tablas del Capítulo 6. Un análisis puntual de estas ejecuciones, no integrado formalmente en el *pipeline*, muestra que RetDec presenta un 31,04 % de fallos (298 de 960), concentrados en MIPS-64 y MIPSEB-64, con un 100 % de fallos por rechazo explícito de ambas arquitecturas, y en ARM-64, con un 48,3 % debido principalmente a instrucciones NEON/SIMD no reconocidas durante el *lifting*. angr presenta un 4,58 % de fallos, concentrados en MIPS-64 y MIPSEB-64, e IDA Pro un 0,62 % (ma-

yoritariamente por *timeouts* en ARM-32 y x86-32). Se plantea como trabajo futuro incorporar una tasa de fallo a nivel de ejecución, con desglose `by_backend/by_arch` para los ocho *lifters*.

Una segunda limitación del bloque de robustez es que no captura errores de fidelidad semántica en funciones que sí se elevan sin excepción. VEX IR presenta una de las tasas de error más bajas del conjunto pese a que la literatura documenta problemas de fidelidad semántica concretos en PyVEX, incluyendo traducciones que alteran el estado de la CPU de forma no definida por la instrucción original. Estas traducciones no lanzan ninguna excepción y, por tanto, no se clasifican como fallo bajo ninguna de las cuatro categorías del bloque de robustez: una función puede completarse sin error y, aun así, contener una traducción incorrecta. Verificar la fidelidad semántica de los ocho *backends* exigiría un marco de validación independiente, ajeno al alcance de este trabajo.

Por último, la granularidad a nivel de función de la tasa de error no distingue el origen del fallo. Tal como se señaló en la definición de las métricas de robustez, una función se clasifica como fallida en su totalidad si el proceso de elevación lanza una excepción, sin que el bloque de robustez permita determinar si dicho fallo se debe a una única instrucción problemática dentro de una función por lo demás correcta, o a una incompatibilidad estructural de la función completa con el *backend* correspondiente.

En cuanto a la cobertura y comparabilidad de las métricas, no todas se calculan sobre la totalidad de los ocho *backends*. La profundidad de anidamiento se reporta únicamente para VEX IR, las capas principales de BNIL y Hex-Rays Microcode, los *backends* para los que se instrumentó el cómputo de esta métrica. La métrica de SSA se calcula únicamente para BNIL y VEX IR, con P-code y LLVM IR excluidos por producir un valor trivialmente constante dadas las decisiones de extracción de este *pipeline*, y Hex-Rays Microcode excluido por el coste de implementar una segunda vía de extracción sobre `gen_microcode()`. La fracción de escape excluye a RetDec en su totalidad, al no haberse podido confirmar una construcción de escape única y consistente en su salida. Estas exclusiones están documentadas y justificadas individualmente, pero limitan la comparabilidad directa entre los ocho *backends* para estas tres métricas concretas.

RetDec carece de cobertura para MIPS de 64 bits. Esta limitación de la herramienta se manifiesta de forma sistemática en las tablas de resultados: toda celda de LLVM IR correspondiente a MIPS-64 o MIPSEB-64, o a cualquier par de agnosticismo que involucre alguna de estas dos combinaciones, queda marcada como no disponible. Ninguna de las siete alternativas de *lifter* evaluadas resuelve esta limitación sin in-

introducir un problema distinto, por lo que se trata de un hueco real del entorno de herramientas disponibles, no de una elección subóptima de este trabajo.

El delta de complejidad ciclomática se calcula sobre construcciones de grafo de flujo de control que no son idénticas entre *backends*. Recuérdese que cada *backend* emplea su propia noción de bloque y arista a nivel de IR o decompilador (el grafo de bloques de BNIL, el **CFGFast** de VEX IR, el grafo de flujo de Hex-Rays Microcode, entre otros), consistente dentro de cada uno pero no uniforme entre los ocho. Las comparaciones de esta métrica entre *backends* deben interpretarse, por tanto, como diferencias en la estabilidad de la forma del CFG tal como cada *backend* la define, no como una medida de un grafo de control nativo común.

El emparejamiento de funciones del bloque de agnosticismo se basa en la coincidencia exacta de nombres, sin coincidencias aproximadas. Este criterio puede reducir el número de funciones comparables en binarios muy optimizados cuando el compilador aplica transformaciones diferentes según la arquitectura de destino, limitando el conjunto sobre el que se calculan las cuatro métricas del bloque.

El recuento de funciones de VEX IR no es directamente comparable al del resto de *backends*. La reconstrucción de límites de función mediante **CFGFast** sobreestima sistemáticamente el número de funciones, entre dos y tres veces más que el resto. Esta sobreestimación no afecta a las métricas por función (verbosidad, coste, robustez), pero sí compromete la comparabilidad del emparejamiento en el bloque de agnosticismo: si el exceso varía entre arquitecturas, la tasa de emparejamiento de **angr** no sería directamente comparable, un efecto que este trabajo no cuantifica.

En relación con el entorno y la medición de recursos, la medición de memoria de ESIL puede subestimar el consumo real cuando ambos procesos, Python y **radare2**, mantienen una ocupación de memoria significativa de forma simultánea. La combinación de los picos de memoria de ambos procesos se realiza mediante el máximo de sus valores independientes, heredado por la semántica de *rusage* de procesos hijo del sistema operativo, y no mediante su suma; si ambos picos no coinciden en el mismo instante, la cifra reportada puede infravalorar el consumo conjunto real.

El entorno de ejecución es una máquina virtual cuya disponibilidad efectiva de recursos no puede determinarse por completo desde el sistema invitado. Mecanismos de configuración del hipervisor, como el límite de ejecución de CPU o el *ballooning* de memoria de VirtualBox, se configuran desde el sistema anfitrión sin que sus valores queden necesariamente expuestos a la máquina virtual. El robo de tiempo de CPU y la contención con otras máquinas virtuales, en cambio, sí pueden dejar indicios medibles desde el sistema invitado. Una comprobación puntual mediante esta vía no mostró indicios de ninguno de estos fenómenos, aunque se trata de una instantánea

y no de una garantía sostenida durante los varios días que ocupó la ejecución completa del corpus. Por tanto, no puede descartarse por completo su influencia sobre las cifras del bloque de coste.

Capítulo 7

Conclusiones y trabajo futuro

Este TFM ha comparado seis representaciones intermedias de uso extendido en el análisis estático de binarios (P-code, BNIL y sus capas internas, VEX IR, ESIL, LLVM IR y Hex-Rays Microcode) mediante un doble enfoque cualitativo y cuantitativo. La comparativa cualitativa estableció catorce dimensiones de comparación, desde el nivel de abstracción y la forma normal hasta la madurez y la facilidad de uso, y evidenció que la elección de arquitecturas y herramientas condiciona de forma directa qué aspectos de cada IR resultan observables. La comparativa cuantitativa definió cuatro bloques de métricas (verbosidad y estructura, coste, robustez y agnosticismo interarquitectura) y las ejecutó mediante un *pipeline* de extracción propio sobre un corpus de 960 binarios compilados para ocho combinaciones de arquitectura y anchura de registro, con seis compiladores y cuatro niveles de optimización. Los resultados obtenidos confirmaron varias expectativas derivadas del análisis cualitativo, como la mayor invarianza interarquitectura de las representaciones de mayor nivel de abstracción dentro de la propia familia de BNIL, pero también revelaron patrones no anticipados, como la disociación entre nivel de abstracción y verbosidad, el coste temporal elevado de BNIL en ARM y MIPS y una reducción de robustez especialmente marcada en ARM-32; así como una proporción considerable de ejecuciones que no llegan a completarse en **RetDec** (31,04 %) y, en menor medida, en **angr** (4,58 %).

Destaca también la identificación de Hex-Rays Microcode y P-code como los *backends* con mayor agnosticismo interarquitectura del conjunto, ocupando de forma consistente las primeras posiciones en las cuatro métricas del bloque (Hex-Rays Microcode liderando tres de ellas, P-code la restante), y la validación empírica del bloque de agnosticismo mediante el par de control MIPS/MIPSEB. Este par confirmó en todos los *backends* con datos disponibles el comportamiento esperado ante

una variación exclusiva de orden de bytes.

Este trabajo presenta, no obstante, varias limitaciones que abren líneas concretas de trabajo futuro. La más inmediata es la incorporación de una métrica formal de fallos catastróficos de ejecución completa, con el mismo desglose `by_backend/by_arch` que el resto de bloques, actualmente cubierta solo mediante el análisis puntual ya mencionado. En la misma línea, la ampliación de la métrica de SSA a Hex-Rays Microcode, mediante una segunda vía de extracción sobre `gen_microcode()` en lugar de la API de decompilación de alto nivel empleada en este trabajo, completaría la cobertura de esta métrica a las tres representaciones intermedias (BNIL, VEX IR y Hex-Rays Microcode) cuya forma de SSA admite variación medible. Una tercera línea, de mayor alcance, es el desarrollo de un marco de validación de fidelidad semántica que complementa al bloque de robustez actual, capaz de distinguir errores operativos de traducciones semánticamente incorrectas que no lanzan ninguna excepción.

Finalmente, el propio corpus de este trabajo admite una extensión directa: de los nueve compiladores que ofrece BinKit, este trabajo emplea seis, dejando fuera `gcc-5.5.0`, `gcc-7.3.0` y `clang-5.0`. La ampliación a los 1 440 binarios resultantes de incluir los nueve compiladores permitiría, entre otras cosas, comprobar si los patrones de error observados en las pruebas preliminares de este trabajo con un compilador concreto se sostienen a escala completa. Asimismo, sería de interés procesar binarios de otros formatos además de ELF, binarios de Windows o de macOS, para evaluar si los patrones observados en este trabajo se generalizan a entornos distintos a Unix. Complementariamente, la incorporación de mecanismos de coincidencia aproximada en el emparejamiento de funciones del bloque de agnosticismo permitiría evaluar en qué medida el emparejamiento actual, basado en coincidencia exacta de nombre, subestima el agnosticismo real de las representaciones intermedias sobre binarios muy optimizados.

Durante el desarrollo de este Trabajo Fin de Máster se han utilizado herramientas de inteligencia artificial generativa, concretamente Claude (Anthropic), principalmente para mejorar la redacción de algunos apartados y resolver errores de código surgidos durante el desarrollo del *pipeline* de extracción y cálculo de métricas. Esta herramienta se ha utilizado únicamente como apoyo y no ha sustituido el trabajo realizado por el autor, incluida la implementación, ejecución y validación del *pipeline* experimental. El autor mantiene plena responsabilidad sobre la originalidad, veracidad, rigor técnico y científico, metodología y autoría del contenido presentado.

Bibliografía

- [1] Cardama, F.J., Vázquez-Pérez, J., Piñeiro, C., Pichel, J.C., Pena, T.F., Gómez, A.: Review of intermediate representations for quantum computing. *The Journal of Supercomputing* 81(2), 418 (2025)
- [2] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al.: SOK:(State of) The Art of War: Offensive Techniques in Binary Analysis. In: 2016 IEEE symposium on security and privacy (SP). pp. 138–157. IEEE (2016)
- [3] Stanier, J., Watson, D.: Intermediate Representations in Imperative Compilers: A Survey. *ACM Computing Surveys (CSUR)* 45(3), 1–27 (2013)
- [4] Cannan, L., Morris, T.: Lost in Translation: How Intermediate Language Representations Affect Malware Classification. *AI, Computer Science and Robotics Technology* 4 (2025)
- [5] Vector 35: 2024 Reverse Engineering Survey: The State of the Industry. <https://binary.ninja/2024/11/08/user-survey-results.html> (2024), Accedido: 30 de mayo de 2026
- [6] Schulte, E., Dorn, J., Flores-Montoya, A., Ballman, A., Johnson, T.: GTIRB: GrammaTech Intermediate Representation for Binaries. *arXiv preprint arXiv:1907.02859* (2019)
- [7] Andriesse, D., Chen, X., Van Der Veen, V., Slowinska, A., Bos, H.: An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 583–600 (2016)
- [8] Eagle, C.: *The IDA Pro Book*. No Starch Press (2011)
- [9] Pang, C., Yu, R., Chen, Y., Koskinen, E., Portokalidis, G., Mao, B., Xu, J.: SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But

- Were Afraid to Ask. In: 2021 IEEE symposium on security and privacy (SP). pp. 833–851. IEEE (2021)
- [10] Liu, Z., Yuan, Y., Wang, S., Bao, Y.: SoK: Demystifying Binary Lifters Through the Lens of Downstream Applications. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 1100–1119. IEEE (2022)
- [11] Kim, S., Faerevaag, M., Jung, M., Jung, S., Oh, D., Lee, J., Cha, S.K.: Testing intermediate representations for binary analysis. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 353–364. IEEE (2017)
- [12] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, 2 edn. (2006)
- [13] Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A binary analysis platform. In: International Conference on Computer Aided Verification. pp. 463–469. Springer (2011)
- [14] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13(4), 451–490 (1991)
- [15] Jung, M., Kim, S., Han, H., Choi, J., Cha, S.K.: B2R2: Building an Efficient Front-End for Binary Analysis. In: Proceedings of the NDSS Workshop on Binary Analysis Research (2019)
- [16] Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formalizing the llvm intermediate representation for verified program transformations. In: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 427–440 (2012)
- [17] Naus, N., Verbeek, F., Walker, D., Ravindran, B.: A formal semantics for p-code. In: Working Conference on Verified Software: Theories, Tools, and Experiments. pp. 111–128. Springer (2022)
- [18] Dasgupta, S., Dinesh, S., Venkatesh, D., Adve, V.S., Fletcher, C.W.: Scalable validation of binary lifters. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 655–671 (2020)

- [19] Patterson, D.A., Hennessy, J.L.: Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann, 5 edn. (2014)
- [20] Kim, D., Kim, E., Cha, S.K., Son, S., Kim, Y.: Revisiting Binary Code Similarity Analysis Using Interpretable Feature Engineering and Lessons Learned. *IEEE Transactions on Software Engineering* pp. 1–23 (2022)
- [21] Fossbytes: NSA Open Sources Ghidra Reverse Engineering Tool for Linux, Windows. <https://fossbytes.com/nsa-open-sources-ghidra-reverse-engineering-linux-windows/> (2019), Accedido: 6 de agosto de 2026
- [22] Vector 35: Binary Ninja — FAQ. <https://binary.ninja/faq/>, Accedido: 14 de junio de 2026
- [23] Valgrind Developers: Valgrind. <https://valgrind.org/info/>, Accedido: 28 de mayo de 2026
- [24] radareorg: radare2. <https://github.com/radareorg/radare2>, Accedido: 21 de junio de 2026
- [25] Hex-Rays: Discover IDA 9.0: Exciting new features and improvements. <https://hex-rays.com/blog/discover-ida-9.0-exciting-new-features-and-improvements>, Accedido: 21 de junio de 2026
- [26] Guilfanov, I.: Decompiler internals: microcode. <https://i.blackhat.com/us-18/Thu-August-9/us-18-Guilfanov-Decompiler-Internals-Microcode-wp.pdf> (2018), *White paper*, presentado en Black Hat USA 2018 y REcon Brussels 2018. Accedido: 13 de junio de 2026
- [27] LLVM Project: LLVM developer policy. <https://llvm.org/docs/DeveloperPolicy.html>, Accedido: 20 de junio de 2026
- [28] Avast: RetDec: A retargetable machine-code decompiler. <https://github.com/avast/retdec>, Accedido: 26 de mayo de 2026
- [29] rev.ng Labs: The rev.ng decompiler goes open source + start of the UI closed beta. <https://rev.ng/blog/open-sourcing-renvg-decompiler-ui-closed-beta> (2024), Accedido: 27 de mayo de 2026
- [30] rev.ng Labs: rev.ng — roadmap. <https://rev.ng/roadmap> (2024), Accedido: 27 de mayo de 2026
- [31] Trail of Bits: Remill: Library for lifting machine code to LLVM bitcode. <https://github.com/lifting-bits/remill>, Accedido: 26 de mayo de 2026

- [32] Centre for Development of Advanced Computing (C-DAC): Remill (MIPS fork). <https://github.com/CDACesec/remill>, Accedido: 27 de mayo de 2026
- [33] Trail of Bits: McSema: Framework for lifting x86, amd64, aarch64, sparc32, and sparc64 program binaries to LLVM bitcode. <https://github.com/lifting-bits/mcsema>, Accedido: 27 de mayo de 2026
- [34] Trail of Bits, University of California Irvine, and Vrije Universiteit Amsterdam: BinRec: Dynamic binary lifting and recompilation. <https://github.com/trailofbits/binrec-tob>, Accedido: 27 de mayo de 2026
- [35] Galois, Inc.: reopt: A tool for analyzing x86-64 binaries. <https://github.com/GaloisInc/reopt>, Accedido: 27 de mayo de 2026
- [36] Schulte, E., Brown, M.D., Folts, V.: A Broad Comparative Evaluation of x86-64 Binary Rewriters. In: Proceedings of the 15th Workshop on Cyber Security Experimentation and Test. pp. 129–144 (2022)
- [37] Microsoft: llvm-mctoll. <https://github.com/microsoft/llvm-mctoll>, Accedido: 27 de mayo de 2026
- [38] ElykDeer: McNinja: Compile Binary Ninja's MLIL to LLVM. <https://github.com/ElykDeer/McNinja>, Accedido: 27 de mayo de 2026
- [39] Vector 35: BNIL overview. <https://docs.binary.ninja/dev/bnil-overview.html>, Accedido: 14 de junio de 2026
- [40] angr Project: Intermediate representation. <https://docs.angr.io/en/latest/advanced-topics/ir.html>, Accedido: 18 de junio de 2026
- [41] radareorg: ESIL — the radare2 book. <https://book.rada.re/emulation/esil.html>, Accedido: 21 de junio de 2026
- [42] Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: International symposium on code generation and optimization, 2004. CGO 2004. pp. 75–86. IEEE (2004)
- [43] icecr4ck: Notes on Hex-Rays microcode. <https://gist.github.com/icecr4ck/6c744d489efbb07a32bb22e8a3c748e3>, Accedido: 21 de junio de 2026
- [44] Lew, L.: Ghidra decompiler analysis engine. <https://lemuellew.github.io/Ghidra- Decompiler-Analysis-Engine-Document/>, Accedido: 14 de junio de 2026

- [45] Vector 35: Low level IL. <https://docs.binary.ninja/dev/bnil-llil.html>, Accedido: 14 de junio de 2026
- [46] Reno, R.: Static taint analysis using Binary Ninja: A case study of MySQL cluster vulnerabilities. <https://www.zerodayinitiative.com/blog/2022/2/14/static-taint-analysis-using-binary-ninja-a-case-study-of-mysql-cluster-vulnerabilities> (2022), Accedido: 5 de agosto de 2026
- [47] angr Project: libvex_ir.h. https://github.com/angr/vex/blob/master/pub/libvex_ir.h, Accedido: 14 de junio de 2026
- [48] Debian Project: radare2(esil). <https://manpages.debian.org/bookworm-backports/radare2/esil.7.en.html>, Accedido: 25 de mayo de 2026
- [49] Hex-Rays: IDA 9.3 release notes. https://docs.hex-rays.com/release-notes/9_3, Accedido: 8 de julio de 2026
- [50] National Security Agency: P-Code reference manual. https://ghidra.re/ghidra_docs/languages/html/pcoderef.html, Accedido: 6 de mayo de 2026
- [51] Vector 35: Medium level IL. <https://docs.binary.ninja/dev/bnil-mlil.html>, Accedido: 27 de mayo de 2026
- [52] LLVM Project: MemorySSA. <https://llvm.org/docs/MemorySSA.html>, Accedido: 20 de junio de 2026
- [53] Küchler, A., Banse, C.: Representing LLVM-IR in a Code Property Graph. In: International Conference on Information Security. pp. 360–380. Springer (2022)
- [54] Hex-Rays: Spying on decompiler internals: The Hex-Rays microcode viewer. <https://hex-rays.com/blog/spying-on-decompiler-internals-the-hex-rays-microcode-viewer>, Accedido: 4 de agosto de 2026
- [55] LLVM Project: The LLVM target-independent code generator. <https://llvm.org/docs/CodeGenerator.html>, Accedido: 20 de junio de 2026
- [56] Vector 35: High level IL. <https://docs.binary.ninja/dev/bnil-hlil.html>, Accedido: 28 de mayo de 2026
- [57] angr Project: pyvex — binary translator. <https://api.angr.io/pyvex.html>, Accedido: 28 de junio de 2026
- [58] Hex-Rays: Supported processors. <https://docs.hex-rays.com/core/disassembler/specification/supported-processors>, Accedido: 14 de junio de 2026

-
- [59] Help Net Security: Ghidra: Open-source software reverse engineering framework. <https://www.helpnetsecurity.com/2024/06/17/ghidra-nsa-open-source-software-reverse-engineering-framework/> (2024), Accedido: 1 de junio de 2026
- [60] angr Project: Throwing a tantrum, part 4: VEX and Gymrat. http://angr.io/blog/throwing_a_tantrum_part_4/ (2017), Accedido: 29 de mayo de 2026
- [61] Hex-Rays: IDA 9.4 beta is now available. <https://hex-rays.com/blog/ida-9.4-beta-is-now-available> (2026), Accedido: 21 de junio de 2026
- [62] LLVM Project: The new pass manager. <https://llvm.org/docs/NewPassManager.html>, Accedido: 20 de junio de 2026
- [63] National Security Agency: Headless analyzer. <https://static.grumpycoder.net/pixel/support/analyzeHeadlessREADME.html>, Accedido: 4 de julio de 2026
- [64] LLVM Project: LLVM language reference manual. <https://llvm.org/docs/LangRef.html>, Accedido: 20 de junio de 2026
- [65] Elastic Security Labs: Introduction to Hex-Rays decompilation internals. <https://www.elastic.co/security-labs/introduction-to-hexrays-decompilation-internals> (2024), Accedido: 10 de junio de 2026
- [66] Nethercote, N.: Twenty years of Valgrind. <https://nnethercote.github.io/2022/07/27/twenty-years-of-valgrind.html> (2022), Accedido: 28 de agosto de 2026
- [67] angr: angr: A Powerful Binary Analysis Platform, <https://angr.io>, Accedido: 28 de agosto de 2026
- [68] Hex-Rays: IDA: celebrating 30 years of binary analysis innovation (2021), <https://hex-rays.com/blog/ida-celebrating-30-years-of-binary-analysis-innovation>, Accedido: 28 de agosto de 2026
- [69] Radare2: History, <https://book.rada.re/intro/history.html>, Accedido: 28 de agosto de 2026
- [70] pancake: Emulating Code In Radare2. Lacon 2015 (2015), <https://radare.org/get/lacon2k15-esil.pdf>, Accedido: 28 de agosto de 2026

- [71] Lattner, C.: LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL (Dec 2002), <https://llvm.org/pubs/2002-12-LattnerMSThesis-book.pdf>, Accedido: 28 de agosto de 2026
- [72] Vector 35: Binary ninja – features (2024), <https://binary.ninja/features/>, Accedido: 20 de junio de 2026
- [73] Casinghino, C., Paasch, J., Roux, C., Altidor, J., Dixon, M., Jamner, D.: Using Binary Analysis Frameworks: The Case for BAP and angr. In: NASA Formal Methods Symposium. pp. 123–129. Springer (2019)
- [74] McCabe, T.J.: A Complexity Measure. IEEE Transactions on software Engineering SE-2(4), 308–320 (1976)

Anexo A

Tiempo invertido y diagrama de Gantt

En este anexo se tratan las fases de las que se compone este Trabajo de Fin de Máster y el tiempo invertido en cada una de ellas.

El número total de horas invertidas en el desarrollo de este TFM asciende a 372 h, distribuidas entre las tres fases recogidas en la Tabla A.1: 68 h corresponden a la comparativa cualitativa, 222 h a la metodología cuantitativa y 82 h a la elaboración y revisión de la memoria. La Figura A.1 recoge la distribución temporal de estas tres fases y sus tareas a lo largo de los cuatro meses de desarrollo del trabajo, desde el 1 de mayo hasta el 1 de septiembre de 2026.

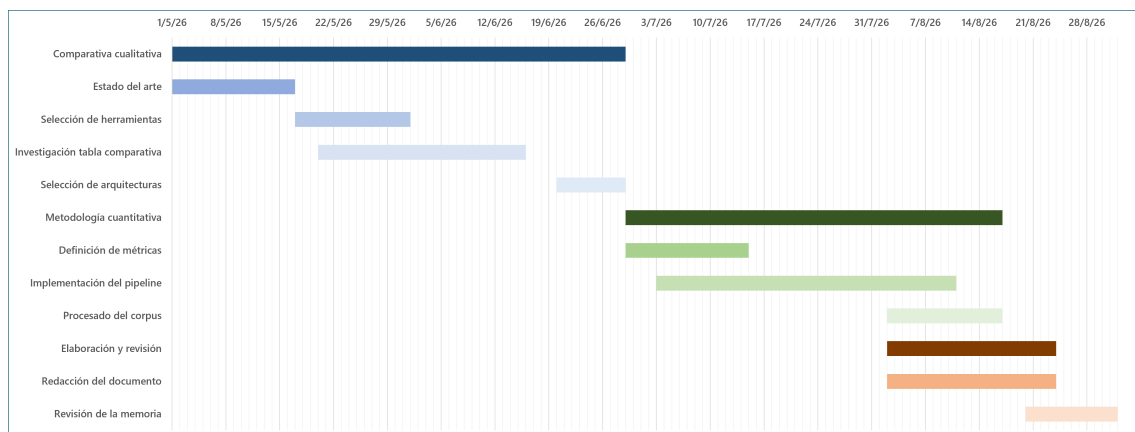


Figura A.1: Diagrama de Gantt con la planificación del proyecto

Tabla A.1: Estimación de horas invertidas por fase y tarea.

ID	Tarea	Horas
Fase 1 — Comparativa cualitativa		
T1.1	Estado del arte	20 h
T1.2	Selección de herramientas y entornos por IR	15 h
T1.3	Investigación para la tabla comparativa	27 h
T1.4	Selección de arquitecturas de conjunto de instrucciones	6 h
Subtotal		68 h
Fase 2 — Metodología cuantitativa		
T2.1	Definición de métricas	12 h
T2.2	Implementación del <i>pipeline</i>	98 h
T2.3	Procesado del corpus y generación de métricas	112 h
Subtotal		222 h
Fase 3 — Elaboración y revisión de la memoria		
T3.1	Redacción del documento	60 h
T3.2	Revisión y corrección de la memoria	22 h
Subtotal		82 h
Total		372 h