ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

DRAGONLIFTER: UN LIFTER DA P-CODE A C

Elaborato in
PROGRAMMAZIONE A OGGETTI

Relatore
Prof. MIRKO VIROLI

Presentata da
SAMUELE TURCI

Correlatore
LUCA TREMAMUNNO

Anno Accademico 2020 – 2021

Sommario

L'analisi di codice compilato è un'attività sempre più richiesta e necessaria, critica per la sicurezza e stabilità delle infrastrutture informatiche utilizzate in tutto il mondo. Le tipologie di file binari da analizzare sono numerose e in costante evoluzione, si può passare da applicativi desktop o mobile a firmware di router o baseband. Scopo della tesi è progettare e realizzare Dragonlifter, un convertitore da codice compilato a C che sia estendibile e in grado di supportare un numero elevato di architetture, sistemi operativi e formati file. Questo rende possibile eseguire programmi compilati per altre architetture, tracciare la loro esecuzione e modificarli per mitigare vulnerabilità o cambiarne il comportamento.

Indice

Introduzione

L'analisi di codice compilato è un'attività sempre più richiesta e necessaria, in quanto molto rilevante in svariati ambiti, critici per la sicurezza e stabilità delle infrastrutture informatiche utilizzate in tutto il mondo. Negli ultimi anni sono stati rilasciati sempre più software per facilitare questa analisi, rendendo possibile effettuare operazioni come l'analisi di malware, la ricerca di vulnerabilità e il monitoraggio di software non attendibile. Le tipologie di file binari da analizzare sono numerose e in costante evoluzione, si può passare da applicativi desktop o mobile a firmware di router o baseband. È quindi utile che le soluzioni sviluppate supportino il maggior numero possibile di architetture, sistemi operativi e formati file.

Uno dei software più utilizzati per l'analisi di codice compilato è Ghidra [1], un software open-source rilasciato nel 2019 dall'NSA [2]. Ghidra supporta numerose architetture e rende facile aggiungere il supporto per nuove. Per evitare di dover adattare il codice di analisi a ogni architettura supportata, Ghidra utilizza un linguaggio intermedio, chiamato p-code. Il p-code è un linguaggio utilizzato per modellare il significato semantico delle istruzioni macchina e permette quindi di generalizzare un binario scritto per una specifica architettura a un insieme di istruzioni comuni.

Questa tesi presenta il design e lo sviluppo di Dragonlifter, un software per binary lifting che si basa sul p-code generato da Ghidra. Con binary lifting si intende la conversione o traduzione di un file binario a un linguaggio di livello più alto che deve essere compilabile e avere lo stesso identico comportamento del programma originale. Il codice generato può essere utilizzato per sviluppare analisi più complesse, tracciare l'esecuzione del programma e inserire controlli per mitigare vulnerabilità.

Tradizionalmente, per ogni architettura che un lifter vuole supportare, è necessario scrivere codice apposito per convertire la semantica di ogni istruzione nel proprio linguaggio target. Tuttavia, basandosi sull'analisi effettuata da Ghidra, Dragonlifter è in grado di supportare automaticamente un numero

elevato di architetture e allo stesso tempo di avere una bassa complessità di implementazione.

Più nello specifico, Dragonlifter è un lifter da p-code a C e consiste di due parti separate: l'estrazione delle informazioni generate dall'analisi di Ghidra, cioè principalmente le istruzioni p-code contenute in ogni funzione, e la conversione (o lifting) di queste informazioni in codice C. La fase di lifting si occupa sia di convertire le istruzioni p-code in C, sia di generare l'infrastruttura necessaria per la simulazione dell'ambiente di esecuzione originale.

Dato che certi comportamenti e istruzioni non sono modellabili da Ghidra, è necessario generare codice apposito per emularli. Per questo Dragonlifter fornisce un sistema di plugin che permette di modificare e estendere tutte le fasi di lifting. I plugin possono anche essere utilizzati per inserire in automatico codice per tracciare l'esecuzione del programma, per esempio per contare quante istruzioni vengono eseguite o quante volte viene chiamata una funzione.

Il codice generato dal lifter viene validato tramite numerosi test. Ogni istruzione p-code viene testata singolarmente, inoltre viene anche verificato che il comportamento di un intero eseguibile liftato sia identico a quello dell'eseguibile originale.

Unito a Ghidra, Dragonlifter permette di convertire in codice C un binario compilato per una qualsiasi architettura supportata, permettendo anche di emulare dettagli specifici del sistema sottostante come le system call.

Capitolo 1

Introduzione all'analisi di codice compilato

In questo capitolo si introduce il concetto di analisi di codice compilato, indicando possibili casi d'uso e la principali tecniche.

1.1 Motivazione

Nella maggior parte dei casi, l'analisi di un programma avviene tramite l'analisi del suo codice sorgente. Tuttavia non sempre è possibile avere accesso al sorgente o a volte è necessario analizzare direttamente il codice prodotto dal compilatore.

Ci possono essere vari motivi per i quali questo è necessario. Per esempio:

- Si ha a disposizione il codice sorgente originale, ma si vuole osservare il codice generato dal compilatore
  - Valutare l'efficienza del codice compilato (e.g., se sono state utilizzate istruzioni SIMD)
  - Verificare quale parte di codice nello specifico causa crash o bug
- Non si è in possesso del codice sorgente originale e si vuole capire più a fondo il comportamento del programma
  - Il programma era di nostra proprietà ma i sorgenti sono andati persi
  - Il binario non è nostro ma si sospetta che possa essere un malware
  - Si vuole comprendere per quale motivo si verifichi un bug o comportamento, per poterlo evitare o sfruttare

1.2 Disassembler

Il tool più di basso livello per l'analisi di codice compilato è il disassembler. Un disassembler si occupa di convertire un insieme di byte rappresentanti codice macchina in una stringa di istruzioni comprensibili da un essere umano. Per esempio, i byte `[0x31, 0xc0]` in un processore con architettura `x86` corrispondono all'istruzione `xor eax, eax`. La rappresentazione testuale di codice macchina viene chiamata assembly.

Per un essere umano è ovviamente molto difficile ricordarsi cosa faccia ogni serie di numeri ma è molto più semplice interpretare l'assembly, quindi è necessario l'ausilio di un disassembler per automatizzare questa conversione. Ogni architettura ha un set di istruzioni diverso e molto spesso anche una rappresentazione assembly differente. È necessario quindi creare un disassembler separato per ognuna.

Disassembler più avanzati sono anche in grado di analizzare l'intero binario ed estrarre più informazioni. Per esempio, cercare di capire quali aree di memoria vengono usate come variabili globali e dove iniziano e finiscono le funzioni.

Alcuni tra i disassembler più noti sono:

- Capstone [3]
  - Supporta un numero molto elevato di architetture
  - Fornisce un API che permette di automatizzare la disassemblazione
- Zydis [4]
  - Supporta solo `x86` e `x86_64`
  - Elevata velocità di decompilazione
- GDB [5]
  - Debugger del progetto GNU
  - Permette, tra le altre cose, di mostrare codice disassemblato
  - Supporta numerosi formati file e architetture

1.3 Decompiler

Esistono tool ancora più avanzati che permettono di decompilare le funzioni all'interno di un file binario. La decompilazione è una tecnica tramite la quale si cerca di ottenere del codice che assomigli il più possibile al codice sorgente originale, avendo a disposizione solo il codice compilato.

Creare un decompiler è molto più difficile rispetto a creare un disassembler in quanto molte informazioni presenti nel codice sorgente vengono perse durante la compilazione. Dato che il decompilatore deve analizzare codice già compilato, dovrà utilizzare euristiche (e/o aiuto da parte di un umano) per tentare di recuperare queste informazioni. Per esempio il concetto di variabili viene parzialmente perso durante la compilazione e il loro tipo può essere predetto solo osservando le operazioni che ci vengono effettuate sopra.

Creare un decompilatore che generi codice soddisfacente è molto complesso e infatti ne esistono pochi. I principali, che convertono da codice macchina a un codice simile a C sono:

- Hex-Rays Decompiler [6] per IDA Pro [7]
  - Esiste da circa 20 anni ed è il più avanzato in commercio
  - Ha un costo elevato ma è professionalmente lo standard de-facto
- Ghidra [1]
  - Reso pubblico e open source nel 2019
  - Supporta molte architetture con una qualità soddisfacente
  - Facile da estendere
- Binary Ninja [8]
  - Rilasciato a luglio 2016, include un decompiler da Maggio 2020
  - È a pagamento ma ha costi ridotti
  - Genera codice peggiore rispetto agli altri, ma riceve frequentemente aggiornamenti e migliora a ogni release
  - Usato principalmente per le sue capacità di scripting

Il fatto che Ghidra sia open source e che renda possibile aggiungere il supporto a nuove architetture in maniera relativamente facile, lo ha reso molto popolare. Questo progetto utilizza Ghidra e sarà quindi soggetto a un'analisi

più approfondita nel prossimo capitolo.

Esistono anche decompilatori per architetture di livello più alto, che convertono per esempio dal bytecode di Java a Java [9] oppure da CIL a C# [10]. Tuttavia in questi casi l'implementazione è più semplice in quanto i binari interpretati da una virtual machine contengono solitamente molte più informazioni utili (e.g., la signature dei metodi) e le operazioni sono più di alto livello.

1.4 Binary lifter

Un binary lifter è un programma che converte codice macchina in una rappresentazione di livello più alto, mantenendo lo stesso identico funzionamento originale. Lo scopo del lifting può essere sia quello di semplificare l'analisi del binario, sia di ricompilarlo per applicargli modifiche o utilizzarlo in ambienti diversi.

Alcuni utilizzi concreti sono:

- Binary instrumentation [11, 12]
  - Tracciare certi comportamenti del binario (e.g., numero di istruzioni eseguite, accessi alla memoria)
- Binary re-targeting [13]
  - Ricompilare il binario per un'architettura diversa da quella originale
- Software hardening [14, 15]
  - Aggiungere controlli al binario per evitare o mitigare certe vulnerabilità (e.g., verificare che gli accessi alla memoria non escano da confini prestabiliti)
- Sandboxing [16]
  - Limitare l'accesso al sistema sottostante (e.g., emulando le syscall)
- Obfuscation [17]
  - Modificare il binario in modo che sia difficile capirne il comportamento solo osservando il codice macchina

Esistono vari binary lifter con diverse caratteristiche. La maggior parte di questi hanno in comune il linguaggio target, LLVM IR [18], creato e usato da LLVM [19] come linguaggio intermedio. LLVM IR è stato creato appositamente per fare da tramite tra codice sorgente e codice macchina, cioè solitamente viene usato come linguaggio di destinazione per un qualche linguaggio di alto livello (e.g., C++, Rust) e poi viene compilato nel linguaggio macchina di destinazione finale. Nel caso dei lifter il verso è contrario: si parte dal linguaggio macchina e si "risale" (da cui "lifting") verso LLVM IR, che è considerato più di alto livello.

Esempi di lifter già esistenti, che generano tutti LLVM IR, sono:

- McSema [20]
  - Supporta binari per Linux (ELF) e Windows (PE) e la maggior parte delle istruzioni di x86, x86_64, AArch64, SPARC32, SPARC64.
  - Per il supporto a varie architetture, e quindi la loro rappresentazione semantica, utilizza remill [21] un'altra libreria mantenuta dallo stesso team
- revng [22]
  - Supporta solo binari ELF compilati per le seguenti architetture: i386, x86-64, MIPS, ARM, AArch64, s390x.
  - Utilizza la rappresentazione intermedia di QEMU [23] (TCG) per supportare le varie architetture e traduce quella in LLVM IR.
- llvm-mctoll [24]
  - Supporta solo ELF compilati per x86_64 e ARM32.
  - Non supporta istruzioni SIMD.

1.5 Differenza tra decompiler e lifter

Lo scopo di un decompiler è quello di generare, dato del codice compilato, del codice il più possibile simile a quello originale. In casi sufficientemente banali, il codice decompilato risulta quasi identico a quello originale. Tuttavia, per binari più complessi, scritti in linguaggi diversi o che usano tecniche particolari, è improbabile che il codice finale risulti funzionante e sempre corretto. Per generare codice così di alto livello, i decompilatori utilizzano varie euristiche e devono fare delle assunzioni che non sempre sono vere ma permettono di

generare codice più bello e leggibile nella maggior parte dei casi.

Lo scopo di un lifter è invece quello di generare codice compilabile che sia funzionalmente equivalente a quello originale, ma espresso in un linguaggio più di alto livello. Il focus è quindi sul funzionamento e non sulla forma. Il codice generato da un lifter è solitamente poco interpretabile da un essere umano e si presta più ad essere utilizzato per analisi automatiche.

In generale, possiamo individuare due metodi di analisi di un binario:

- Analisi statica: si comprende il comportamento del binario tramite lo studio del suo codice
  - Il codice decompilato può essere utilizzato per semplificare l'analisi manuale di un essere umano, in quanto solitamente più semplice e immediato da comprendere
  - Il codice liftato può essere utilizzato per certi tipi di analisi automatiche (e.g., capire con quali parametri viene chiamata una funzione, cercare accessi fuori dai limiti della memoria)
- Analisi dinamica: si comprende il comportamento del binario tramite la sua esecuzione
  - Il codice decompilato può essere utilizzato durante il debugging, mostrando da quale riga C è stata generata l'attuale istruzione assembly [25]
  - Il codice liftato può essere usato come sostituto del binario originale nel caso in cui non si possa eseguirlo direttamente (e.g., il binario era scritto per un'architettura diversa da quella per cui lo si vuole eseguire) oppure se ne voglia testare solo una parte (e.g., si vuole richiamare una funzione specifica con certi parametri)

Capitolo 2

Ghidra

In questo capitolo si presentano le principali caratteristiche di Ghidra e si descrive più nel dettaglio come faccia a supportare un numero così elevato di architetture.

2.1 Introduzione

Ghidra è un software open source per reverse engineering sviluppato dall'NSA, rilasciato in Marzo 2019. Lo scopo di Ghidra è assistere gli utenti nell'analisi di codice compilato, fornendo varie funzionalità, tra cui:

- Elenco delle funzioni all'interno del binario
- Recupero del control-flow graph delle funzioni
- Lista delle istruzioni assembly all'interno di ogni funzione
- Decompilazione di ogni funzione
- Possibilità di automatizzare analisi tramite estensioni e script

Supporta ufficialmente numerosi formati file, piattaforme e architetture e permette di aggiungerne di nuove attraverso la scrittura di estensioni.

Alcune architetture ufficialmente supportate degne di nota sono: `x86`, `x86_64`, `ARM`, `PowerPC`, `MIPS`, `AVR`, `Z80`, `Java` e `Dalvik` bytecode.

È in grado di riconoscere e caricare svariati formati file, tra cui: `PE`, `ELF`, `Mach-O`, `COFF`, `PDB`, `DWARF`, `APK`. Questo gli permette di gestire file eseguibili per tutti i principali sistemi operativi, come ad esempio: Windows, Linux,

macOS, Android

Ghidra presenta un'interfaccia grafica che permette di navigare le varie funzioni all'interno del binario e, per ognuna, mostrare il codice assembly e il codice decompilato.

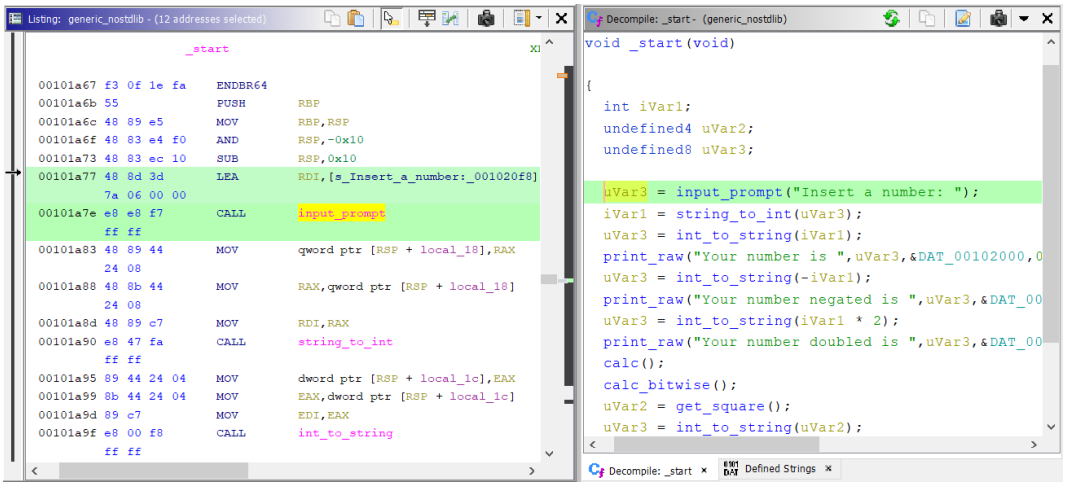

Figura 2.1: Pannelli per codice assembly e decompilato all'interno di Ghidra.

È possibile selezionare parte del codice assembly e vedere a quali parti del codice decompilato corrispondono e viceversa. Un'altra funzionalità molto importante è la possibilità di modificare e interagire con il codice decompilato. Per esempio permette di:

- Modificare il nome a variabili e funzioni
- Modificare il tipo delle variabili
- Modificare i parametri di una funzione
- Cambiare la funzione visualizzata tramite doppio click all'interno del codice
- Elencare tutti i punti in cui una funzione viene richiamata
- Elencare tutti i punti in cui una variabile viene utilizzata
- Definire nuovi tipi di dato (`struct`)

2.2 SLEIGH

SLEIGH [26] è un linguaggio creato appositamente per Ghidra, utilizzato per descrivere il set di istruzioni di un microprocessore generico. SLEIGH permette di descrivere, sia la conversione delle istruzioni da bit a testo leggibile da un umano (cioè da linguaggio macchina a assembly), sia il significato semantico delle istruzioni.

La parte più interessante ai fini di questo progetto è la descrizione semantica delle istruzioni, cioè la specifica di come vengano manipolati i dati e di quali effetti comporti la sua esecuzione all'interno del sistema. Per esempio, in `x86` l'addizione non si limita a fare la somma ma setta anche a `1` la `zero flag` nel caso in cui il risultato della somma sia `0`.

La semantica viene espressa tramite un altro linguaggio appositamente creato, il p-code [27]. SLEIGH permette di definire la conversione tra un'istruzione macchina e una lista di istruzioni p-code.

2.3 P-Code

Il p-code [27] è un linguaggio creato appositamente per supportare l'analisi all'interno di Ghidra. Contiene circa 70 istruzioni che assomigliano a quelle base che ci si aspetterebbe di trovare in un linguaggio assembly moderno.

Lo scopo del p-code è quello di riuscire a modellare qualsiasi set di istruzioni general purpose, in modo da poter convertire codice di vari processori in un'unica rappresentazione intermedia. In questo modo è sufficiente scrivere codice di analisi direttamente su un unico linguaggio, il p-code, invece che doverlo riscrivere per ogni architettura che si vuole supportare.

Uno degli utilizzi principali del p-code all'interno di Ghidra è durante la fase di decompilazione. In Ghidra, la decompilazione avviene esclusivamente tramite analisi del p-code, ignorando completamente il codice macchina dell'architettura originale. Questo permette di supportare la decompilazione di ogni architettura per la quale è stata definita la conversione delle istruzioni a p-code e rende Ghidra estremamente versatile.

2.3.1 Istruzioni p-code

- Data Moving

  - `COPY`: Copia un valore da una posizione a un'altra.
  - `LOAD`: Carica un valore dalla memoria.
  - `STORE`: Scrive un valore nella memoria.

- Arithmetic
  - `INT_ADD`: Somma tra due numeri interi.
  - `INT_SUB`: Sottrazione tra due numeri interi.
  - `INT_CARRY`: Overflow nella somma tra due numeri interi senza segno.
  - `INT_SCARRY`: Overflow nella somma tra due numeri interi con segno va.
  - `INT_SBORROW`: Overflow nella sottrazione tra due numeri interi con segna va.
  - `INT_2COMP`: Negazione di un numero intero tramite complemento a 2.
  - `INT_MULT`: Moltiplicazione tra due numeri interi.
  - `INT_DIV`: Divisione tra due numeri interi senza segno.
  - `INT_SDIV`: Divisione tra due numeri interi con segno.
  - `INT_REM`: Resto della divisione tra due numeri interi senza segno.
  - `INT_SREM`: Resto della divisione tra due numeri interi con segno.

- Logical
  - `INT_NEGATE`: Negazione dei bit di un numero intero.
  - `INT_XOR`: XOR bit a bit tra due numeri interi.
  - `INT_AND`: AND bit a bit tra due numeri interi.
  - `INT_OR`: OR bit a bit tra due numeri interi.
  - `INT_LEFT`: Shift a sinistra dei bit in un numero intero, riempiendo con 0.
  - `INT_RIGHT`: Shift a destra dei bit in un numero intero, riempiendo con 0.
  - `INT_SRIGHT`: Shift a destra dei bit in un numero intero, riempiendo con il bit più a sinistra (viene mantenuto il segno).
  - `POPCOUNT`: Conteggio dei bit settati a 1 in un numero intero.

- Integer Comparison

  - `INT_EQUAL`: Uguaglianza tra due numeri interi.
  - `INT_NOTEQUAL`: Non uguaglianza tra due numeri interi.
  - `INT_SLESS`: Se un numero intero con segno è minore di un altro.
  - `INT_SLESSEQUAL`: Se un numero intero con segno è minore o uguale a un altro.
  - `INT_LESS`: Se un numero intero senza segno è minore di un altro.
  - `INT_LESSEQUAL`: Se un numero intero senza segno è minore o uguale a un altro.

- Boolean
  - `BOOL_NEGATE`: Negazione di un numero booleano.
  - `BOOL_XOR`: XOR tra due numeri booleani.
  - `BOOL_AND`: AND tra due numeri booleani.
  - `BOOL_OR`: OR tra due nuemeri booleani.

- Floating Point
  - `FLOAT_ADD`: Somma tra due numeri floating-point (IEEE 754).
  - `FLOAT_SUB`: Sottrazione tra due numeri floating-point.
  - `FLOAT_MULT`: Moltiplicazione tra due numeri floating-point.
  - `FLOAT_DIV`: Divisione tra due numeri floating-point.
  - `FLOAT_NEG`: Negazione di un numero floating-point.
  - `FLOAT_ABS`: Valore assoluto di un numero floating-point.
  - `FLOAT_SQRT`: Radice quadrata di un numero floating-point.
  - `FLOAT_NAN`: Se un numero floating-point è NaN.

- Floating Point Compare
  - `FLOAT_EQUAL`: Se due numeri floating-point sono uguali.
  - `FLOAT_NOTEQUAL`: Se due numeri floating-point sono diversi.
  - `FLOAT_LESS`: Se un numero floating-point è minore di un altro.
  - `FLOAT_LESSEQUAL`: Se un numero floating-point è minore o uguale a un altro.

- Floating Point Conversion
  - `INT2FLOAT`: Cast da numero intero a floating-point.

  - `FLOAT2FLOAT`: Cast a floating-point di dimensione differente.
  - `TRUNC`: Cast da numero floating-point a intero.
  - `CEIL`: Arrotondamento per eccesso di un numero floating-point.
  - `FLOOR`: Arrotondamento per difetto di un numero floating-point.
  - `ROUND`: Arrotondamento di un numero floating-point.

- Branching
  - `BRANCH`: Salto incondizionato a una posizione costante.
  - `CBRANCH`: Salto a una posizione costante se la condizione è vera.
  - `BRANCHIND`: Salto incondizionato a una posizione calcolata durante l'esecuzione.
  - `CALL`: Chiamata di una funzione costante.
  - `CALLIND`: Chiamata di una funzione calcolata durante l'esecuzione.
  - `RETURN`: Ritorno dalla funzione corrente a quella chiamante.

- Extension/Truncation
  - `INT_ZEXT`: Aumento della dimensione di un numero intero, estesa con `0`.
  - `INT_SEXT`: Aumento della dimensione di un numero intero, estesa con il bit del segno.
  - `PIECE`: Unione di due blocchi di bit in un numero unico.
  - `SUBPIECE`: Estrazione di una porzione di un blocco di bit.

2.3.2 Conversione da assembly a p-code

Dato che le operazioni p-code sono tutte relativamente semplici, solitamente un'istruzione assembly viene convertita in svariate istruzioni p-code, soprattutto in architetture CISC come `x86_64`.

Per esempio, una banale istruzione `x86_64` di `ADD` tra un registro e una costante:

```
ADD RSP ,0x8
```

Viene convertita in 9 istruzioni p-code:

```
CF = INT_CARRY RSP , 8:8
OF = INT_SCARRY RSP , 8:8
RSP = INT_ADD RSP , 8:8
SF = INT_SLESS RSP , 0:8
ZF = INT_EQUAL RSP , 0:8
$U12e80 :8 = INT_AND RSP , 0xff :8
$U12f00 :1 = POPCOUNT $U12e80 :8
$U12f80 :1 = INT_AND $U12f00 :1, 1:1
PF = INT_EQUAL $U12f80 :1, 0:1
```

In questo esempio, la `ADD` vera e propria avviene nella terza riga (`INT_ADD`), tuttavia in `x86` l'addizione modifica non solo il registro di destinazione ma anche vari registri di flag (carry, overflow, sign, zero, parity). È quindi necessario che le istruzioni p-code modellino tutti i side effects dell'architettura originale, in quanto le istruzioni successive potrebbero utilizzare queste flag.

Capitolo 3

Dragonlifter

In questo capitolo si descrive nel dettaglio il funzionamento interno di Dragonlifter e la struttura del codice generato.

3.1 Introduzione

Dragonlifter è un lifter che converte istruzioni p-code in codice C, basandosi sull'analisi condotta da Ghidra.

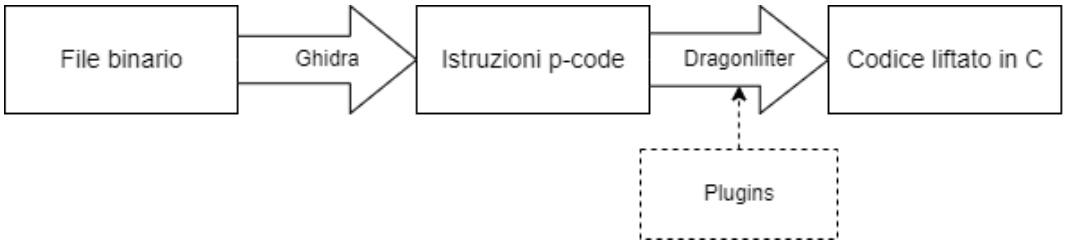

Figura 3.1: Binary lifting tramite Dragonlifter.

Utilizzare Ghidra permette di supportare sin da subito un elevato numero di architetture, formati file e semplifica l'implementazione del lifter.

A differenza di altri lifter che convertono a LLVM IR, Dragonlifter converte in codice C. Questo comporta svariati benefici, tra cui:

- Non è necessario avere una dipendenza a LLVM, è sufficiente generare stringhe
- C è immediatamente leggibile e interpretabile da chiunque
- C è di livello più elevato e permette di generare codice più conciso

Per rendere l'implementazione il più semplice ed estendibile possibile, Dragonlifter è stato scritto in Python.

Dato che è impossibile modellare tutte le caratteristiche di un'architettura tramite p-code (e.g., syscall, MMIO), una funzionalità a cui è stata data molta importanza all'interno di questo progetto è la possibilità di modificare ed estendere il codice generato. Dragonlifter include infatti un sistema di plugin molto potente che permette, in maniera molto semplice, di modificare tutte le fasi di lifting e quindi il codice generato.

3.2 Casi d'uso di Dragonlifter

Lo scopo finale di Dragonlifter è quello di generare codice C compilabile che abbia lo stesso comportamento del binario originale. Dato che lo scopo principale del p-code è quello di supportare l'analisi statica del codice, Dragonlifter si concentra solo sull'analisi dinamica.

Gli utilizzi principali di Dragonlifter sono:

- Esecuzione di funzioni o interi programmi compilati per altre architetture e sistemi operativi
  - Per esempio, si vuole eseguire nel proprio computer x86_64 una funzione presente in un'app Android/iOS compilata solo per ARM
- Patching senza accesso al codice sorgente
  - Aggiungere funzionalità a un binario già esistente
  - Fixare un bug
  - Modificare totalmente l'implementazione di una funzione
- Binary instrumentation, cioè aggiunta di codice al fine di tracciare certi comportamenti
  - Instruction counting
  - Tracing di tutte le istruzioni eseguite
  - Tracing degli accessi alla memoria
  - Tracing delle syscall eseguite

3.3 Estrazione dati da Ghidra

3.3.1 Ghidra extractor

Per poter liftare il programma è quindi necessario ottenere delle informazioni da Ghidra. Ghidra è scritto in Java e fornisce un'API [28] molto dettagliata per estenderlo e per accedere a tutti i dati disponibili tramite script.

Per creare gli script è tecnicamente possibile usare qualsiasi linguaggio che compili per la JVM, tuttavia quelli supportati ufficialmente sono Java e Jython [29]. Jython è un'implementazione di Python per la JVM che compila sorgenti Python in bytecode Java. Essendo il resto del progetto implementato in Python, è stata naturale la scelta di Jython per lo script.

Un problema di Jython è che supporta solo Python 2, il quale è oltretutto ufficialmente abbandonato dall'inizio del 2020. Dragonlifter è scritto in Python 3 e non è quindi possibile eseguirlo direttamente su Ghidra tramite Jython. La soluzione è stata quella di dividere il lifting in due passaggi separati:

- l'estrazione delle informazioni da Ghidra e
- l'utilizzo di queste informazioni per fare lifting.

È stato quindi creato uno script Jython per estrarre le informazioni necessarie da Ghidra. Queste informazioni vengono scritte in un file JSON che poi verrà importato da Dragonlifter.

3.3.2 Dati estratti

Ad alto livello, le informazioni necessarie sono:

- Elenco di funzioni all'interno del binario
  - Per ogni funzione un elenco di istruzioni
  - Per ogni istruzione un elenco di p-code
  - Per ogni p-code i varnode di input e quello di output
- Elenco di blocchi di registri (e.g., il blocco in `0` è grande `8` byte e contiene `RAX`, `EAX`, `AX`, `AL`)
  - Per ogni blocco, l'elenco di registri all suo interno
- Elenco di blocchi di memoria e a quale address space corrispondono

- Elenco di nomi di operazioni non direttamente gestite da Ghidra (e.g., syscall)

Questi dati vengono importati dal file JSON all'interno di Dragonlifter. In modo particolare, la classe principale che mantiene tutte le informazioni è `Program`.

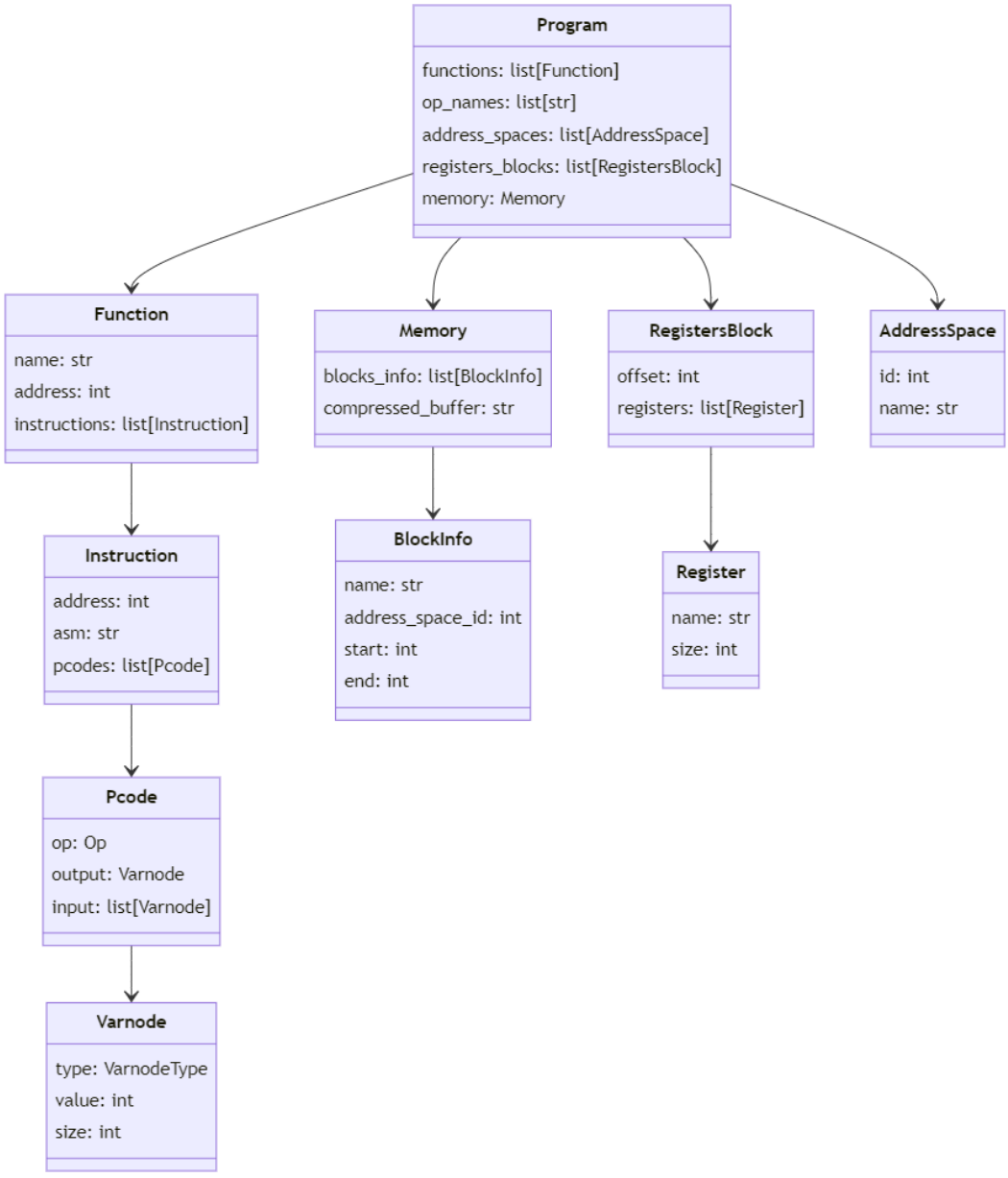

Figura 3.2: Struttura dei dati estratti da Ghidra.

3.3.3 Varnode

Per modellare il fatto che le istruzioni p-code prendono degli input e hanno un output, vengono usati degli oggetti chiamati varnode. Un varnode può essere visto come una variabile simbolica con informazioni su dove i dati vengono memorizzati, la loro dimensione e in certi casi il valore vero e proprio.

Importante tenere a mente il fatto che un varnode non descrive il tipo di dato della variabile (e.g., int, unsigned, float), ma rappresenta solo un insieme di byte. Come quei byte vengano poi interpretati dipende solamente dall'istruzione p-code che viene eseguita.

Per i nostri scopi, possiamo individuare le seguenti tipologie di varnode:

- Temp: creato sinteticamente da Ghidra per modellare meglio certe operazioni, non ha corrispondenza nel programma originale
- Constant: contiene un valore costante, memorizzato direttamente dentro al varnode
- Register: il valore è memorizzato in uno specifico registro
- RAM: il valore è memorizzato a uno specifico indirizzo in memoria

Salvo casi speciali (e.g., syscall), le operazioni p-code effettuano input/output interamente tramite i varnode definiti e non hanno comportamenti nascosti. Per esempio, `INT_ADD` prende in input due varnode, accede al loro valore come se fosse un numero intero, li somma e scrive il risultato all'interno del varnode di output. Tutti questi varnode devono avere la stessa dimensione (e.g., 8 byte per somme tra `long long` in C).

3.4 Lifters

La classe `Dragonlifter` è l'entry-point dell'intero programma. Permette di specificare il lifter utilizzato per ogni passo e quindi di modificare l'intera procedura di lifting. Questa possibilità è la chiave per implementare il sistema di plugin.

- `Dragonlifter` istanzia un `ProgramLifter` al quale passa il `Program` da liftare e all'interno del quale avviene il lifting vero e proprio.
- Il `ProgramLifter` istanzia, per ogni funzione da esportare, un `FunctionLifter`.

- Il `FunctionLifter` istanzia, per ogni istruzione all'interno della funzione, un `InstructionLifter`.

- L'`InstructionLifter` istanzia, per ogni p-code generato dall'istruzione, un `PcodeLifter`.

- Il `ProgramLifter` istanzia anche un CoreLifter per generare tutto il codice di infrastruttura necessario affinché il lifting funzioni (e.g., emulazione della memoria e dei registri).

Nel costruttore di ognuno di questi lifter, oltre ai dati necessari (e.g., nel caso del `FunctionLifter` la funzione da liftare), viene passata anche l'istanza di Dragonlifter creata inizialmente. Questo è necessario perché al suo interno è presente, per ogni lifter, la classe corretta da istanziare (necessario per supportare i plugin) e l'istanza di `CoreLifter` associata al `Program` da liftare.

`CoreContext` contiene tutte le informazioni trasversali ai vari lifter, per esempio quali registri sono stati effettivamente usati e quali sono gli indirizzi delle variabili temporanee.

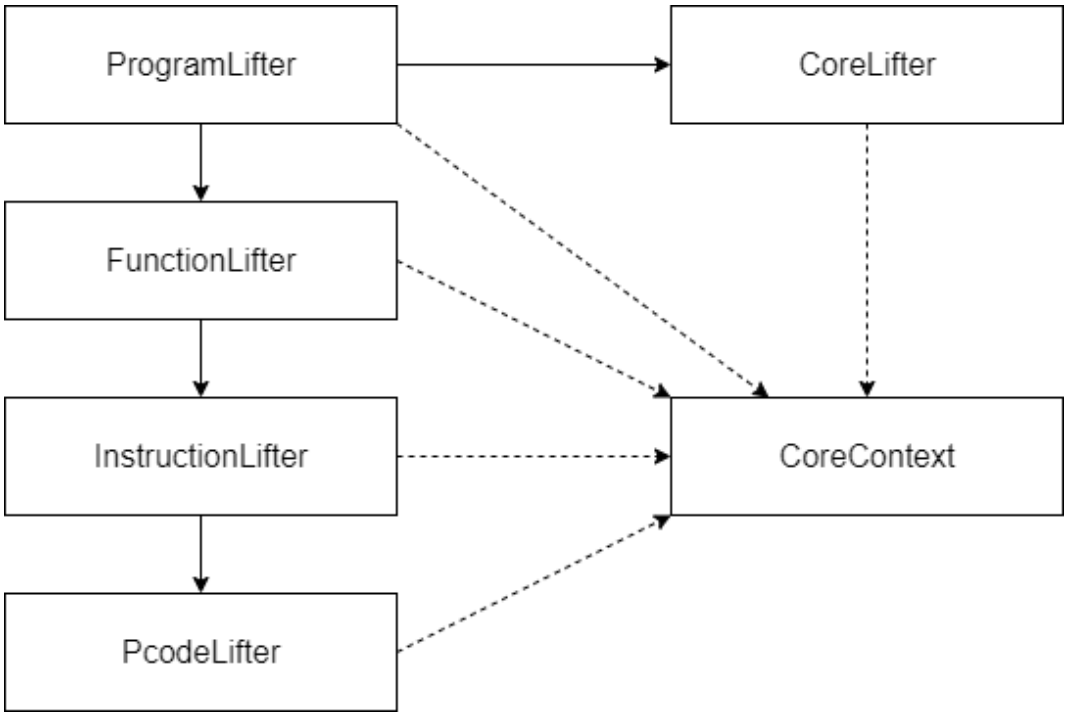

Figura 3.3: Flusso di esecuzione dei lifter.

3.4.1 ProgramLifter

`ProgramLifter` è la classe che si occupa di liftare l'intero programma. Il costruttore prende come parametro il `Program` da liftare, istanzia gli altri lifter e genera tutti i file necessari.

La funzione principale `emit`, prende come argomento il percorso di una cartella e lifta l'intero programma al suo interno, creando i necessari file. Nello specifico, genera il contenuto di `dragonlifter.c` e `dragonlifter.h` tramite `CoreLifter` e il contenuto di ogni funzione tramite `FunctionLifter`.

3.4.2 CoreLifter

`CoreLifter` è la classe che si occupa di generare l'infrastruttura utilizzata da tutto il codice liftato. Genera la memoria, i registri, i tipi di dato, funzioni, define.

Dovendo generare varie sezioni di codice, spesso dipendenti l'una dall'altra, il suo codice ha una struttura piuttosto diversa rispetto a quella degli altri lifter.

Contiene tre funzioni principali:

- `setup`, da chiamare per prima, popola le strutture utilizzate poi dalle altre funzioni
- `lift_header`, genera il codice di infrastruttura che deve essere incluso da ogni file (e.g., include, define, typedef, extern) e che verrà poi esportato in `dragonlifter.h`
- `lift_c`, genera il codice di infrastruttura che deve essere compilato una sola volta (e.g., l'array di byte della memoria e dei registri) e che verrà poi esportato in `dragonlifter.c`

All'interno di questa classe vengono definiti vari campi, che conterranno le righe di codice da inserire nel file `dragonlifter.c` e nel file `dragonlifter.h`.

La funzione `setup` si occupa di popolare questi campi, mentre le funzioni `lift_*` si limitano a concatenare le stringhe all'interno dei campi, tramite il carattere di "a capo". Per esempio `lift_c` è semplicemente un join tra `c_includes`, `c_fields` e `c_body`; che contengono rispettivamente le righe di `include`, i campi da creare (e.g., memory, registers) e del codice più generico

inserito alla fine del file.

Internamente, la funzione `setup` si limita a richiamare altre funzioni di `setup` più specifiche, per esempio `setup_math` e `setup_temp_variables`.

```
def setup_math(self):
    self.h_includes.append("#include <math.h>")
    self.defines.append(self.generate_math_defines())
```

In questo caso, per poter supportare certe operazioni matematiche, è richiesto importare `math.h`, quindi la aggiungiamo alla lista di `includes`. Inoltre generiamo anche il codice necessario per gestire effettivamente le chiamate alle funzioni matematiche, che in questo caso viene implementato tramite delle define simili a:

```
#define SIGN(sz, v) ((v) >= 0)
#define SQRT(sz, v) sqrt(v)
```

La definizione di nuove variabili globali può essere esemplificata da questa parte di codice di `setup_temp_variables`:

```
def setup_temp_variables(self):
    self.h_fields.append(f"extern varnode_t {temp_variables};")
    self.c_fields.append(f"varnode_t {temp_variables};")
```

Dove `temp_variables` contiene una lista di nomi di variabili temporanee (e.g., `temp_12345`), già separate da virgola. È necessario creare sia la dichiarazione `extern` all'interno del file header (per far sapere al codice liftato che quelle variabili esistono) sia le variabili vere e proprie all'interno del file C.

Questa separazione in vari metodi di "setup" rende molto facile aggiungere nuove sezioni di codice indipendenti dalle precedenti e rende più semplice estendere `CoreLifter`.

3.4.3 FunctionLifter

`FunctionLifter` è la classe che si occupa di liftare le funzioni presenti nel binario originale. Il costruttore prende come parametro una `Function`:

```
class Function:
    name: str
    address: int
    instructions: list[Instruction]
```

Viene istanziato un `FunctionLifter` per ogni funzione da liftare.

Il metodo principale `lift`, ritorna una stringa contenente la conversione in C della funzione liftata. Internamente `lift` è semplicemente l'unione delle righe ritornate da altri metodi più specifici, e.g., `generate_header`, `generate_body` e `generate_footer`.

Il codice C generato si può dividere concettualmente nelle seguenti parti:

- Header
  - Signature della funzione: tipo di ritorno (sempre `void`), nome, parametri (sempre vuoti)
  - Parentesi graffa aperta
- Body
  - Potenziale codice di inizializzazione
  - Dichiarazione variabili e costanti
  - Istruzioni liftate
  - Potenziale codice di clean up
- Footer
  - Parentesi graffa chiusa

Un esempio di codice generato è:

```
void fun() {
    init();
    // variables/constants
    [...]
    //instructions
    [...]
    cleanup();
}
```

3.4.4 InstructionLifter

`InstructionLifter` è la classe che si occupa di liftare le istruzioni assembly presenti nel binario originale. Il costruttore prende come parametro una `Instruction`:

```
class Instruction:
    address: int
    asm: str
    pcodes: list[Pcode]
```

Viene istanziato un `InstructionLifter` per ogni istruzione da liftare.

Il metodo principale `lift`, ritorna una stringa contenente la conversione in C dell'istruzione liftata. Internamente `lift` è semplicemente l'unione delle righe ritornate da altri metodi più specifici, e.g., `generate_instruction_label` e `generate_body`.

Il codice generato si può dividere concettualmente in:

- Label
  - Contrassegna l'inizio di una nuova istruzione
  - Necessario per istruzioni di branching
  - Utile durante il debugging e per rendere più chiaro il codice generato
- Body
  - Istruzioni p-code liftate

Un esempio di codice generato è:

```
ADDR_123456:; // MOV RBP,RSP
[...]
```

3.4.5 PcodeLifter

`PcodeLifter` è la classe che si occupa di liftare le istruzioni p-code generate dalle istruzioni macchina presenti nel binario originale. Il costruttore prende come parametro un `Pcode`:

```
class Pcode:
    op: Op
    output: Varnode
    input: list[Varnode]
```

Dato che la maggior parte delle operazioni p-code sono elementari, il loro lifting è solitamente banale. Nella maggior parte dei casi infatti, a un'istruzione

p-code corrisponde una sola riga di `C`. Per esempio, il p-code `INT_ADD` viene convertito in codice `C` simile a:

```
varnode_out = varnode_in0 + varnode_in1
```

Il corretto lifting delle istruzioni p-code è cruciale ai fini del funzionamento di Dragonlifter. Nonostante le istruzioni p-code siano concettualmente semplici, il loro numero è elevato e hanno associate delle informazioni e vincoli di cui è necessario tenere conto durante il lifting:

- Nome dell'istruzione p-code
- Numero di varnode in input
- Presenza o meno del varnode di output
- Per ogni varnode, tramite quale tipo interpretare i byte al suo interno (intero senza segno, intero con segno, floating-point, bool)
- Vincoli sulla dimensione dei varnode
- Vincoli sulla tipologia dei varnode

Il design di PcodeLifter è stato quindi oggetto di particolari attenzioni per far sì che fosse possibile esprimere tutte queste informazioni in maniera concisa ed efficace.

Per aggiungere il supporto al lifting di una nuova istruzione p-code è sufficiente creare un metodo con una determinata struttura. Infatti tutte le informazioni e vincoli relativi all'istruzione verranno espressi esclusivamente all'interno di questo metodo.

- Il nome del metodo indica a quale p-code si fa riferimento
- I parametri del metodo indicano i varnode (sia di input che di output) e il loro tipo (con/senza segno, float)
- Il valore di ritorno è il codice `C` equivalente all'istruzione
- Altri vincoli (e.g., quelli sulla dimensione) vengono verificati tramite assert

Per esempio, l'implementazione di `INT_ADD` è:

```
def _int_add(self, out: Output, in0: Input, in1: Input):
    assert out.size == in0.size == in1.size
    return f"{self.var(out)} = {self.var(in0)} + {self.var(in1)};"
```

Da questo metodo possiamo ottenere tutte le seguenti informazioni:

- Il nome del p-code implementato è `INT_ADD`
- L'istruzione ha un varnode di output e due di input
- Tutti i varnode sono trattati come interi senza segno
- La dimensione di tutti i varnode deve essere uguale
- L'implementazione in C è una somma tra i due varnode

In generale, l'implementazione di tutte le altre istruzioni p-code segue una struttura molto simile a questa.

Il nome del metodo deve essere quello del p-code, ma in minuscolo e preceduto da un underscore. L'underscore è necessario in quanto ci sono alcune istruzioni p-code che produrrebbero nomi di metodi non validi in Python (e.g., `RETURN`). Per ottenere il nome del metodo che gestisce un determinato p-code è possibile usare il seguente codice `f"_{pcode.name.lower()}"`.

I varnode non hanno un tipo di dato, tuttavia il p-code sì. Ogni istruzione lavora con un tipo specifico di dato, cioè interpreta i byte del varnode come se fossero di un determinato tipo. Per esempio, `INT_ADD` interpreta tutti i suoi varnode come se fossero degli interi senza segno, mentre `INT_SDIV` (signed integer division) li interpreta come se fossero con il segno. In entrambi i casi la dimensione dei varnode non è prefissata ma è necessario che, all'interno di un'istruzione, sia sempre la stessa. Nel caso in cui si volesse sommare un intero a 4 byte con uno a 8 byte, sarebbe prima necessario estendere quello a 4 in uno a 8 (tramite `INT_ZEXT` se interpretato senza segno o tramite `INT_SEXT` con il segno).

Per esprimere la tipologia di dato (chiamata `kind` all'interno del codice) con cui viene interpretato un varnode all'interno di un'istruzione p-code, sono state introdotte delle apposite sottoclassi di `Varnode`.

```
class VarKind(Enum):
    UNSIGNED = auto()
    SIGNED = auto()
    FLOATING = auto()
class Var(Varnode):
    kind: VarKind
    constraint_size: Optional[int] = None

class Output(Var): kind = VarKind.UNSIGNED
```

```
class Input(Var): kind = VarKind.UNSIGNED
class OutputSigned(Output): kind = VarKind.SIGNED
class InputSigned(Input): kind = VarKind.SIGNED
class OutputFloating(Output): kind = VarKind.FLOATING
class InputFloating(Input): kind = VarKind.FLOATING
class OutputBool(Output): constraint_size = 1
class InputBool(Input): constraint_size = 1
```

Queste classi sono utilizzate solamente all'interno del PcodeLifter e appaiono come annotazioni per i tipi dei parametri.

La classe `Var` è quella base e permette di esprimere sia il `kind` del varnode sia eventuali vincoli di dimensione (e.g., un bool deve essere grande 1 byte). La classe `Output` rappresenta un varnode di output e `Input` rappresenta un varnode di input, entrambi interi senza segno. Ci sono delle varianti di `Input` e `Output` per interi con segno, floating-point e booleani.

Per esempio, `INT_SLESS` rappresenta l'operazione booleana di minore tra due numeri interi con segno e viene codificata così:

```
def _int_sless(self, out: OutputBool, in0: InputSigned, in1:
    InputSigned):
    assert in0.size == in1.size
    return f"{self.var(out)} = {self.var(in0)} < {self.var(in1)};"
```

Sappiamo quindi che gli input devono essere interpretati come interi con segno mentre invece l'output come un bool. I vincoli sono che l'output deve avere dimensione 1 (perché è un bool) e che la dimensione dei due input deve essere la stessa.

Vediamo ora l'implementazione di `INT_LESS`, cioè l'operazione booleana di minore tra due numeri interi senza segno:

```
def _int_less(self, out: OutputBool, in0: Input, in1: Input):
    assert in0.size == in1.size
    return f"{self.var(out)} = {self.var(in0)} < {self.var(in1)};"
```

Possiamo notare come il corpo delle due funzioni, `INT_SLESS` e `INT_LESS`, sia identico e che l'unica differenza sta nel tipo dei parametri del metodo (`InputSigned` invece di `Input`). Questo accade perché, il metodo `self.var`, usato per convertire il varnode in codice `C`, ritorna una stringa diversa in funzione di quale `Var` gli viene passato. Nello specifico, esegue in automatico il cast al tipo giusto in funzione del `kind`.

Viene istanziato un `PcodeLifter` per ogni istruzione p-code da liftare. Il metodo principale `lift`, ritorna una stringa contenente la conversione in `C` dell'istruzione p-code.

Internamente, il metodo `lift` deve chiamare il metodo giusto e passargli i parametri corretti. Più nello specifico:

- Trova il metodo corrispondente p-code da liftare, cioè quello con il nome `f"_{pcode.name.lower()}"`
- Ottiene il tipo (annotazione) dei parametri della funzione
  - In questo modo conosce quali e quanti varnode vengono richiesti
- Itera tutti i `Var` richiesti e cerca di costruirli usando i varnode del p-code che sta considerando
  - Se è `Output` o una sua qualche sottoclasse (e.g., `OutputSigned`), allora prende il varnode di output del p-code
  - Se è `Input` o una sua qualche sottoclasse (e.g., `InputSigned`), allora prende il prossimo varnode di input del p-code
  - Se è `Optional[Output]`, allora prende il varnode di output del p-code solo se non è `None`
  - Se è `list[Input]`, allora prende tutti i varnode di input rimanenti nel p-code
- Controlla che il numero di input e output richiesti siano uguali a quelli processati
- Chiama il metodo passandogli come argomenti gli input e output costruiti

3.5 Plugins

Lo scopo dei plugin è quello di modificare o estendere ogni fase del processo di lifting. Durante la fase di progettazione, è stata data molta importanza nel rendere il processo di creazione di plugin il più semplice possibile. Questo ha inciso fortemente sull'architettura finale del progetto.

Il meccanismo di creazione di un plugin risulta quindi molto semplice in superficie, a discapito di maggiore complessità interna e dell'utilizzo di delle

funzionalità un po' inusuali di Python.

Per creare un plugin è infatti sufficiente creare una sottoclasse del lifter che si vuole modificare e sovrascrivere le funzioni adatte. Basta poi passare questa classe (o un intero modulo contenete più classi) durante la costruzione di Dragonlifter per far sì che venga utilizzata durante il lifting.

Ipotizziamo che la classe `FunctionLifter` contenga un metodo `function_name` usato per generare il nome della funzione liftata.

```
class FunctionLifter:
    def function_name(self):
        return self.function.name
```

Supponiamo ora di voler creare un plugin che modificare il nome di tutte le funzioni generate, aggiungendogli un suffisso.

```
class FooFunctionLifter(FunctionLifter):
    def function_name(self):
        return super().function_name() + "_foo"
```

Nonostante il plugin si presenti come una normale sottoclasse, in realtà internamente viene gestito in maniera diversa. Questo permette di usare contemporaneamente più plugin che modificano lo stesso lifter.

```
class BarFunctionLifter(FunctionLifter):
    def function_name(self):
        return super().function_name() + "_bar"
```

Applicare `FooFunctionLifter` e `BarFunctionLifter` (in questo ordine) porta a generare funzioni con il nome che finisce con `_foo_bar`.

Per supportare più plugin per la stessa classe e mantenere la loro dichiarazione così semplice, internamente i plugin vengono uniti in una classe unica. Per farlo, Dragonlifter sfrutta l'ereditarietà multipla di Python e la possibilità di definire nuovi tipi a runtime. Più nel dettaglio, il modo in cui vengono uniti i vari plugin è analogo al creare una classe che eredita da tutti i plugin:

```
class FunctionLifterPlugins(FooFunctionLifter, BarFunctionLifter):
    pass
```

In Python, le classi possono ereditare da più tipi in contemporanea. La risoluzione dei metodi viene eseguita da sinistra verso destra.

È quindi sufficiente utilizzare un'istanza di `FunctionLifterPlugins` invece che quella di `FunctionLifter` per applicare entrambi i plugin creati. Degno di nota è il fatto che, quando si utilizza `FunctionLifterPlugins`, la chiamata a `super().function_name()` dentro a `BarFunctionLifter` ritorna l'istanza di `FooFunctionLifter` e non quella di `FunctionLifter` da cui ufficialmente eredita. Per quello che ci interessa, è come se ottenessimo una copia di `BarFunctionLifter` che però eredita da `FooFunctionLifter` e non da `FunctionLifter`.

In Dragonlifter tuttavia, i plugin vengono dichiarati a runtime, quindi non è possibile usare la sintassi mostrata sopra per creare la `class` che unisce tutti i plugin.Per fortuna, Python mette a disposizione una funzione `type` che permette di creare un nuovo tipo a runtime. Questa funzione prende 3 parametri:

- Il nome del tipo
- La tupla di classi da cui eredita
- Un dizionario di variabili globali (che per i nostri fini non è interessante e sarà sempre vuoto)

È possibile quindi riscrivere l'equivalente di quello sopra utilizzando `type`:

```
FunctionLifterPlugins = type(
    "FunctionLifterPlugins",
    (FooFunctionLifter, BarFunctionLifter),
    {})
```

Dragonlifter permette di importare i plugin anche specificando un intero modulo e non solo una classe. Questo permette di importare interi file di Python come plugin ed è particolarmente comodo per usarli dalla riga di comando. Infatti la CLI permette di specificare quali plugin applicare semplicemente specificando la path dei file. Per fare ciò viene utilizzata un funzionalità di Python che permette di caricare un file `.py` a runtime e di ottenerlo come modulo.

Una volta ottenuto un modulo, vengono ispezionati tutti i membri del modulo e si cercano le classi che ereditano da un lifter. Alla fine di questo procedimento, si ottiene un dizionario dove a ogni lifter è associata una lista di plugin, inseriti nello stesso ordine con cui sono stati ricevuti.

3.6 Struttura del codice generato

Il lifter genera un file `.c` per ogni funzione da liftare all'interno del binario originale e altri due file speciali: `dragonlifter.h` e `dragonlifter.c`.

Supponiamo esistano due funzioni all'interno del nostro binario: `_start` e `_exit`. Il lifting produrrà 4 file: `_start.c`, `_exit.c`, `dragonlifter.h` e `dragonlifter.c`.

In generale, per compilare il programma liftato, è sufficiente includere tutti i file `.c`:

```
gcc *.c
```

I file `.c` corrispondenti a una funzione iniziano tutti importando `dragonlifter.h`. Il resto del file contiene l'implementazione della funzione liftata.

I file `dragonlifter.c/h` si occupano di emulare l'ambiente previsto dal binario originale (e.g., registri) e di fornire delle funzioni usate dalle istruzioni liftate. Più nello specifico, `dragonlifter.h` contiene:

- Import necessari al funzionamento del programma (e.g., `stddef.h`, `stdint.h`, `math.h`)
- Typedef di alias e struct (e.g., `u8`, `varnode_t`)
- Define di macro (e.g., `CEIL`, `RAM_ADDR`, `CALL_FUNCTION_AT`) e registri
- Funzioni usate internamente (e.g., `popcount`, `find_memory_pos`)
- Elenco di tutte le funzioni liftate

`dragonlifter.c` contiene i valori concreti iniziali della memoria, dei registri, puntatori alle funzioni.

3.6.1 `varnode_t`

Dato che un varnode può avere dimensioni diverse e interpretare i dati come tipi diversi, nel codice liftato viene rappresentato come una `union` contenente tutte le combinazioni di tipi e dimensioni che verranno richieste.

```
typedef union {
    i8 _1s;
    i16 _2s;
    i32 _4s;
```

```
    i64 _8s;
    u8 _1;
    u16 _2;
    u32 _4;
    u64 _8;
    f32 _4f;
    f64 _8f;
    f80 _10f;
} varnode_t;
```

Il nome di ogni campo all'interno della `union` è composto da:

- Underscore (_)
- Numero di byte contenuti
- Eventuale suffisso per specificare il tipo
  - Vuoto nel caso di interi senza segno
  - '`s`' nel caso di interi con segno
  - '`f`' nel caso di numeri floating-point

Per esempio il campo `_2s` vuole dire "intero con segno grande 2 byte", cioè quello che in C solitamente corrisponde a uno `short`.

Nello stesso indirizzo di memoria può essere richiesto accedere con tipo e dimensione diversa in funzione dell'istruzione p-code. Grazie a questa `union`, è sufficiente fare un cast a un puntatore di `varnode_t` e poi accedere al campo giusto. Per esempio, per accedere a valore puntato da `address` come se fosse un intero di 2 byte con segno si può scrivere:

```
((varnode_t *)address)->_2s
```

3.6.2 Emulazione della memoria

Il contenuto di tutti i blocchi di memoria del programma viene codificato in un unico array di byte chiamato `memory`, il quale agirà come una sorta di memoria virtuale che emula quella del binario originale. I blocchi vengono tutti memorizzati contigui ma è necessario sapere, dato un indirizzo del binario originale, a quale blocco di memoria corrisponde e, di conseguenza, da quale posizione dell'array parte. Per fare questo si utilizza un array di blocchi che mette in corrispondenza un indirizzo emulato (cioè quello utilizzato dal codice originale) a un indirizzo fisico:

```
memory_block_t memory_blocks[]
```

Dove la definizione di `memory_block_t` è:

```
typedef struct {
  address_t addr;
  byte * ptr;
} memory_block_t;
```

Il quale memorizza, per ogni blocco di memoria estratto da Ghidra, in `addr` l'indirizzo nel quale iniziava il blocco nel binario originale e in `ptr` il puntatore a dove si trova effettivamente.

Ogni accesso alla memoria all'interno del codice liftato deve quindi passare tramite questa indirezione e convertire l'indirizzo virtuale in un indirizzo fisico. Difatti, per semplificare la modifica di questo comportamento e per rendere il codice generato più leggibile, vengono definite queste due macro:

```
#define RAM_ADDR(addr) ((varnode_t*)find_memory_pos(addr))
#define RAM(addr) (*RAM_ADDR(addr))
```

`RAM_ADDR` effettua la conversione da indirizzo virtuale a indirizzo fisico mentre `RAM` semplicemente dereferenzia quell'indirizzo. Dato che l'accesso ai valori avviene soltanto attraverso varnode, il puntatore viene castato a un `varnode_t`. `find_memory_pos` è una funzione che scorre `memory_blocks` per trovare il blocco all'interno del quale rientra l'indirizzo virtuale richiesto.

Questi define permettono di esprimere un accesso alla memoria nel codice liftato tramite la macro RAM. Per esempio per accedere al primo valore sullo stack è sufficiente scrivere `RAM(RSP)`.

3.6.3 Emulazione dei registri

La rappresentazione dei registri assomiglia molto a quella usata per la memoria. Infatti i registri vengono semplicemente definiti come un array di byte:

```
byte registers[8810];
```

Tuttavia, a differenza della memoria, quando accediamo a un registro sappiamo sempre esattamente a quale indirizzo stiamo facendo l'accesso, quindi

non è necessario cercare dinamicamente la posizione giusta.

Per accedere al registro `RAX` per esempio, possiamo scrivere:

```
((varnode_t*)&registers[0])->_8
```

Che significa: prendi il registro a indice 0 e interpretalo come se fosse un intero senza segno grande 8 byte.

Questo ragionamento si può applicare a tutti i registri dell'architettura, per ogni registro utilizzato dal programma liftato viene quindi creato un apposito define per mappare da nome del registro a accesso alla memoria:

```
#define RAX ((varnode_t*)&registers[0])->_8
```

In questo modo il codice liftato risulta molto più leggibile e semplice da debuggare.

3.6.4 Esempio di codice liftato

Creiamo appositamente una funzione molto semplice di esempio, per verificare come venga convertita dal nostro lifter. La funzione `add42` prende come parametro un numero intero e gli somma `42`:

```
int add42(int n) {
    return n + 42;
}
```

Dopo aver compilata per `x86_64`, l'assembly ottenuto da Ghidra corrispondente alla funzione `add42` è:

```
ENDBR64
PUSH RBP
MOV RBP, RSP
MOV dword ptr [RBP + local_c], EDI
MOV EAX, dword ptr [RBP + local_c]
ADD EAX, 0x2a
POP RBP
RET
```

Eseguiamo infine Dragonlifter. Quello seguente è il codice generato dal nostro lifter per la funzione `add42`:

```
ADDR_101577:; // ENDBR64
```

```
ADDR_10157B:; // PUSH RBP
temp_59904._8 = RBP;
RSP = RSP - ((u64)8);
RAM(RSP)._8 = temp_59904._8;
ADDR_10157C:; // MOV RBP,RSP
RBP = RSP;
ADDR_10157F:; // MOV dword ptr [RBP + -0x4],EDI
temp_12544._8 = RBP + ((u64)-4);
temp_48896._4 = EDI;
RAM(temp_12544._8)._4 = temp_48896._4;
ADDR_101582:; // MOV EAX,dword ptr [RBP + -0x4]
temp_12544._8 = RBP + ((u64)-4);
temp_48896._4 = RAM(temp_12544._8)._4;
EAX = temp_48896._4;
RAX = EAX;
ADDR_101585:; // ADD EAX,0x2a
CF = (u32)(EAX + ((u32)42)) < EAX;
OF = SIGN(4, EAXs) == SIGN(4, ((i32)42)) && SIGN(4, (i32)(EAXs +
    ((i32)42))) != SIGN(4, ((i32)42));
EAX = EAX + ((u32)42);
RAX = EAX;
SF = EAXs < ((i32)0);
ZF = EAX == ((u32)0);
temp_77440._4 = EAX & ((u32)255);
temp_77568._1 = POPCOUNT(4, temp_77440._4);
temp_77696._1 = temp_77568._1 & ((u8)1);
PF = temp_77696._1 == ((u8)0);
ADDR_101588:; // POP RBP
RBP = RAM(RSP)._8;
RSP = RSP + ((u64)8);
ADDR_101589:; // RET
RIP = RAM(RSP)._8;
RSP = RSP + ((u64)8)
```

Possiamo osservare come a ogni istruzione assembly corrispondano svariate righe di C, infatti è stata generata una riga di C per ogni istruzione p-code generata da Ghidra. Fortunatamente i compilatori C moderni sono eccellenti nell'ottimizzare le parti di codice inutilizzate, quindi la maggior parte delle istruzioni generate verrà inclusa nel binario liftato solo se effettivamente necessaria.

3.7 Plugin inclusi

3.7.1 Instruction counting plugin

Questo plugin è uno di quelli già inclusi in Dragonlifter e viene anche utilizzato internamente per il testing. Il suo scopo è quello di contare le istruzioni macchina eseguite dal binario liftato ed è un ottimo esempio di binary instrumentation.

In generale serve a misurare la lunghezza dell'esecuzione del binario e può essere sfruttato, per esempio, per scoprire la password richiesta dal un eseguibile (come dimostrato nella sezione 4.3).

Implementare questo plugin è molto semplice ed è un ottimo esempio della flessibilità di Dragonlifter. È infatti sufficiente aggiungere aggiungere l'incremento di una variabile globale ogni volta che un'istruzione viene liftata:

```
class InstructionLifterInstructionCount(InstructionLifter):
    def generate_body(self) -> str:
        return "\n".join((
            "++__instruction_count;",
            super().generate_body(),
        ))
```

È ovviamente anche necessario definire e inizializzare a `0` la variabile globale `__instruction_count` tramite `CoreLifter`:

```
class CoreLifterInstructionCount(CoreLifter):
    def setup(self):
        super().setup()
        self.c_fields.append("u64 __instruction_count = 0;")
        self.h_fields.append("extern u64 __instruction_count;")
```

3.7.2 Linux x86_64 syscalls plugin

Questo plugin è uno di quelli già inclusi in Dragonlifter e viene anche utilizzato internamente per far funzionare correttamente dei test.

Aggiunge il supporto alle syscall per Linux `x86_64` nel caso in cui sia il binario originale sia quello liftato siano per Linux `x86_64`. È banale modificarlo per aggiungere il supporto anche per altre architetture (e.g., `x86`, `ARM`), mantenendo tuttavia il vincolo che anche il binario liftato venga compilato per

la stessa architettura.

Il p-code non modella il comportamento delle syscall, quindi uno degli utilizzi principali dei plugin all'interno di Dragonlifter è proprio quello di fornire un modo semplice per generare codice per supportare le syscall dell'architettura originale.

Per fare questo ci possono essere vari approcci. Nel caso in cui l'architettura originale sia diversa da quella attuale, l'unico approccio possibile è quello di emulare completamente le syscall che sappiamo essere usate dal binario. Nel nostro caso però, l'architettura è la stessa e quindi sappiamo che le syscall utilizzate sono le stesse identiche che abbiamo a disposizione anche nella nostra macchina. Per questo possiamo prendere una scorciatoia e utilizzare direttamente le syscall vere messe a disposizione dal nostro sistema operativo, invece che doverle emulare.

Per richiamare una syscall in C, è necessario usare del codice simile a questo:

```
__asm__ __volatile__(
   "syscall"
   : "=rax"(ret)
   : "a"(rax), "D"(rdi), "S"(rsi), "d"(rdx)
   : "rcx", "r11", "memory"
);
```

Dove dentro a `rax` è presente il numero della syscall da chiamare e `rdi`, `rsi` e `rdx` sono gli argomenti che gli vengono passati. Dopo l'esecuzione, all'interno della variabile `ret` ci sarà il valore di ritorno della syscall.

Per esempio, per eseguire la syscall `write` è necessario settare `rax` a `1`, scrivere dentro a `rdi` il file descriptor in cui si vuole scrivere, dentro a `rsi` il puntatore al buffer contenente i byte da scrivere e in `rdx` il numero di byte da scrivere.

In generale, possiamo dire che per sfruttare questo codice all'interno del nostro plugin è sufficiente passargli i valori dei registri emulati all'interno del codice liftato. Questo funziona senza problemi nel caso di syscall semplici che non hanno a che fare con la memoria, per esempio `exit`. Tuttavia dato che il nostro lifter emula anche lo spazio di memoria, gli indirizzi presenti nel programma non corrispondono agli indirizzi fisici della nostra macchina. Normalmente tutti gli accessi alla memoria passano da `RAM_ADDR`, che si occupa di tradurre da indirizzo emulato a indirizzo vero. Nel caso delle syscall però,

l'accesso alla memoria viene fatto dal sistema operativo e quindi non è più controllato dal lifter.

Per esempio il programma potrebbe volere fare una `write` di un buffer a indirizzo `1234` (quindi dentro a `RSI` ci sarà scritto `1234`), tuttavia nella nostra macchina fisica questo indirizzo corrisponde a `561234`. Ci serve quindi un modo per sapere quando un argomento viene utilizzato come puntatore alla memoria e solo in quei casi convertirlo in un indirizzo vero. Nel caso della `write`, vogliamo trattare `RSI` come un indirizzo e quindi convertirlo, ma vogliamo lasciare `RDI` e `RDX` al loro valore originale in quanto costanti.

La soluzione applicata da questo plugin è stata quella di ottenere un elenco di tutte le syscall e dei loro parametri. In automatico, per ogni syscall, capire se il parametro è un puntatore alla memoria o meno e in tal caso generare del codice per convertire il registro a un indirizzo, altrimenti passare il valore non modificato.

Il codice generato alla fine è un grande `switch` su RAX dove, per ogni syscall, vengono passati i registri necessari. Per esempio la `write` assomiglia a:

```
__syscall(RAX, RDI, RAM_ADDR(RSI), RDX);
```

3.7.3 int128 plugin

Questo plugin è uno di quelli già inclusi in Dragonlifter e viene anche utilizzato internamente per far funzionare correttamente dei test.

Aggiunge il supporto agli interi a 128 bit, necessario per esempio se il binario è per `x86_64` e utilizza istruzioni SSE.

Per esprimere un intero a 128 bit, nel codice liftato viene utilizzato il tipo `__int128`. Un'estensione non standard supportata dai principali compilatori `C` (GCC e clang). Il motivo per cui non è stato incluso di default ma solo tramite plugin, è proprio il fatto che non siano sempre supportati essendo non standard.

Il plugin semplicemente aggiunge due nuovi typedef:

```
typedef __int128 i128;
typedef unsigned __int128 u128;
```

E aggiunge due nuovi campi a `varnode_t` grandi 16 byte:

```
typedef union {
  [...]
  i128 _16s;
  u128 _16;
} varnode_t;
```

Per implementare questo plugin è sufficiente aggiungere `i128` e `u128` ai typedef generici:

```
class CoreLifter128(CoreLifter):
    def generate_generic_typedefs(self) -> str:
        return '\n'.join((
            super().generate_generic_typedefs(),
            'typedef __int128 i128;',
            'typedef unsigned __int128 u128;',
        ))
```

E aggiungerli ai tipi disponibili all'interno di `CoreContext`:

```
class CoreContext128(CoreContext):
    available_types = CoreContext.available_types | {
        (VarKind.SIGNED, 16): 'i128',
        (VarKind.UNSIGNED, 16): 'u128',
    }
```

128bit bug

L'introduzione di questo plugin ha causato dei problemi in alcuni binari liftati: una volta ricompilati (usando GCC) andavano in segmentation fault su istruzioni apparentemente innocue che però utilizzavano registri a 128 bit.

Il problema non era causato direttamente da questo plugin, ma era necessario che fosse abilitato affinché ci fossero le condizioni necessarie per causare problemi. Procediamo quindi a un'analisi passo per passo del crash e a successive ipotesi e soluzioni.

La riga di codice C liftato che causava il crash era simile alla seguente:

```
temp_42._16s = RAXs
```

Che genera il seguente codice `x86_64`:

```
mov QWORD PTR [rbp-0x40], rax
sar rax, 0x3f
mov QWORD PTR [rbp-0x38], rax
movdqa xmm0, XMMWORD PTR [rbp-0x40]
movaps XMMWORD PTR [rip+0x827ef5], xmm0
```

Il codice sopra sta semplicemente assegnando un valore a 64 bit con segno a uno a 128 bit con segno. Ad alto livello, la strategia usata per fare questo consiste in: prendere due "blocchi" di memoria adiacenti da 64 bit, in uno scrivere il valore da copiare e nell'altro il segno (tutti 1 se negativo, tutti 0 se positivo), infine interpretare questi due blocchi da 64 bit come uno unico da 128 bit.

Nello specifico dell'assembly mostrato sopra:

- La prima `mov` copia `rax` in un blocco nello stack a `rbp-0x40`
- `sar` riempie di `1` o `0` `rax`, in funzione dell'ultimo bit (`1` se negativo, `0` se positivo)
- La seconda `mov` copia `rax` (ora contenente il segno) nell'altro blocco, a `rbp-0x38`
- `movdqa` interpreta i due blocchi come un unico blocco da 128 bit e lo scrive in un registro a 128 bit (`xmm0`)
- `movaps` scrive il valore del registro `xmm0` nella posizione finale in memoria

Da notare che in questo specifico esempio il compilatore poteva essere più intelligente e scrivere direttamente nell'area di memoria interessata, senza passare dallo stack (e infatti compilando con le ottimizzazioni è quello che accade). Tuttavia ci sono altri casi meno banali in cui questo non è possibile e si presenta quindi lo stesso crash.

Il codice mostrato sopra va in segmentation fault nella riga in cui fa `movdqa`, si può quindi presumere che ci sia qualche problema durante l'accesso alla memoria puntata da `rbp-0x40`. La memoria puntata di per sé non ha nessun problema di permessi (è all'interno dello stack, che è ovviamente sia leggibile che scrivibile dall'attuale processo).

Il motivo del crash è che, in questo caso, `rbp` non è allineato a 16 byte ma a 8 e le operazioni SSE possono causare segmentation fault nel caso in cui non vengano eseguite allineate alla loro dimensione (16 byte per questa istruzione).

GCC dovrebbe tuttavia rendersi conto che quella funzione usa istruzioni SSE che richiedono di essere allineate; inoltre su `x86_64` dovrebbe allineare lo stack a 16 byte di default. Per quale motivo quindi è disallineato? Il problema sta nel fatto che, in questo specifico esempio, durante la compilazione del codice liftato viene usata la flag `-nostartfiles`, che fa sì che non venga utilizzata la funzione di entry-point `_start` di GCC ma di riutilizzare quella già esistente nel binario liftato. Questo viene fatto per semplificare i test all'interno di Dragonlifter e sarebbe possibile cambiarlo, tuttavia è scorretto forzare tutti gli utenti del lifter a fare lo stesso.

Il motivo per cui questo causa problemi è che è proprio la funzione `_start` di GCC ad assicurarsi che lo stack sia allineato correttamente all'avvio. Una volta compreso completamente il problema, è facile trovare soluzioni per risolverlo. Infatti è sufficiente forzare l'allineamento dello stack all'ingresso di ogni funzione, tramite la flag `-mstackrealign`.

In un binario compilato per `x86_64`, questa flag inserisce all'inizio di ogni funzione il seguente assembly che forza l'ultimo nibble di `rsp` a `0` e quindi l'allineamento a 16 byte:

```
and rsp, 0xfffffffffffffff0
```

Capitolo 4

Validazione

In questo capitolo si descrivono i metodi utilizzati per validare la correttezza del codice generato.

4.1 PcodeLifter test

Ogni istruzione p-code è sottoposta a vari test per verificare che il comportamento del codice generato sia quello desiderato. Il test prende il p-code da testare, un elenco di valori in input e si assicura che il valore restituito in output sia quello previsto.

Ai fini del test, vengono creati dei varnode di input di tipo costante contenenti i valori specificati nella chiamata al test e un varnode temporaneo di output, per poter leggere il risultato. Questi varnode vengono poi passati al lifter che genera il corrispettivo codice C. Il codice C generato viene inserito all'interno del `main` di un programma molto minimale, contente solo il codice liftato e alla una `printf` del valore contenuto nel varnode di output. In questo modo è possibile compilare il programma generato, eseguirlo e ottenere l'output calcolato con quegli input.

Il programma generato viene dato in input a TCC, un compilatore C noto per la sua velocità di compilazione. TCC offre anche la possibilità di compilare ed eseguire direttamente il codice in un solo comando tramite la flag `-run`, il che si è rivelato comodo per mantenere la complessità del codice bassa (non è necessario creare e gestire file temporanei) ed eseguire i test più velocemente.

Per esempio, alcuni test per `INT_ADD` assomigliano a:

```
arithmetic_test(Op.INT_ADD, [1, 2], 3)
```

```
arithmetic_test(Op.INT_ADD, [255, 1], 0)
arithmetic_test(Op.INT_ADD, [1, 255], 0)
```

In questo caso i varnode creati sono da 1 byte (ma è possibile specificare dimensioni differenti), quindi `255+1` va in overflow ed è corretto ritorni `0`.

Bug di TCC

Dato che attualmente ci sono più di 300 test solo per il lifting del p-code, è stato significativo mantenere le iterazioni di test rapide grazie alla velocità di TCC.

Usare TCC si è tuttavia rivelato problematico in certi casi particolari, nello specifico certe operazioni floating point con `NaN` non ritornano il valore previsto:

- `FLOAT_NEG(NaN)` dovrebbe ritornare `-NaN` ma ritorna `NaN`.
- `FLOAT_ABS(-NaN)` dovrebbe ritornare `NaN` ma su Windows ritorna `-NaN`.

Il problema della negazione è stato in realtà risolto in un commit[1] di Gennaio 2021, ma ancora non c'è stata una release ufficiale di TCC che lo includa.

4.2 Output test

Un altro tipo di test più generico, che copre il codice nella sua interezza, è quello dell'output.

È stato appositamente creato un sorgente che prende input da `stdin`, fa calcoli vari e restituisce dei valori su `stdout`. Il modo in cui fa i calcoli e in generale la struttura del codice, è stato appositamente pensato per coprire casi particolari tipo jump table (switch) o chiamate a funzioni indirette (tramite un array di puntatori a funzioni).

Il test esegue i seguenti passi:

- Compilare il sorgente in un binario
- Liftare il binario
- Compilare i sorgenti liftati

[1] `https://github.com/TinyCC/tinycc/commit/29d8871d6196819caff83b4359e22cc3655bc234`

- Eseguire il binario originale con un determinato input
- Eseguire il binario liftato con lo stesso input
- Verificare che l'output e il codice di uscita dei due binari sia identico

Durante il lifting vengono utilizzati due plugin necessari a produrre un binario funzionante: quello per supportare le syscall Linux `x86_64` su un sistema Linux `x86_64` e quello per gli interi a 128 bit.

In questo caso i binari vengono compilati da GCC perché vengono utilizzate estensioni non standard, sia per implementare le syscall che per gli `int128`.

4.3 Instruction counting test

Il plugin di instruction counting viene usato per effettuare dei test che permettono di validare il funzionamento generale del lifter e le sua capacità di binary instrumentation.

Nello specifico, è stato creato un apposito eseguibile che richiede una password in input, con una logica di funzionamento simile al seguente codice:

```
char * in = input("password: ");
for (int i = 0; i < password_len; i++) {
    if (password[i] != in[i])
        return false;
}
return true;
```

Durante ogni iterazione del ciclo `for`, viene comparato il prossimo carattere della password inserita con quella vera: nel caso in cui non siano uguali allora esce subito, altrimenti continua con il prossimo carattere.

Una cosa importante da notare è il fatto che il numero di controlli, e quindi di istruzioni, eseguiti nel caso in cui venga inserita la password corretta è sempre maggiore rispetto al caso in venga inserita una password errata. Più nello specifico, possiamo dire che il numero di istruzioni eseguite è proporzionale al suffisso di caratteri corretti nella password inserita.

Questo test si occupa proprio di verificare che questa osservazione che abbiamo fatto sia vera utilizzando il plugin di instruction counting. Ipotizziamo che la password corretta sia "`password`" e di ottenere il conteggio delle istruzioni eseguite per i seguenti input:

```
one_letter_wrong1 = execute_count("a")
one_letter_wrong2 = execute_count("b")
one_letter_right = execute_count("p")
second_letter_wrong = execute_count("pxssword")
third_letter_wrong = execute_count("paxsword")
correct = execute_count("password")
```

Allora dovranno essere veri tutti i seguenti assert:

```
assert one_letter_wrong1 == one_letter_wrong2
assert one_letter_right > one_letter_wrong1
assert third_letter_wrong > second_letter_wrong
assert correct > third_letter_wrong
```

Conclusioni

Dragonlifter permette di liftare a codice C tutte le istruzioni p-code generate da Ghidra e di emulare certi aspetti dell'architettura sottostante.

Dragonlifter è già in grado di liftare interi binari e di generare un nuovo eseguibile con lo stesso comportamento di quello originale. Si presenta quindi come una valida alternativa agli altri binary lifter attualmente esistenti e, nel caso di architetture poco comuni, probabilmente l'unica opzione.

Grazie alla possibilità di estenderlo tramite plugin in tutte le sue fasi, è semplice instrumentare il codice generato, emulare le system call e in generale aggiungere il supporto alle funzionalità mancanti. Il fatto di di non richiedere LLVM e di generare codice C lo rende inoltre accessibile da chiunque, senza un oneroso setup iniziale.

Ovviamente, per poterlo comparare equamente ai lifter più maturi, sarà necessario testarlo più a fondo su esempi più complessi; inoltre la correttezza del lifting dipende anche dalla qualità del codice p-code generato da Ghidra che però, essendo open source, sarà possibile migliorare.

Sviluppi futuri

- Testare più a fondo il codice liftato:
  - Migliorare i test già esistenti per coprire più casi particolari
  - Creare un test che, dopo ogni istruzione eseguita, compara lo stato dei registri e della memoria del binario liftato con quello originale
- Creare un plugin per allocare i blocchi di memoria al loro indirizzo originale. Questo permette di:
  - Rendere significativamente più veloce e semplice l'accesso alla memoria

  - Semplificare le chiamate a funzioni di librerie esterne in quanto non sarà più necessario tradurre da indirizzo "emulato" a quello fisico

- Testare quanto sia oneroso l'impatto che il lifting ha sulle performance
- Emulare in automatico tutte le system call (e.g., tramite QEMU)

Bibliografia

[1] National Security Agency. Ghidra: A software reverse engineering suite. https://ghidra-sre.org.

[2] Nsa. https://www.nsa.gov.

[3] Capstone. https://www.capstone-engine.org.

[4] Zydis. https://zydis.re.

[5] GNU Project. GDB. https://www.gnu.org/software/gdb/gdb.html.

[6] Hex-Rays. IDA Pro. https://hex-rays.com/ida-pro.

[7] Hex-Rays. Hex-Rays Decompiler. https://hex-rays.com/decompiler.

[8] Vector 35. Binary Ninja. https://binary.ninja.

[9] JetBrains. Fernflower. https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine.

[10] ILSpy. https://github.com/icsharpcode/ILSpy.

[11] Michael A. Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 175–183, 2010.

[12] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 190–200, New York, NY, USA, 2005. Association for Computing Machinery.

[13] Lukáš Ďurfina, Jakub Křoustek, Petr Zemek, Dušan Kolář, Tomas Hruska, Karel Masařík, and Alexander Meduna. Design of a retargetable decompiler for a static platform-independent malware analysis. volume 200, pages 72–86, 08 2011.

[14] Bryan Ford and Russ Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX 2008 Annual Technical Conference*, ATC'08, page 293–306, USA, 2008. USENIX Association.

[15] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*, pages 559–573, 2013.

[16] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009.

[17] G. Wróblewski. General method of program code obfuscation. 2002.

[18] LLVM IR. `https://llvm.org/docs/LangRef.html`.

[19] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis amp; transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[20] Trail of Bits. McSema. `https://github.com/lifting-bits/mcsema`.

[21] Trail of Bits. Remill. `https://github.com/lifting-bits/remill`.

[22] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. Rev.ng: A unified binary analysis framework to recover cfgs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, page 131–141, New York, NY, USA, 2017. Association for Computing Machinery.

[23] Fabrice Bellard. Qemu, a fast and portable dynamic translator. ATEC '05, page 41, USA, 2005. USENIX Association.

[24] S. Bharadwaj Yadavalli and Aaron Smith. Raising binaries to llvm ir with mctoll (wip paper). LCTES 2019, page 213–218, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Samuele Turci and Giovanni Di Santi. Ghidra2Dwarf. `https://github.com/cesena/ghidra2dwarf`.

[26] SLEIGH. `https://github.com/NationalSecurityAgency/ghidra/blob/master/GhidraDocs/languages/html/sleigh.html`.

[27] p-code. `https://github.com/NationalSecurityAgency/ghidra/blob/master/GhidraDocs/languages/html/pcoderef.html`.

[28] GhidraScript. `https://ghidra.re/ghidra_docs/api/ghidra/app/script/GhidraScript.html`.

[29] Josh Juneau, Jim Baker, Frank Wierzbicki, Leo Soto, and Victor Ng. *The Definitive Guide to Jython: Python for the Java Platform*. Apress, USA, 1st edition, 2010.