UNIVERSITÀ
DI PARMA

Dipartimento di Scienze Matematiche, Fisiche e Informatiche

Corso di Laurea Triennale in Informatica

Valutazione sperimentale sull'individuazione automatica di errori di programmazione nel codice generato da LLM

Candidato:
Manuel Di Agostino

Matricola:
332233

Relatore:
Prof. Enea Zaffanella

Correlatore:
Prof. Vincenzo Arceri

Anno Accademico 2023–2024

Alla mia famiglia

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei codici

Capitolo 1

Introduzione

Negli ultimi anni si è assistito a un imponente sviluppo dei *Large Language Model* e a una loro progressiva integrazione nella pipeline di sviluppo del software. L'obiettivo primario è quello di migliorare le attuali tecniche di Automatic Code Generation (e.g., Code templates, Domain Specific Language) e aumentare di conseguenza la produttività degli sviluppatori a fronte di attività spesso ripetitive.

Sebbene i LLM siano abili nella generazione di contenuto che replica il linguaggio naturale, precedenti studi evidenziano che il codice da essi generato può essere insicuro. Questo aspetto è influenzato dal fatto che i dati e i programmi su cui i modelli vengono allenati non rispecchiano adeguatamente casi d'uso reali, la cui peculiarità è proprio quella di garantire elevati standard di sicurezza. È necessario inoltre considerare che gli attuali benchmark valutano la *correttezza* dei programmi prodotti, tralasciando considerazioni inerenti alla mancanza di *vulnerabilità*. In contesti lavorativi che quotidianamente includono strumenti basati su Artificial Intelligence (e.g., Copilot [1]) per la produzione e verifica del software, è dunque essenziale riuscire a misurare l'affidabilità del codice da questi prodotto.

Obiettivo di questa tesi è quello di effettuare una valutazione sulla qualità e sicurezza del codice generato dai LLM e sulla loro abilità di correggere eventuali vulnerabilità in esso presenti. Ai fini dello studio, sono stati selezionati otto modelli open-source appartenenti alle famiglie di Llama 2 [2], Code Llama [3] e Mistral [4, 5]. Tale scelta è stata effettuata sulla base del differente livello di *fine-tuning*[1] applicato a modelli di egual architettura (e.g., Code Llama e Llama 2) e sull'utilizzo di modelli dalle caratteristiche eterogenee (e.g., Llama 2 e Mistral).

I LLM sono stati utilizzati per generare un esteso insieme di sorgenti in C; esso ha costituito il benchmark sul quale poter valutare l'affidabilità e le capacità dei modelli sopra citati. Tale valutazione è frutto dei risultati ottenuti tramite l'utilizzo di due analizzatori statici: Mopsa [6, 7] e Infer [8, 9, 10]. Entrambi i

[1]Il *fine-tuning* è una tecnica utilizzata nel machine learning per specializzare un modello linguistico pre-allenato su di uno specifico sottoinsieme di task, come ad esempio la generazione di codice.

tool permettono la valutazione automatica del codice sorgente, rilevando comuni vulnerabilità ed errori (e.g., *memory-leak*, *overflow/underflow* aritmetici, ecc.) che rendono il software inaffidabile.

Infine, è stata esaminata la capacità dei LLM di correggere i problemi rilevati nei codici sorgente. Ciò è avvenuto allegando nel prompt di generazione un resoconto dettagliato delle vulnerabilità trovate assieme al task d'interesse; l'ulteriore analisi tramite MOPSA e INFER ha poi permesso di stabilire se il nuovo output fosse più o meno affidabile rispetto al precedente.

È possibile riassumere il contenuto di questo documento come segue:

1. nel Capitolo 2 sono esposte e descritte le tecniche di analisi utilizzate, assieme a una disamina degli analizzatori MOPSA e INFER; in particolare, sono presentate anche le modalità di integrazione della libreria PPLITE [11] con il primo, parte integrante del mio lavoro di tirocinio;

2. il Capitolo 3 racchiude la descrizione delle varie fasi di cui l'esperimento si compone;

3. nel Capitolo 4 sono delineate le modalità di analisi dei task generati, con particolare riguardo alle attività che si sono rese necessarie per poter completare correttamente l'analisi con MOPSA;

4. il Capitolo 5 presenta i risultati della prima valutazione dei sorgenti ottenuti e quelli relativi alla nuova generazione, ottenuta a partire dal feedback restituito dagli analizzatori;

5. per concludere, nel Capitolo 6 vengono messi in relazione gli obiettivi dell'esperimento con i risultati ottenuti.

La valutazione è stata realizzata in collaborazione con: il prof. Enea Zaffanella, relatore di questa tesi; il prof. Vincenzo Arceri, correlatore di questa tesi; Greta Dolcetti, studentessa di dottorato presso l'Università Ca' Foscari di Venezia; Saverio Mattia Merenda, laureando presso l'Università di Parma. Particolare enfasi è posta sul contributo personale che ho fornito al fine di permettere l'utilizzo di MOPSA per analizzare il codice prodotto dai modelli.

Capitolo 2

Background

2.1 Analisi statica

Quando osserviamo la definizione di un metodo, di un ciclo `for` o di una qualsiasi altra porzione di codice, ci interroghiamo sul suo comportamento, speculando sul significato e sulle possibili esecuzioni. Mentalmente effettuiamo quello che può essere considerato un esempio di **analisi statica** del codice sorgente, ossia il ricavarne un certo numero di proprietà, informazioni, senza però realmente eseguirlo su di una macchina. Ciò può essere svolto in maniera automatica da un altro programma, detto per l'appunto *analizzatore statico*.

A differenza di altri metodi formali, come ad esempio la Logica di Hoare [12] o il Model Checking di Clarke et al. [13], l'analisi statica inferisce delle proprietà che sono *matematicamente dimostrabili* senza l'intervento umano; questo avviene attraverso la formalizzazione della semantica del programma e l'utilizzo di un certo livello di astrazione, dovuto all'incomputabilità di determinate caratteristiche. L'introduzione di tale approssimazione comporta spesso l'incompletezza[1] di un sistema formale di questo tipo ma al contempo permette di tralasciare dettagli per semplificare la verifica.

2.2 Interpretazione Astratta

L'**Interpretazione Astratta (AbsInt)** viene introdotta da Cousot e Cousot [14] come teoria di approssimazione della semantica formale dei programmi. Essa mette in correlazione l'esecuzione vera e propria di un programma con un'altra definita invece *astratta*; se la prima manipola oggetti in un tipo di universo

[1] L'*incompletezza* deriva dalla necessità di approssimare in tempo finito un certo insieme di proprietà indecidibili. Di fatto, qualsiasi metodo di analisi statica computabile deve poter essere incompleto, ossia non può essere in grado di fornire risposte *definitive* e al contempo *corrette* per tutti i possibili programmi analizzabili.

concreto (e.g., interi di macchina, indirizzi di memoria etc.), la seconda tratta astrazioni formali di questi ultimi. Il vantaggio è quello di riuscire a calcolare, attraverso opportune approssimazioni, un output da poter mettere in relazione con quello dell'esecuzione concreta, spesso incomputabile.

Una delle applicazioni più immediate dell'AbsInt è quella del calcolo di proprietà numeriche. Si consideri il seguente frammento di codice:

```
1 int foo(int x, int y) {
2   while
3     (x < y) {
4     x = x - 1;
5     y = y + 1;
6   }
7   return y;
8 }
```

Codice 2.1: Esempio di funzione che manipola interi con segno.

Come farebbe un qualunque debugger, per controllare il comportamento di questa funzione è possibile tenere traccia dei valori delle variabili ad ogni step di esecuzione. L'evoluzione dello *stato* dell'esecuzione alla riga $i \in \mathbb{N}$ è quindi rappresentabile tramite la tupla $\langle i, x, y \rangle$. Considerando come valori iniziali $\{x \leftarrow 19, y \leftarrow 13\}$, si ottiene:

$$\begin{aligned}\langle 1,19,13\rangle &\to \langle 2,19,13\rangle \to \langle 3,19,13\rangle \to \langle 4,18,13\rangle \to \langle 5,18,14\rangle \\ &\to \langle 2,18,14\rangle \to \langle 3,18,14\rangle \to \langle 4,17,14\rangle \to \langle 5,17,15\rangle \\ &\to \langle 2,17,15\rangle \to \langle 3,17,15\rangle \to \langle 4,16,15\rangle \to \langle 5,16,16\rangle \\ &\to \langle 2,16,16\rangle \to \langle 3,16,16\rangle \\ &\to \langle 6,16,16\rangle \to \langle 7,16,16\rangle\end{aligned}$$

Intuitivamente, se all'entrata della funzione x e y sono positive e verificano $x > y$ il valore restituito dalla funzione sarà anch'esso positivo. Un compilatore potrebbe sfruttare questa informazione introducendo ottimizzazioni a basso livello per la computazione, utilizzando il tipo *unsigned*. A questo scopo, tramite un opportuno *dominio astratto* si possono staticamente inferire, ad esempio, proprietà relative al segno di un insieme di variabili. Analogamente, qualora si riuscisse a dimostrare che il valore di x appartenga ad uno specifico intervallo limitato, si potrebbe utilizzare tale informazione per eseguire l'*unrolling*[2] del ciclo `while` in fase di compilazione.

[2]Tecnica utilizzata nei compilatori per trasformare ove possibile i cicli, riducendo le istruzioni sul controllo di flusso con l'obiettivo di diminuire il tempo di esecuzione.

2.2.1 Esempio informale: i segni

Si consideri l'insieme dei *segni* $\{(+), (-)\}$. Associando ogni variabile intera del Codice 2.1 al corrispondente segno (per convenzione si associa 0 a $(+)$) e sfruttando le regole dei segni per definire la semantica delle operazioni tra interi, è possibile rimpiazzare l'esecuzione concreta con una astratta. Si supponga che la funzione `foo` venga invocata con argomenti positivi e che soddisfino $x > y$. A questo punto, i valori iniziali delle variabili precedentemente considerate diventano $\{x \leftarrow (+), y \leftarrow (+)\}$. Ricordando inoltre che

$$\begin{aligned}(+) * (+) &= (+)\\ (+) + (+) &= (+)\end{aligned}$$

ne risulta l'esecuzione:

$$\begin{aligned}\langle 1, (+), (+)\rangle \rightarrow \langle 2, (+), (+)\rangle \rightarrow \langle 3, (+), (+)\rangle\\ \rightarrow \langle 4, (+), (+)\rangle \rightarrow \langle 5, \top, (+)\rangle\,;\end{aligned}$$

Si noti che il test `(x > y)` viene interpretato come $(+) > (+) = (+)$, ossia si considerano tutti quei valori di x che rendono vera la guardia del ciclo: infatti, se il test del `while` è verificato e $y \geq 0$, necessariamente x è non negativa; inoltre, l'esecuzione di `x = x-1;` alla riga 4 comporta la perdita di informazione sul segno di x, identificata dal simbolo $\top$. In sintesi vale

$$(+) - (+) = \top$$

Dopo la riga 5, l'esecuzione torna a valutare il test del ciclo. Questa volta, la valutazione per x è

$$\top \geq (+) = (+)$$

Questo significa che si entrerebbe nel corpo del `while` con lo stato $\langle(+), (+)\rangle$, lo stesso dell'ultima valutazione del test. Intuitivamente, si potrebbe eseguire il corpo un numero arbitrario di volte, senza però aggiungere nuove informazioni; nel punto 3 si otterrà sempre lo stato $\langle(+), (+)\rangle$, mentre al punto 5 si avrà $\langle\top, (+)\rangle$. Nella teoria dell'AbsInt questa proprietà prende il nome di **punto fisso** (o *fixpoint* in inglese) e, oltre a essere sinonimo di un'analisi corretta, permette di determinare quando terminare la computazione.

Per poter tenere traccia di tutti i possibili flussi di esecuzione, si è soliti associare agli statement che coinvolgono un ciclo la cosiddetta **semantica collecting**; essa mira a tenere traccia dell'insieme di tutti gli stati che attraversano un particolare punto di programma. Si prenda come esempio la riga 3 che precede la valutazione del test `(x > y)`; se alla prima iterazione l'unico stato da considerare è $\langle(+), (+)\rangle$ (in ingresso alla funzione), quella successiva deve tener traccia anche

dello stato $\langle \top, (+) \rangle$, derivante dell'esecuzione del corpo del `while`. Lo stato che meglio approssima i precedenti in quel passo di esecuzione è quello rappresentato dal loro *least upper bound*[3] , ossia $\langle \top, (+) \rangle$. Associare la valutazione del test a riga 3 a questo stato vuol dire affermare che, computato il corpo del ciclo, non è più possibile inferire nulla riguardo al segno della variabile x mentre si è certi della non negatività di y.

A tal punto, l'esecuzione può proseguire giungendo al termine:

$$\begin{aligned}\langle 1, (+), (+) \rangle &\to \langle 2, (+), (+) \rangle \to \langle 3, (+), (+) \rangle \\ &\to \langle 4, (+), (+) \rangle \to \langle 5, \top, (+) \rangle \\ &\to \langle 6, \top, (+) \rangle \,.\end{aligned}$$

Considerando infatti il punto fisso raggiunto nella valutazione del loop, dalla riga 3 si procede direttamente a riga 6, *filtrando* dallo stato $\langle 6, \top, (+) \rangle$ i valori di entrambe le variabili che rendono falso il test. In particolare, `!(x > y)` viene valutato come

$$\top \leq (+) = \top$$

In soli 5 passi di esecuzione si è giunti alla conclusione che, qualsiasi siano $x, y \geq 0$, la funzione `foo` restituisce *sempre* un valore non negativo.

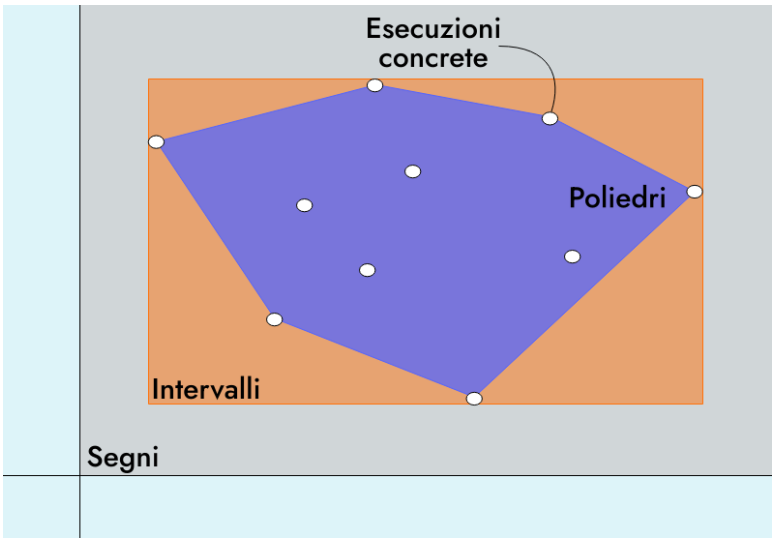
2.3 Risultati

Come evidenziato nelle sezioni precedenti, il compito dell'analisi statica tramite AbsInt è quello di approssimare l'esecuzione del programma, computandone un insieme di possibili tracce di esecuzione. Quanto più espressivo è il dominio astratto considerato, tanto più precisa sarà l'approssimazione che ne risulta. Questo comportamento è descritto in Figura 2.1 in cui vengono mostrate le differenti modalità di approssimazione di un insieme di tracce di esecuzione concrete (rappresentate tramite cerchi) utilizzando il dominio dei segni (piano grigio), degli intervalli (rettangolo arancione) e dei poliedri (poliedro convesso viola). È importante notare che l'insieme approssimato include *tutte* le esecuzioni concrete.

Definiamo **sound** un'analisi le cui invarianti calcolate siano valide per tutte le esecuzioni reali; d'altro canto, definiamo **unsound** un'analisi che tralasci un certo insieme di possibili tracce di esecuzione. Bisogna inoltre considerare la presenza di eventuali **falsi positivi**, ovvero la segnalazione di stati che in realtà non si verificheranno mai e che permettono all'astrazione di rimanere sound, a costo di perdere precisione. L'obiettivo della maggior parte degli analizzatori statici basati su Abstract Interpretation (AbsInt) è quello di evitare **falsi negativi**, ossia la certificazione di proprietà che in realtà vengono violate da almeno una possibile traccia di esecuzione, il che è sintomo di un'analisi unsound. Queste

[3]Per una trattazione più esaustiva dell'argomento si rimanda a [15].

Figura 2.1: Approssimazione della semantica concreta di un programma attraverso domini differenti: segni (grigio), intervalli (arancione), poliedri convessi (viola).



caratteristiche sono ben riassunte nella Figura 2.2. In genere, ad ogni programma è possibile associare un insieme di stati $\mathcal{P}$ che esso può raggiungere durante il suo ciclo di vita; inoltre, è solitamente richiesto che questi appartengano ad una determinata *specifica* $\mathcal{S}$ che identifica l'insieme di quelli considerati *sicuri*. Ad esempio, uno dei possibili requisiti potrebbe essere quello di asserire che il codice non generi mai accessi *illegali* alla memoria. Quindi, per un programma **safe** vale $\mathcal{P} \subseteq \mathcal{S}$ e nel caso di un'analisi sound, l'approssimazione $\mathcal{A}$ degli stati concretamente raggiungibili rispetta

$$\mathcal{P} \subseteq \mathcal{A} \subseteq \mathcal{S}$$

come mostrato in Fig. 2.2a. Qualora l'approssimazione ottenuta non sia conforme alla specifica, l'analisi risulta invece inconclusiva: in questo caso, il programma potrebbe egualmente essere soggetto a vulnerabilità o al contrario soddisfare i requisiti di sicurezza; quest'ultimo caso è riportato in Fig. 2.2b nella quale si evidenzia che in presenza di falsi positivi

$$\mathcal{P} \subseteq \mathcal{S} \wedge \mathcal{A} \not\subseteq \mathcal{S}$$

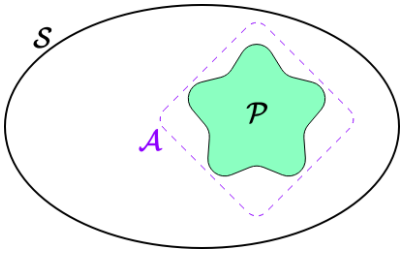
Il caso presentato in Fig. 2.2c caratterizza invece un'analisi unsound per la quale viene asserito che un programma *unsafe* rispetta la specifica; un falso negativo si verifica infatti se

$$\mathcal{P} \not\subseteq \mathcal{S} \wedge \mathcal{A} \subseteq \mathcal{S}$$

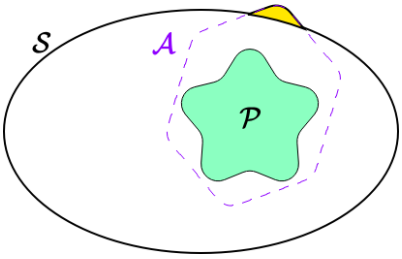
Si consideri come esempio il seguente frammento:

Figura 2.2: Raffigurazione dell'approssimazione di un programma $\mathcal{P}$ tramite astrazione $\mathcal{A}$: 2.2a analisi sound, 2.2b analisi sound ma con falsi positivi, 2.2c analisi unsound e presenza di falsi negativi.

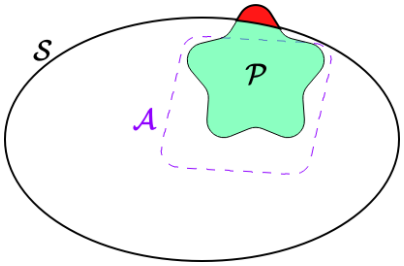
(a) analisi corretta, $\mathcal{P} \subseteq \mathcal{A} \subseteq \mathcal{S}$.



(b) falso positivo, $\mathcal{P} \subseteq \mathcal{S} \wedge \mathcal{A} \nsubseteq \mathcal{S}$.



(c) falso negativo, $\mathcal{P} \nsubseteq \mathcal{S} \wedge \mathcal{A} \subseteq \mathcal{S}$.




```
1  int main() {
2    int dim = 10;
3    int A[dim];
4
5    for (int i = 0; i <= dim; ++i) {
6      A[i] = 0;
7    }
8
9    return 0;
10 }
```

Codice 2.2: Esempio di main che manipola un array di interi.

Nell'ultima iterazione del ciclo, `dim` $\leftarrow$ `10` e lo statement `A[dim] = 0;` rappresenta un accesso *illegale* alla memoria. In casi come questo, un'analisi sound segnalerebbe una vulnerabilità, potenziale (*warning*) o definitiva (*error*), a seconda del dominio astratto considerato.[4] Si noti infine che in presenza di un falso negativo questo frammento sarebbe classificato come *safe*, privo di errori.

[4]La scelta determina la precisione sui valori assegnabili alle variabili considerate; un dominio di tipo relazionale permetterebbe di mettere in relazione `i` e `dim` computando esattamente i valori assunti da entrambe.

2.4 MOPSA

MOPSA [7] è un analizzatore statico open-source[5] multilinguaggio. Il progetto è stato finanziato dallo *European Research Council* (ERC) a partire dal 2016 e si è concluso nel 2021. L'obiettivo è stato quello di creare un insieme di strumenti in grado di rilevare, a tempo di compilazione, la presenza di vulnerabilità ed errori in software generici e di grandi dimensioni. Al momento della stesura di questa tesi, i linguaggi di programmazione supportati sono due (un sottoinsieme del `C` e un sottoinsieme di `Python3`) ma non si escludono sviluppi futuri in grado di estenderne le possibilità di utilizzo.

Essendo fortemente basato sulla teoria dell'Interpretazione Astratta, questo strumento si propone di progettare analisi che sono *approssimate*, in modo da essere scalabili su sorgenti di grandi dimensioni, e *sound*, così da garantire l'affidabilità dei risultati ottenuti. L'analisi avviene su di un programma nella sua interezza, ossia su tutto il codice sorgente, a partire da un entry-point (e.g., la funzione `main` per il `C`); inoltre, è assicurato che:

1. vengono presi in considerazione tutti i possibili percorsi d'esecuzione;
2. l'analisi termina in un tempo finito, grazie all'uso di operatori di **widening** (che forzano la convergenza del calcolo del punto fisso);
3. inoltre, l'analisi si conclude entro un tempo ragionevole, grazie all'eventuale uso di timeout.

Siccome MOPSA è uno strumento sound, esso è anche *incompleto*, ossia è prevista la possibilità che vengano segnalati errori che in realtà non appartengono a nessuna delle esecuzioni reali. Ciò è dovuto ad una perdita di precisione nell'astratto, a volte risolvibile scegliendo un'opportuna configurazione di analisi.

2.4.1 Utilizzo in breve e *Reports*

Tra i vari file binari resi disponibili per le analisi, è stato preso in considerazione `mopsa-c`. L'utilizzo[6]è intuitivo. Di default vengono riportati una lista dettagliata dei **check**[7] falliti e il numero di quelli che hanno invece avuto successo.

```
1 #define LEN 100
2
```

[5]L'intero codice sorgente è disponibile su GitLab: `https://gitlab.com/mopsa/mopsa-analyzer`.

[6]`https://mopsa.gitlab.io/mopsa-manual/user-manual/quick-start/usage.html`.

[7]Il nome del *check* (controllo) è relativo alla proprietà che si sta dimostrando, e.g., *Invalid memory access*, *Division by zero*, etc.. Quelli effettuati dipendono dalla configurazione scelta.

```
int main(int argc, char* argv[]) {
  int a[LEN];

  for(int i = 0; i <= LEN; i++)
    a[i] = 0;

  return 0;
}
```

Codice 2.3: Esempio di codice.

```
Analysis terminated successfully
Analysis time: 0.073s

Check #2:main.c: In function 'main':
main.c:7.4-8: warning: Invalid memory access

  7:      a[i] = 0;
          ^^^^
  accessing 4 bytes at offsets [4,400] of variable 'a' of
   size 400 bytes
  Callstack:
    from main.c:3.4-8: main

Checks summary: 2 total, 1 safe, 1 warning
  Invalid memory access: 1 total, 1 warning
  Integer overflow: 1 total, 1 safe
```

Codice 2.4: Esempio di report.

Il report può essere ottenuto anche in formato JSON con `-format=json`:

```
{
  "success": true,
  "time": 0.070330000000000004,
  "mopsa_version": "1.0~pre2",
  "mopsa_dev_version": "git branch:master commit:767350f5
   commit-date:2023-11-28 11:29:59 +0000",
  "file": [ "main.c" ],
  "checks": [
    {
      "kind": "warning",
      "title": "Invalid memory access",
      "messages": "accessing 4 bytes at offsets [4,400] of
   variable 'a' of size 400 bytes",
      "range": {
        "start": { "file": "main.c", "line": 7, "column": 4
   },
```

```
14          "end": { "file": "main.c", "line": 7, "column": 8 }
15        },
16        "callstack": [
17          {
18            "function": "main",
19            "range": {
20              "start": { "file": "main.c", "line": 3, "column"
    : 4 },
21              "end": { "file": "main.c", "line": 3, "column":
    8 }
22            }
23          }
24        ]
25      }
26    ],
27    "assumptions": []
28  }
```

Codice 2.5: Esempio di report in formato JSON.

MOPSA distingue tre differenti tipologie di risultati:

- `Safe`: il controllo è superato. In questo scenario, *tutti* i possibili flussi di esecuzione che attraversano il check sono considerati sicuri, perché lo soddisfano. D'altro canto, potrebbe essere coumunque considerato superato qualora *nessun* flusso di esecuzione lo raggiunga;

- `Error`: il controllo è invalidato da *almeno* un flusso di esecuzione. Viene asserito che è presente sicuramente una sequenza che attraversa il check e non lo soddisfa;

- `Warning`: la validità del controllo *non può* essere determinata. L'errore può esserci ma non viene rilevato a causa di un'approssimazione troppo generica. Il potenziale allarme è dovuto al fatto che esiste una duplice fonte di incertezza: da un lato, è presente almeno un flusso che potenzialmente attraversa quel punto di codice; dall'altro, c'è la possibilità che quest'ultimo fallisca. Tuttavia, se si è in grado di dimostrare che il flusso attraversa il check *e* lo invalida (i.e., non è presente nessuna delle due ambiguità sopra citate), l'allarme diviene un errore.

A volte, potrebbe essere riportata una `unsound analysis`; è il caso, ad esempio, dell'invocazione di funzioni che non sono state definite. In questo contesto, sono effettuate una serie di *assunzioni* per poter continuare l'analisi (e.g., si assume che la funzione in questione non modifichi variabili).

2.4.2 Le configurazioni

Il framework mette a disposizione la possibilità di utilizzare differenti domini astratti per realizzare l'analisi. Ognuno di questi differisce dall'altro nella rappresentazione delle variabili o ad esempio nel modo in cui vengono gestiti i cicli. La caratteritstica più importante è che questi sono considerati *moduli componibili* [16] al fine di realizzare un'analisi quanto più possibile completa. L'opzione `-config` permette di selezionare la configurazione desiderata; oltre a quelle già disponibili, è possibile inoltre crearne di personalizzate.

Tra quelle comuni, è possibile trovare:

- `c/cell-itv.json`: la configurazione predefinita, che include i domini degli intervalli di numeri interi e a virgola mobile, un dominio per gestire i cicli e un altro per l'analisi intra-procedurale;
- `c/cell-pack-rel-itv.json`: aggiunge la possibilità di sfruttare un dominio di tipo relazionale;[8]
- `c/cell-string-length-pack-rel-itv-congr.json`: mette a disposizione anche un dominio specifico per tenere traccia della lunghezza delle stringhe in stile C e il dominio delle *congruenze*.

2.4.3 Stubs

Uno **stub** è una definizione che viene aggiunta al codice sorgente per eseguire l'analisi. Può essere estremamente utile ricorrere all'utilizzo di stubs qualora il codice difetti della definizione di una o più funzioni (e.g., librerie esterne); essi permettono infatti di descriverne le *specifiche*, permettendo quindi di concludere l'analisi.

Un aspetto importante è dato dalla possibilità di rendere le specifiche generiche, introducendo imprecisione attraverso funzioni non deterministiche messe a disposizione dal framework. Possono essere utilizzate includendo l'header file `"mopsa.h"` e permettono di modellare la maggior parte dei tipi primitivi del C (e.g., `[signed|unsigned] int`, `float`, `void *`, etc.).

È possibile implementare uno stub attraversi due modalità:

1. utilizzando le funzioni in C del framework, come quelle riportate nel Codice 2.6;
2. sfruttando l'apposito linguaggio per contratti.[9]

[8] È possibile selezionare il dominio relazionale tramite l'opzione `-numeric (lineq | octagon | polyhedra)`.

[9] `https://mopsa.gitlab.io/mopsa-manual/user-manual/stubs/c-contracts.html#c-contracts`.

In particolare, la seconda opzione è utilizzata per specificare il comportamento della maggior parte delle funzioni della libreria standard del linguaggio C.

```
// interi
signed int _mopsa_rand_s32();
unsigned int _mopsa_rand_u32();

signed long _mopsa_rand_s64();
unsigned long _mopsa_rand_u64();

// virgola mobile
float _mopsa_rand_float();
double _mopsa_rand_double();

// puntatori
void *_mopsa_rand_void_pointer();
```

Codice 2.6: Esempi di funzioni built-in per gli stubs.

2.4.4 Integrazione della PPLite

Parte del progetto di tirocinio è consistito nell'integrazione della libreria PPLite [11] all'interno di Mopsa. L'obiettivo è stato quello di aggiungere la possibilità di utilizzare un dominio relazionale dei *poliedri convessi* differente rispetto a quello già reso disponibile da Apron [17].

Grazie al recente wrapper della libreria introdotto in Apron, si sono rese necessarie due modifiche al codice sorgente in `OCaml`:

1. nel file `instances.ml`[10], in cui veniva utilizzata la libreria Polka [18] per i domini relazionali:

```
diff --git a/analyzer/languages/universal/numeric/
   relational/instances.ml b/analyzer/languages/
   universal/numeric/relational/instances.ml
index 663f897f..19f8488a 100644
--- a/analyzer/languages/universal/numeric/relational/
   instances.ml
+++ b/analyzer/languages/universal/numeric/relational/
   instances.ml
@@ -35,6 +35,13 @@ module Polyhedra = Domain.Make(struct
   let man = Polka.manager_alloc_loose ()
 end)

```

[10] `https://gitlab.com/mopsa/mopsa-analyzer/-/blob/master/analyzer/languages/universal/numeric/relational/instances.ml`.

```
+module PplitePolyhedra = Domain.Make(struct
+  type t = Pplite.loose Pplite.t
+  let name = "universal.numeric.relational"
+  let man = Pplite.manager_alloc_loose ()
+  let () = Pplite.manager_set_kind man "F_Poly"
+end)
+
 module LinEqualities =
   Domain.Make(struct
       type t = Polka.equalities Polka.t
@@ -61,7 +68,7 @@ let () =
     category = "Numeric";
     doc = " select the relational numeric domain.";
     spec = ArgExt.Symbol (
-        ["octagon"; "polyhedra"; "lineq"],
+        ["octagon"; "polyhedra"; "pplite_polyhedra"; "
   lineq"],
         (function
           | "octagon"   ->
@@ -73,6 +80,11 @@ let () =
             numeric_domain := (module Polyhedra :
   RELATIONAL);
             register_simplified_domain (module
  Polyhedra)

+          | "pplite_polyhedra" ->
+            opt_numeric := "pplite_polyhedra";
+            numeric_domain := (module PplitePolyhedra :
+  RELATIONAL);
+            register_simplified_domain (module
  PplitePolyhedra)
+
           | "lineq" ->
             opt_numeric := "lineq";
             numeric_domain := (module LinEqualities :
  RELATIONAL);
```

Codice 2.7: Modifiche in `instances.ml`.

2. nel file `dune`,[11] per specificare le modalità di compilazione del progetto.

[11] `https://gitlab.com/mopsa/mopsa-analyzer/-/blob/master/analyzer/languages/universal/numeric/relational/dune`.

```
diff --git a/analyzer/languages/universal/numeric/
   relational/dune b/analyzer/languages/universal/
   numeric/relational/dune
index d2ae1f6d..0f534a4b 100644
--- a/analyzer/languages/universal/numeric/relational/
   dune
+++ b/analyzer/languages/universal/numeric/relational/
   dune
@@ -1,6 +1,6 @@
 (library
  (name relational)
  (public_name mopsa.mopsa_analyzer.universal.numeric.
   relational)
- (libraries numeric_common numeric_values lang apron
   apron.boxMPQ apron.octMPQ apron.polkaMPQ )
+ (libraries numeric_common numeric_values lang apron
   apron.boxMPQ apron.octMPQ apron.polkaMPQ apron.pplite
   )
  (library_flags -linkall)
  (flags :standard -open Lang -open Numeric_common -open
    Numeric_values))
```

Codice 2.8: Modifiche in `dune`.

Ricompilato Mopsa, viene quindi resa disponibile l'opzione da riga di comando `-numeric pplite_polyhedra` per utilizzarla in abbinamento ad un'opportuna configurazione.

2.4.5 Dockerfile

Per questioni di portabilità e facilità d'utilizzo, ho realizzato un `Dockerfile` che rende automatica l'installazione della versione modificata di Mopsa su container Linux. Inoltre, ho reso disponibile la build dell'immagine su DockerHub[12] con tag `mopsa:pplite`; è possibile installarla da riga di comando utilizzando il comando

```
$ docker pull manueldiagostino/mopsa:pplite
```

Di seguito riporto i file coinvolti nella realizzazione dell'immagine.

1. `Dockerfile`

[12] `https://hub.docker.com/layers/manueldiagostino/mopsa/pplite/images/sha256-06bd11658ba153426ad6e132a3e0aea74a28ac0edf1743162e615e1b5fec1408?context=repo`.

```
################################################
# Docker image with MOPSA installed on Ubuntu #
################################################

FROM ubuntu:latest

RUN \
    apt update && \
    apt install -y opam && \
    mkdir /home/mopsa && cd /home/mopsa && \
    opam init --yes && \
    eval `opam config env` && \
    mkdir workspace

WORKDIR /home/mopsa

######################
#  INSTALLING PPLITE #
######################

ENV MOPSA_HOME=/home/mopsa
ENV PPLITE_REPO=https://github.com/ezaffanella/PPLite
ENV APRON_REPO=https://github.com/antoinemine/apron
ENV MOPSA_REPO=https://gitlab.com/mopsa/mopsa-analyzer.
   git

WORKDIR ${MOPSA_HOME}

RUN \
    apt update && apt install -y           \
    make autoconf automake libtool         \
    libgmp-dev libmpfr-dev libflint-dev

RUN \
    cd ${MOPSA_HOME} && \
    git clone $PPLITE_REPO && \
    cd PPLite && \
    mkdir build && \
    autoreconf --install && \
    cd build && \
    ../configure && \
    make -j$(expr $(nproc) + 1) && \
    make install

ENV LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/lib:/usr/
```

```
   local/lib:${MOPSA_HOME}
45
46 RUN opam install --confirm-level=unsafe-yes apron
47
48 RUN \
49     cd ${MOPSA_HOME} && \
50     git clone ${MOPSA_REPO}
51
52 COPY script.sh ${MOPSA_HOME}/script.sh
53 RUN \
54     cd ${MOPSA_HOME} && \
55     chmod +x script.sh && \
56     ./script.sh
57
58 RUN \
59     cd ${MOPSA_HOME}/mopsa-analyzer && \
60     opam install --deps-only --with-doc --with-test --
   confirm-level=unsafe-yes .
61
62 RUN \
63     cd ${MOPSA_HOME}/mopsa-analyzer && \
64     eval $(opam env) && \
65     apt update && apt install -y clang && \
66     ./configure --prefix ${MOPSA_HOME} && \
67     make -j$(expr $(nproc) + 1) && \
68     make install
69
70 ENV PATH=${PATH}:/root/.opam/default/bin
```

Codice 2.9: Dockerfile per installazione su `ubuntu:latest`.

2. `script.sh`

```
1  #!/bin/bash
2
3  # Author: Di Agostino Manuel
4  # https://github.com/manueldiagostino
5
6  process() {
7      inFile="$1"
8      outFile="$2"
9      startingPoint="$3"
10     endingPoint="$4"
11     string="$5"
12
13
```

```
14      inside='false'
15      echo -n "" > $outFile
16
17      echo "$(date) : processing $inFile" >&1
18
19      while IFS= read -r line || [ -n "$line" ]; do
20      # echo "$line"
21
22      if [ $(echo $line | grep -c -e "$startingPoint") -eq
        1 ]; then
23          inside='true'
24          echo "$string" >>$outFile
25      else
26          if [ $inside == 'false' ]; then
27              echo "$line" >> $outFile
28          else
29              if [ $(echo $line | grep -c -e "$endingPoint
       ") -eq 1 ]; then
30                  inside='false'
31              fi
32          fi
33      fi
34      done < ${inFile}
35  }
36
37
38  # instances.ml
39  inFile="${MOPSA_HOME}/mopsa-analyzer/analyzer/languages/
       universal/numeric/relational/instances.ml"
40  outFile='out_instances.ml'
41
42  startingPoint='module Polyhedra = Domain.Make(struct'
43  endingPoint='end)'
44  string=$(cat <<HEREDOC
45  module Polyhedra = Domain.Make(struct
46    type t = Pplite.loose Pplite.t
47    let name = "universal.numeric.relational"
48    let man = Pplite.manager_alloc_loose ()
49    let () = Pplite.manager_set_kind man "F_Poly"
50  end)
51  HEREDOC
52
53  )
54
55  process "$inFile" "$outFile" "$startingPoint" "
```

```
    $endingPoint" "$string"
56 mv $outFile "${MOPSA_HOME}/mopsa-analyzer/analyzer/
    languages/universal/numeric/relational/instances.ml"
57
58
59 # dune
60 inFile="${MOPSA_HOME}/mopsa-analyzer/analyzer/languages/
    universal/numeric/relational/dune"
61
62 lineBefore='^[[:space:]]*(libraries.*apron.polkaMPQ[[:
    space:]]*)$'
63 lineAfter=' (libraries numeric_common numeric_values
    lang apron apron.boxMPQ apron.octMPQ apron.polkaMPQ
    apron.pplite)'
64
65 echo "$(date) : processing $inFile" >&1
66 sed -i -e "s/${lineBefore}/${lineAfter}/" $inFile
```

Codice 2.10: Script di supporto.

2.5 Infer

Infer [10] è uno strumento professionale di rilevazione statica di errori nel software, pensato per i linguaggi Java, C, C++ e Objective-C. Sviluppato in OCaml, l'intero codice sorgente è stato reso disponibile su GitHub.[13] Ad oggi viene costantemente utilizzato da *Meta* [19] per verificare la produzione del software destinato ai dispositivi Android e iOS. Sebbene all'inizio Infer fosse completamente fondato su logica di separazione e bi-abduction,[14] negli ultimi anni sono stati integrati moduli basati su interpretazione astratta e su *linters* (per la rilevazione di vulnerabilità sfruttando analisi sintattiche basate sulla visita dell'*AST* ricavato dal codice sorgente). Tecnicamente, ci si riferisce al modulo originario come *Infer.SL*, mentre al nuovo core basato sull'analisi statica ci si riferisce come *Infer.AI*.

Essendo pensato per assistere i programmatori durante l'intero ciclo di produzione del software, esso supporta l'analisi di programmi incompleti. È permesso inoltre effettuare analisi sia di tipo *intra-procedurale*, ossia concentrandosi su di una singola procedura o funzione, sia di tipo *inter-procedurale*, considerando quindi il flusso delle invocazioni ai metodi nel suo complesso.

2.5.1 Utilizzo

Sono distinte due fasi principali nell'utilizzo di Infer:

[13] `https://github.com/facebook/infer`.
[14] `https://fbinfer.com/docs/separation-logic-and-bi-abduction`.

1. fase di **cattura**. In questa prima fase vengono catturati i comandi di compilazione e oltre ad essere compilati, i sorgenti sono tradotti nel linguaggio interno utilizzato dal programma per effettuare l'analisi; pertanto è richiesto che ogni sorgente sia compilabile. Un esempio di utilizzo è il seguente: `infer capture -- clang -c hello.c`. Di default i file risultanti dalla cattura e traduzione sono salvati in `infer-out/` ma si può specificare un percorso diverso con l'opzione `-o <out_dir>`.

2. fase di **analisi**. A questo punto i file ricavati in `infer-out/` sono analizzati. Ogni metodo viene controllato separatamente e, qualora sia presente almeno un errore, l'analisi di quel metodo viene interrotta con una segnalazione; ciò significa che nelle altre procedure la ricerca di vulnerabilità non viene sospesa. L'analisi viene avviata con `infer analyze`.

In alternativa, è possibile unificare i due passaggi con

```
$ infer run -- gcc -c hello.c
```

2.5.2 Checkers

INFER è per costruzione un insieme di differenti componenti, ognuno dei quali specializzato in una determinata categoria di vulnerabilità. Il singolo modulo prende il nome di *Checker* ed è associato ad un tipo ben specifico di analisi. Ogni checker si occupa di rilevare un insieme di *Issue types*, ossia sottocategorie di problemi che il modulo in questione è in grado di verificare. Il framework non è dunque concepito per dimostrare l'assenza di errori nel codice; piuttosto, si occupa di segnalare potenziali bug conformemente a quali controlli sono stati attivati. Infatti, non tutti i checker sono attivati di default ma è possibile abilitare quello d'interesse tramite l'opzione `--<checker_name>`. Ad esempio per attivare i moduli BUFFEROVERRUN, UNINITIALIZED VALUE e PULSE, è sufficiente digitare il comando:

```
$ infer run          \
    --bufferoverrun \
    --uninit         \
    --pulse          \
    -- gcc prova.c
```

Codice 2.11: Esempio di comando per l'attivazione di moduli multipli.

2.5.3 Reports

Una volta che l'analisi è stata effettuata, il sommario viene mostrato a video; inoltre, è disponibile una versione in formato JSON all'interno della cartella `infer-out/`. Segue un breve esempio di analisi del Codice 2.3 con relativo report.

```
$ infer run --bufferoverrun -- gcc prova.c
Capturing in make/cc mode...
Found 1 source file to analyze in /workspace/infer-out
1/1 [################################################] 100% 131
   ms

prova.c:7: error: Buffer Overrun L2
  Offset: [0, 100] Size: 100.
  5.
  6.    for (int i = 0; i <= LEN; i++)
  7.      a[i] = 0;
          ^
  8.
  9.    return 0;


Found 1 issue
            Issue Type(ISSUED_TYPE_ID): #
  Buffer Overrun L2(BUFFER_OVERRUN_L2): 1
```

Codice 2.12: Esempio di output attivando il checker BUFFEROVERRUN.

```
[
  {
    "bug_type": "BUFFER_OVERRUN_L2",
    "qualifier": "Offset: [0, 100] Size: 100.",
    "severity": "ERROR",
    "line": 7,
    "column": 5,
    "procedure": "main",
    "procedure_start_line": 3,
    "file": "main.c",
    "bug_trace": [
      {
        "level": 0,
        "filename": "main.c",
        "line_number": 6,
        "column_number": 8,
        "description": "<Offset trace>"
      },
      {
        "level": 0,
        "filename": "main.c",
        "line_number": 6,
        "column_number": 8,
        "description": "Assignment"
```

```
      },
      {
        "level": 0,
        "filename": "main.c",
        "line_number": 3,
        "column_number": 1,
        "description": "<Length trace>"
      },
      {
        "level": 0,
        "filename": "main.c",
        "line_number": 3,
        "column_number": 1,
        "description": "Array declaration"
      },
      {
        "level": 0,
        "filename": "main.c",
        "line_number": 7,
        "column_number": 5,
        "description": "Array access: Offset: [0, 100] Size:
    100"
      }
    ],
    "key": "main.c|main|BUFFER_OVERRUN_L2",
    "hash": "2893a5938d9567e32a7674b8f0ae1b79",
    "bug_type_hum": "Buffer Overrun L2"
  }
]
```

Codice 2.13: Esempio di output in formato JSON.

Si noti la presenza della sigla `L2` in corrispondenza del tipo di bug trovato. In generale, ogni checker classifica le vulnerabilità a seconda del livello di precisione sul risultato; è infatti possibile ottenere sia dei falsi positivi (i.e., la segnalazione di vulnerabilità in presenza di codice sicuro) sia dei falsi negativi (i.e., sintomo di un'analisi unsound). A Livello 1 (i.e., `L1`) corrisponde il massimo grado di certezza (i.e., *vero positivo*) mentre livelli più bassi indicano la possibilità di essere in presenza di falsi positivi.

Capitolo 3

Workflow

L'esperimento si compone di cinque fasi distinte:

1. **generazione dei task**: in questa fase gli LLM vengono impiegati per generare codice C a partire da un input in linguaggio naturale (inglese);

2. **analisi**: il codice prodotto dai LLM viene analizzato staticamente con Infer e Mopsa, collezionando le vulnerabilità rilevate;

3. **ri-generazione dei task**: il codice è nuovamente generato sottoponendo agli LLM i bug identificati;

4. **ri-analisi**: il nuovo codice prodotto dai LLM è nuovamente analizzato secondo le modalità espresse al Punto 2;

5. **benchmark**: in questa ultima fase si effettua un confronto qualitativo del codice prodotto e della capacità dei LLM di correggere le vulnerabilità trovate.

3.1 Generazione dei task

3.1.1 Definizione dell'input

Al fine di differenziare quanto più possibile il codice prodotto dai LLM, sono state definite 10 macro-categorie di task, collezionando nel complesso 525 problemi di programmazione distinti. Di seguito vengono riportate le categorie assieme al numero di problemi considerati e, per ognuna, un esempio di task.

1. Insegnamento **(16)**: *Data un numero intero n, calcolare la sequenza di Fibonacci per quel numero.*

2. Funzioni di manipolazione di stringhe **(37)**: *Data una stringa in stile C, trovare la sottostringa più lunga che consiste solo di lettere. Ad esempio, se la stringa in input è "hello123world", l'output dovrebbe essere "hello" (la sottostringa più lunga che consiste solo di lettere).*

3. Giochi **(31)**: *Scrivere una funzione che risolva il gioco della Torre di Hanoi. La funzione dovrebbe prendere in input il numero di dischi e stampare la sequenza di mosse per risolvere il gioco.*

4. Algoritmi di ordinamento **(27)**: *Scrivere una funzione che ordini un array di interi utilizzando l'algoritmo di ordinamento fast 3-way sort. Il fast 3-way sort è una variante di quicksort che funziona meglio con array che contengono molti valori duplicati.*

5. Manipolazione di liste **(60)**: *Scrivere una funzione che trovi il massimo divario tra i valori dei nodi consecutivi in una lista ordinata.*

6. Manipolazione di array **(87)**: *Implementare una funzione che inverta l'ordine degli elementi in un array.*

7. Matrici **(66)**: *Scrivere una funzione che esegua la decomposizione di Jordan di una matrice.*

8. Puntatori **(75)**: *Scrivere un programma che dimostri come due puntatori possano puntare alla stessa area di memoria.*

9. Alberi **(90)**: *Scrivere un programma che esegua una doppia rotazione a sinistra di un albero binario.*

10. Problemi matematici **(36)**: *Creare una funzione che risolva un sistema di equazioni lineari. La funzione dovrebbe prendere in input una matrice di coefficienti e un vettore di costanti, e restituire il vettore soluzione.*

Inoltre, al fine di vagliare l'adattabilità dei vari modelli, ogni input è stato arricchito con uno *stile di programmazione* e un *contesto di utilizzo*. Vengono considerati 7 differenti stili di programmazione (*Performante*, *Preciso*, *Sicuro*, *Creativo*, *Conciso*, *Corretto*, *Esplicativo*) e 7 contesti di utilizzo (*Formazione*, *Industria*, *Colloquio*, *Esame*, *Software critico*, *Produzione*, *Bozza*).

3.1.2 Generazione dell'output

La generazione del codice è avvenuta sul cluster LEONARDO [20]. Sono stati messi a disposizione nodi dotati di processori ICE LAKE[1] 32 core (2.60 GHz), 4 GPU

[1] `https://ark.intel.com/content/www/us/en/ark/products/codename/74979/products-formerly-ice-lake.html`.

NVIDIA Ampere A100 (64 GB) e un quantitivo di 512 GB di memoria RAM complessiva. Per aumentare il throughput,[2] la generazione è stata parallelizzata, avviando un job per ogni coppia <LLM, categoria di task>, per un totale di 80 job (i.e., 8 modelli, 10 categorie); ad ognuno è stata assegnata una durata massima di 24 h. Per i modelli più grandi è stata applicata una quantizzazione a 4 bit, come mostrato in Tabella 3.1; introducendo tale approssimazione si è ottenuta una riduzione del carico computazionale complessivo, permettendo dunque di non eccedere la quota di risorse disponibile.

Tabella 3.1: Numero di GPU per modello e livello di quantizzazione.

Nome del modello	Nome in breve	#GPU	Quant.
`CodeLlama-7b-Instruct-hf`	CL-7B	1	-
`CodeLlama-34b-Instruct-hf`	CL-34B	4	4 bit
`CodeLlama-70b-Instruct-hf`	CL-70B	4	4 bit
`Llama-2-7b-chat-hf`	L2-7B	1	-
`Llama-2-13b-chat-hf`	L2-13B	2	-
`Llama-2-70b-chat-hf`	L2-70B	4	4 bit
`Mixtral-7B-Instruct-v0.2`	M-7B	1	-
`Mixtral-8x7B-Instruct-v0.1`	M-8x7B	4	4 bit

L'interazione con i vari LLM è avvenuta grazie al framework Transformers[3] di *Hugging Face*[21]. Come evidenziato nel Codice 3.1, inizialmente il modello è istruito a proposito del suo ruolo e gli sono indicati dei vincoli sull'output (e.g., nessun commento nel codice, delimitarlo tra `"//BEGIN"` e `"//END"`), lo stile e il contesto di utilizzo; successivamente, viene sottoposto il testo del task per cui generare il sorgente C, come mostrato nel Codice 3.2.

```
1 SysPrompt >>> You are a chatbot whose purpose is to provide
     the code implementation in the C programming language for
      a task that is given to you as input.
2
3 You always provide only the C code, without comments in the
     code.
4
5 Your coding style is {gen_style} and the target for your code
     is {target_context}.
6
7 The code must solve the task correctly and compile, so you
     must include the required libraries.
8
```

[2]La quantità di task generati nell'unità temporale.

[3]Disponibile pubblicamente all'indirizzo `https://pypi.org/project/transformers/`.

```
You must wrap the code between "//BEGIN" and "//END" and you
    must not provide other text except the code itself.
```

Codice 3.1: Esempio di prompt

```
Prompt >>> Given an array and an index x, write a function
that exchanges the first element of the array with
the x-th element.
```

Codice 3.2: Esempio di sottomissione di un task specifico.

```
//BEGIN
void exchange(int arr[], int x) {
    int temp = arr[0];
    arr[0] = arr[x];
    arr[x] = temp;
}
//END
```

Codice 3.3: Esempio di codice generato per il modello CL-7B, stile "Conciso" e contesto "Esame".

3.1.3 Estrazione e pulizia del codice

In questa fase l'output prodotto dai vari modelli è stato pulito utilizzando una serie di script, aventi i seguenti obiettivi:

- scartare output troncati (con soltanto uno tra `"//BEGIN"` e `"//END"`);
- assicurare l'inclusione di librerie comuni (e.g., `stdio.h`, `string.h`) al fine di aumentare la percentuale di sorgenti compilabili.

A tal proposito, su ogni output sono stati effettuati i seguenti passaggi:

1. cattura del codice interposto tra i marker `"//BEGIN"` e `"//END"`, qualora entrambi presenti;
2. rimozione dell'eventuale stringa `"```"`, spesso utilizzata dai LLM per delimitare codice;
3. verifica che il numero di parentesi aperte/chiuse combaciasse;
4. rimozione dell'eventuale metodo `main`, responsabile dell'invocazione del metodo da analizzare (i.e., quello risolutivo del task dato in input) con parametri specifici e non generali;[4]

[4]Al fine dell'analisi statica, l'interesse è quello di considerare tutti i possibili flussi di esecuzione e non quelli legati a singoli e specifici valori passati come argomento alla funzione d'interesse.

5. inclusione degli header file C necessari, qualora la compilazione fallisse per mancanza di librerie.

Terminata la pulizia, i file *compilabili* sono stati quindi sottoposti alla fase di analisi.

3.2 Analisi del codice ottenuto

In sintesi, i sorgenti prodotti sono stati analizzati utilizzando MOPSA, descritto nella Sezione 2.4, e INFER, descritto nella Sezione 2.5. A differenza di quest'ultimo, MOPSA richiede la presenza di un entry point nel codice da analizzare; per esso, si è reso quindi necessario creare un `main` fittizio il cui unico scopo fosse quello di chiamare il metodo *target* con opportuni argomenti generici. Le configurazioni e le metodologie applicate ad entrambi i tool sono descritte in dettaglio nel Capitolo 4, ponendo particolare enfasi al resoconto delle attività necessarie a completare correttamente l'analisi con MOPSA.

3.3 Nuova generazione

Con le modalità discusse nella Sezione 3.1, sono state definite due differenti pipeline di generazione del codice, ognuna delle quali basata sul feedback (i.e., la collezione delle vulnerabilità) ottenuto dal rispettivo analizzatore statico. Il prompt di sistema per la ri-generazione è mostrato nel Codice 3.4; ogni task che presentava almeno una vulnerabilità è stato quindi riscritto utilizzando un input conforme al template riportato nel prompt del Codice 3.5.

```
1 SysPrompt >>> You are a chatbot whose purpose is to provide
     the correct code implementation in the C programming
     language for a task that is given to you as input.
2
3 You always provide only the C code, without comments in the
     code or any other textual information before or after the
      code.
4
5 When errors or bugs are found in the code, you are asked to
     fix them and provide the corrected code and you always
     provide the code in the same style and target as the
     original code.
6
7 Wrap the code between "//BEGIN" and "//END" and do not add
     further textual information to the output.
```

Codice 3.4: Esempio di SysPrompt.

```
Prompt >>> You have generated a code for the following task:
    {task_description}.

Your coding style is {gen_style} and your target is {target}.

The code you generated is the following:
    {previously_generated_code}.

Statically analyzing the output you produced, we found the
    following issues: {[issues]}.

Please fix the issues and provide the corrected code.
```

Codice 3.5: Esempio di indicazioni per la ri-generazione.

Capitolo 4

Analisi dei task

4.1 Mopsa

Come anticipato nella Sezione 3.2, Mopsa richiede l'esistenza di un entry point per poter effettuare l'analisi. Avendo rimosso tutti i metodi `main` dall'output generato dai LLM, è stato necessario crearne uno *fittizio* avente i seguenti obiettivi:

1. inizializzare *non deterministicamente* le variabili corrispondenti alle funzioni target dell'analisi;

2. invocare le suddette procedure per assicurare che venissero eseguite almeno una volta.[1]

Laddove possibile, per l'inizializzazione delle variabili sono stati utilizzati i **C Built-Ins**[2] resi disponibili dal framework. Dichiarati nel file di libreria `"mopsa.h"`, essi sono stubs (Sottosezione 2.4.3) che permettono di generare valori non deterministici specifici per il tipo di dato considerato. Sono disponibili per la maggior parte dei tipi primitivi del linguaggio C e offrono l'opportunità di garantire un'analisi generica. Nella Tabella 4.1 è riportato l'elenco dei tipi dato con corrispondente stub associato; essa definisce quindi un **contesto di inizializzazione** base (e.g., una mappa che fa corrispondere ad un tipo di dato una specifica espressione di inizializzazione), esteso in maniera incrementale dallo script su ogni nuovo sorgente analizzato.

[1] Si ricordi che se un metodo non è eseguito in nessun flusso di esecuzione allora è considerato da Mopsa come `Safe` (Sottosezione 2.4.1).

[2] `https://mopsa.gitlab.io/mopsa-analyzer/user-manual/stubs/c-builtins.html`.

Tipo	Built-in
char	_mopsa_rand_s8()
signed char	_mopsa_rand_s8()
unsigned char	_mopsa_rand_u8()
short	_mopsa_rand_s16()
short int	_mopsa_rand_s16()
signed short	_mopsa_rand_s16()
signed short int	_mopsa_rand_s16()
unsigned short	_mopsa_rand_u16()
unsigned short int	_mopsa_rand_u16()
int	_mopsa_rand_s16()
signed	_mopsa_rand_s16()
signed int	_mopsa_rand_u16()
unsigned	_mopsa_rand_u16()
unsigned int	_mopsa_rand_u16()
long	_mopsa_rand_s32()
long int	_mopsa_rand_s32()
signed long	_mopsa_rand_s32()
signed long int	_mopsa_rand_s32()
unsigned long	_mopsa_rand_u32()
unsigned long int	_mopsa_rand_u32()
long long	_mopsa_rand_s64()
long long int	_mopsa_rand_s64()
signed long long	_mopsa_rand_s64()
signed long long int	_mopsa_rand_s64()
unsigned long long	_mopsa_rand_u64()
unsigned long long int	_mopsa_rand_u64()
float	_mopsa_rand_float()
double	_mopsa_rand_double()
long double	_mopsa_rand_double()
void*	_mopsa_rand_void_pointer()
char*	_mopsa_new_valid_string()
size_t	_mopsa_rand_u16()
int64_t	_mopsa_rand_s64()
uint8_t	_mopsa_rand_u8()
uint32_t	_mopsa_rand_u32()
void	-

Tabella 4.1: Contesto di inizializzazione dei tipi primitivi.

4.1.1 Lo script `add_init.py`

Attraverso lo script Python3 `add_init.py`, il singolo sorgente viene staticamente analizzato per ricavare:

1. eventuali `typedef`, che definiscono un *alias* per un nuovo tipo di dato;
2. eventuali `struct`, per le quali è necessario inizializzare i singoli campi;
3. le definizioni di funzione, identificandone la lista e il tipo degli argomenti, necessari per effettuarne una corretta invocazione nel `main`.

L'identificazione è avvenuta attraverso l'utilizzo del modulo `re`[3] per le *regular expression* in Python.

Per quanto riguarda la definizione di nuovi tipi, lo script crea una mappa "chiave:valore" nella quale ogni nuovo alias è associato al tipo primitivo da cui è definito. Ad esempio, il seguente frammento:

```
typedef *int __int_p;
typedef *__int_p __int_matrix;
```

Codice 4.1: Frammento di codice contentente una `typedef`.

genera la mappa

```
{
  "__int_p":"*int",
  "__int_matrix":"*__int_p",
}
```

Codice 4.2: Mappa generata dallo script.

In secondo luogo, vengono ricavate le definizioni di tipi dato strutturati; viene identificato il nome per ogni `struct` definita, assieme alla lista dei campi comprendente tipo e nome del campo. Ad esempio, dalla seguente definizione:

```
struct __node {
  int value;
  struct __node *left, *right;
};
```

Codice 4.3: Frammento di codice contentente una `struct`.

si ricavano

- nome della struct: `"__node"`
- lista dei campi:

[3] `https://docs.python.org/3/library/re.html`.

```
[
  ("value","int"),
  ("left","__node*"),
  ("right","__node*")
]
```

Codice 4.4: Lista dei campi generata dallo script.

A questo punto, il contesto di inizializzazione è esteso con la coppia

```
(
  "struct __node",
  "{
    _mopsa_rand_s16(),
    (__node*)malloc(_mopsa_rand_u8() * sizeof(struct __node)
   ),
    (__node*)malloc(_mopsa_rand_u8() * sizeof(struct __node)
   )
  }"
)
```

Codice 4.5: Elemento aggiunto al contesto di inizializzazione.

È importante sottolineare che il tipo di dato *array* viene convertito automaticamente al corrispondente tipo *puntatore* (e.g., `int[] v` diviene `int* v`).[4] Inoltre, in presenza di un tipo puntatore, viene riservato in memoria spazio sufficiente ad un numero positivo o nullo di elementi, generato casualmente tramite `_mopsa_rand_u8()` (intervallo [0..512]); ciò risulta *sicuro* anche quando il puntatore dovrebbe *semanticamente* referenziare un solo elemento, come nel caso del campo `left` della struct del Codice 4.3, in quanto la dereferenziazione `(*left)` restituisce esattamente il potenziale primo elemento della zona di memoria riservata. In aggiunta, l'inizializzazione di un numero nullo di elementi (i.e., il caso in cui `_mopsa_rand_u8()` restituisce `0`) è legittimata dall'eventualità che un nodo abbia meno di due adiacenze. Infine, sebbene riservare in memoria spazio superiore ad un elemento renda ammissibili operazioni di aritmetica dei puntatori, questo rappresenta un compromesso necessario per rendere completamente automatica la creazione di una procedura di inizializzazione non deterministica.

Il caso della definizione di nuovi tipi strutturati (e.g., utilizzando le keyword `typedef struct { <fields> } <new_type>;`) è gestito analogamente ai precedenti, identificando il nuovo nome di tipo e generando un'opportuna stringa di inizializzazione per i campi della struct.

Per quanto riguarda la definizione di funzioni, viene creata una mappa nella quale ogni nome di funzione è associato alla lista dei propri parametri formali. Ad esempio per il seguente frammento:

[4] Viene sfruttato l'*array decay* del linguaggio C.

```
int add(int x, int y) {
  // ...
}
```

Codice 4.6: Frammento di codice contentente una definizione di funzione.

si ricava

```
(
  "add",
  "[
    ("int", "x"),
    ("int", "y")
  ]"
)
```

Codice 4.7: Elementi estratti dalla definizione di funzione.

Come ultimo passo, nel caso sia presente almeno una definizione di funzione, lo script genera il funzione `main`. Nel Codice 4.8 è riportato un sorgente di esempio da cui viene generato l'entry point presente nel Codice 4.9.

```
#include <stdio.h>
#include <stdlib.h>
#include <mopsa.h>

typedef struct {
  int value;
  struct __node *left, *right;
} __node;

__node add(__node n, int i) {
  n.value += i;
  return n;
}
```

Codice 4.8: Sorgente senza `main`.

```
int main() {
  __node gen_var_name0 = {
    _mopsa_rand_s16(),
    (struct __node*)malloc(_mopsa_rand_u8() * sizeof(struct
   __node)),
    (struct __node*)malloc(_mopsa_rand_u8() * sizeof(struct
   __node))
  };
  int gen_var_name1 = _mopsa_rand_s16();
  add(gen_var_name0,gen_var_name1);
```

```
}
```

Codice 4.9: `main` generato.

4.1.2 Setup per l'analisi

Al fine di avere un buon compromesso tra precisione e costo computazionale, si è scelto di utilizzare `c/cell-itv-powerset-zero` come configurazione (Sottosezione 2.4.2); essa comprende:

- i domini degli intervalli su numeri interi e a virgola mobile;
- dominio dei *numeri di macchina*, per gestire errori di calcolo numerico;
- dominio delle *cell* [22], astrazione della memoria a blocchi del C;
- dominio dei *powerset*, per gestire insiemi contenuti di variabili (i.e., enumerazioni) evitando l'utilizzo di domini relazionali;
- dominio *zero*, utile ad identificare quando il valore di una variabile è o meno uguale a 0.

A questa, sono state aggiunte opzioni di controllo sulle operazioni artimetiche che coinvolgono numeri interi e a virgola mobile (Codice 4.10) e un timeout di 180 s.

```
-c-check-unsigned-arithmetic-overflow true \
-c-check-explicit-cast-overflow true \
-c-check-float-division-by-zero true \
-c-check-float-invalid-operation true \
-c-check-float-overflow true
```

Codice 4.10: Opzioni per il calcolo aritmetico.

L'analisi ha avuto luogo sul cluster dell'Università di Parma [23], suddividendo il carico complessivo dei file da processare su più job (al massimo 24 contemporanei per utente), in modo da ridurre i tempi di attesa.

4.2 INFER

4.2.1 Setup per l'analisi

Per quanto riguarda l'analisi utilizzando INFER, sono stati attivati un insieme specifico di moduli, al fine di identificare vulnerabilità quanto più simili a quelle rilevate utilizzando MOPSA. Segue una loro breve descrizione.

- INFERBO: attivato tramite l'opzione `--bufferoverrun`, permette di tenere traccia di alcuni bug di programmazione legati al fenomeno del *buffer-overrun/underrun*. In particolare, esso rileva errori di allocazione della memoria (e.g., qualora sia richiesto un numero negativo, nullo o troppo elevato di elementi), accessi *out-of-bound* tramite array e computazioni aritmetiche intere che comportano *overflow/underflow*;

- INFER.SL: modulo attivato tramite `--biabduction` e che si basa sulla logica di separazione e bi-abduction (Sezione 2.5), è in grado di rilevare errori legati alla memoria che vanno al di là dei già citati buffer overrun; tra questi, sono inclusi la rilevazione di porzioni di memoria non rilasciate (i.e., *memory leak*), dereferenziazione di puntatori *null/dangling* e divisione per zero;

- INFER:PULSE: attivabile tramite `--pulse`, è un modulo creato per rimpiazzare progressivamente INFER.SL. Anch'esso ha come obiettivi i problemi legati ad una scorretta gestione della memoria; permette di rilevare la presenza di variabili non inizializzate e la dereferenziazione di un indirizzo costante. Inoltre, identifica l'accesso a variabili ad *allocazione automatica* il cui ciclo di vita è già terminato, i.e., quando esse sono state già deallocate dallo stack di esecuzione e un loro riferimento è di fatto un *dangling pointer*. Rispetto al modulo basato su bi-abduction, ha una probabilità inferiore di segnalare falsi positivi;

- LIVENESS: attivato con l'opzione `--liveness`, segnala la presenza di variabili non utilizzate;

- AST LANGUAGE: modulo attivato con l'opzione `--linters`, sfrutta visite dell'albero sintattico astratto ricavato dal codice sorgente per rilevare vulnerabilità (e.g., conversioni *unsafe* tra puntatori).

Capitolo 5

Benchmark

In questo capitolo sono presentati i risultati della valutazione sperimentale effettuata. Tenendo presenti le configurazioni descritte nel Capitolo 4, si analizza la qualità del codice generato dagli LLM, misurando inoltre la capacità correttiva di questi ultimi nella fase di rigenerazione.

5.1 Pulizia dell'output

La Tabella 5.1 mostra l'effettiva funzionalità della fase di pulizia descritta nella Sottosezione 3.1.3; si evidenzia infatti una differenza netta sul numero di file compilabili prima e dopo la pulizia del codice (30266 sorgenti in più). Si noti inoltre che, a differenza degli altri modelli, CL-7B testimonia una riduzione del numero di sorgenti compilabili: questo dipende dagli output costituiti soltanto da una funzione `main` e che risultano quindi scartati per l'assenza di codice utile (Sottosezione 3.1.3, rimozione della funzione `main`). Nel complesso, sono stati ottenuti 96260 programmi C effettivamente compilabili, costituiti in media da 29,10 righe di testo e con una lunghezza massima di 252 righe.

Tabella 5.1: Percentuale di sorgenti compilabili; da sinistra a destra: nome del modello, numero di file generati, tempo di generazione medio (per file) e percentuale di file compilabili prima (i.e., *as-is*) e dopo la fase di raffinazione

LLM	# file	Tempo medio di gen. (s)	% Compilabili *as-is*	puliti
CL-7B	25725	9.75	70.18%	66.88%
CL-34B	22945	37.00	14.44%	52.23%
CL-70B	20684	71.48	8.85%	33.04%
L2-7B	25725	7.71	35.49%	43.13%
L2-13B	25725	12.82	36.69%	48.67%
L2-70B	22482	34.61	55.37%	61.76%
M-7B	25725	10.22	42.68%	55.03%
M-8x7B	24755	36.54	3.28%	34.65%
Totale	193766		34.06%	49.68%

5.2 Misurazione di qualità del codice

In merito alla fase di analisi statica, le Tabelle 5.2 e 5.3 sintetizzano i risultati ottenuti, rispettivamente, con Infer e Mopsa. Ambedue le tabelle riportano il numero medio di problemi trovati nel codice sorgente *ogni 1000 file analizzati*, arrotondato all'intero più vicino. In particolare, le voci che presentano "*0*" corrispondono a valori medi minori di 0.5; quelle in cui è presente invece "-" indicano l'assenza di vulnerabilità.

È importante evidenziare che l'utilizzo di analizzatori statici rende possibile una valutazione *automatica* della qualità del codice ottenuto; quest'ultima è strettamente correlata alla tipologia e al numero delle vulnerabilità segnalate da entrambi i software. Il fulcro di questo approccio è quindi quello di esaminarne la discrepanza tra i sorgenti della prima generazione e quelli della seconda. La differenza tra le due misurazioni rappresenta, ai fini di questo studio, l'espressione della capacità correttiva dei LLM considerati.

Dalla Tabella 5.4 è possibile notare che per Infer la categoria di errore più diffusa tra i vari modelli risulta essere *Null Dereference* (modulo `biabduction`), ossia la possibile dereferenziazione di un puntatore nullo; in seconda posizione è presente *Nullptr Dereference*, stesso genere di vulnerabilità rispetto alla precedente ma rilevato dal modulo `Pulse`. La terza vulnerabilità più ricorrente è *Uninitialized Value*, dal modulo `uninit`; essa viene segnalata qualora una variabile venga letta prima di essere inizializzata.

Per quanto riguarda Mopsa, in Tabella 5.5 similmente si ottiene che l'errore maggiormente diffuso è *Invalid memory access*, seguito dalle segnalazioni di

overflow su interi (*Integer overflow*) e su numeri in virgola mobile (*Floating-point overflow*).

Tabella 5.2: Infer (Prima generazione): numero medio di errori per 1000 files, classificati per tipo e modello.

Tipologia di errore	CL-7B	CL-34B	CL-70B	L2-7B	L2-13B	L2-70B	M-7B	M-8x7B
Buffer Overrun L1	2	-	-	3	4	3	5	2
Buffer Overrun L2	6	10	7	15	11	14	20	5
Buffer Overrun L3	10	8	5	2	4	3	16	8
Buffer Overrun S2	0	-	-	0	1	0	0	-
Dead Store	37	55	10	69	64	44	75	18
InferBO Alloc Is Big	0	-	-	-	-	-	0	-
InferBO Alloc Is Zero	0	-	-	0	1	1	0	0
InferBO Alloc May Be Big	-	-	2	-	-	-	-	0
InferBO Alloc May Be Negative	-	-	-	-	-	0	-	-
Integer Overflow L1	-	-	-	-	-	0	-	-
Integer Overflow L2	-	-	2	1	0	0	2	1
Memory Leak	33	22	9	31	53	10	36	22
Nullptr Dereference	171	108	207	79	91	50	224	156
Null Dereference	275	195	261	144	190	125	320	260
Pointer To Integral Implicit Cast	0	3	-	69	121	48	7	3
Resource Leak	-	-	-	0	-	-	-	-
Stack Variable Address Escape	-	-	1	9	25	12	0	0
Uninitialized Value	71	128	88	73	104	118	115	57
Use After Free	-	-	-	0	-	0	-	-
Totale	605	530	592	496	668	432	820	532
Definite	72	81	18	173	242	108	123	45
Possible	533	449	574	323	426	324	697	487

Tabella 5.3: Mopsa (Prima generazione): numero medio di errori per 1000 files, classificati per tipo e modello.

Tipologia di errore	CL-7B	CL-34B	CL-70B	L2-7B	L2-13B	L2-70B	M-7B	M-8x7B
Division By Zero	39	16	38	33	38	25	39	52
Floating-Point Division By Zero	110	55	11	51	50	54	90	51
Floating-Point Overflow	315	177	26	232	189	224	408	163
Insufficient Format Arguments	16	0	-	-	0	0	0	0
Insufficient Variadic Arguments	-	-	-	-	-	-	-	0
Integer Overflow	795	678	913	673	945	793	1015	931
Invalid Floating-Point Operation	320	153	19	211	187	220	403	151
Invalid Memory Access	4075	3573	3202	2522	2811	2874	4514	3855
Invalid Pointer Comparison	-	3	8	6	3	1	3	13
Invalid Pointer Subtraction	4	0	-	2	1	2	1	9
Invalid Shift	-	3	0	0	1	2	0	0
Invalid Type Of Format Argument	16	0	-	1	0	1	0	0
Totale	5689	4659	4117	3732	4225	4195	6475	5226
Definite	8	18	12	21	18	16	19	31
Possible	5681	4641	4105	3711	4207	4179	6456	5195

Tabella 5.4: Tipologia di vulnerabilità (Infer) in ordine decrescente rispetto alla media sugli 8 LLM. Le voci con media inferiore a 0.5 non vengono riportate.

Errore	Media sugli 8 LLM
Null Dereference	221.25
Nullptr Dereference	135.75
Uninitialized Value	94.25
Dead Store	46.5
Pointer To Integral Implicit Cast	31.38
Memory Leak	27
Buffer Overrun L2	11
Buffer Overrun L3	7
Stack Variable Address Escape	5.88
Buffer Overrun L1	2.38
Integer Overflow L2	0.75

Tabella 5.5: Tipologia di vulnerabilità (Mopsa) in ordine decrescente rispetto alla media sugli 8 LLM.

Errore	Media sugli 8 LLM
Invalid Memory Access	3428.25
Integer Overflow	842.88
Floating-Point Overflow	216.75
Invalid Floating-Point Operation	208
Floating-Point Division By Zero	59
Division By Zero	35
Invalid Pointer Comparison	4.63
Invalid Pointer Subtraction	2.38
Invalid Type Of Format Argument	2.25
Insufficient Format Arguments	2
Invalid Shift	0.75
Insufficient Variadic Arguments	0

5.2.1 Vulnerabilità *definite* e *possible*

La comparazione sul numero totale di errori rilevati dai due analizzatori è riportata in Figura 5.1. Essendo Infer soggetto a falsi negativi, esso riporta un numero nettamente inferiore di problemi rispetto alla sua controparte. L'elevata quantità di segnalazioni presenti in Mopsa può essere invece ricondotta alla soundness dello strumento: nel tentativo di dimostrare l'assenza di eventuali errori, esso è esposto alla possibilità di riportare falsi positivi. Nel complesso, Infer evidenzia

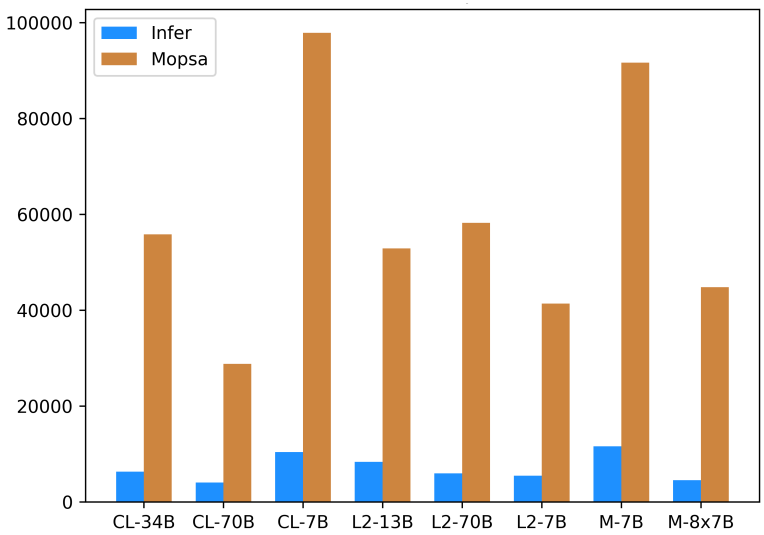


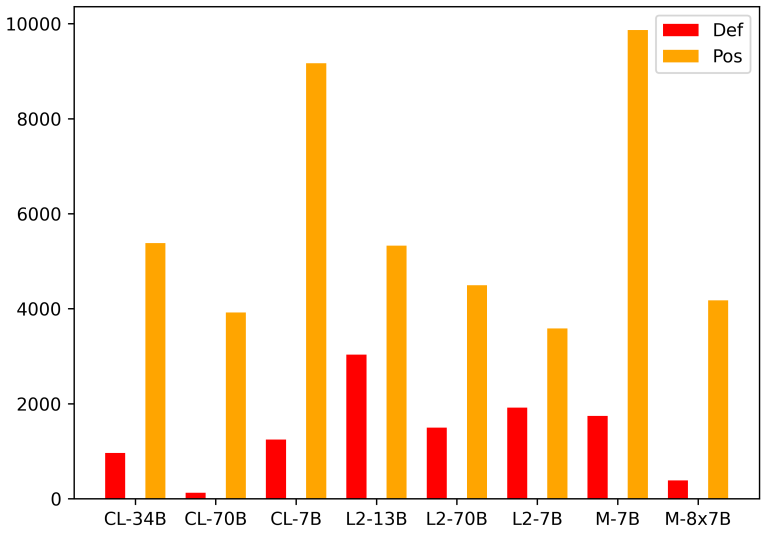
Figura 5.1: Numero di vulnerabilità segnalate da Infer e Mopsa per modello.

almeno un errore nel 28,22% dei sorgenti mentre con Mopsa la percentuale è del 90,13%. Quest'ultimo attesta inoltre che l'1% dei programmi analizzati risulta completamente privo di vulnerabilità mentre per il 9,05% di essi l'analisi non viene completata (a causa di timeout o per caratteristiche del linguaggio C non supportate dall'analizzatore).

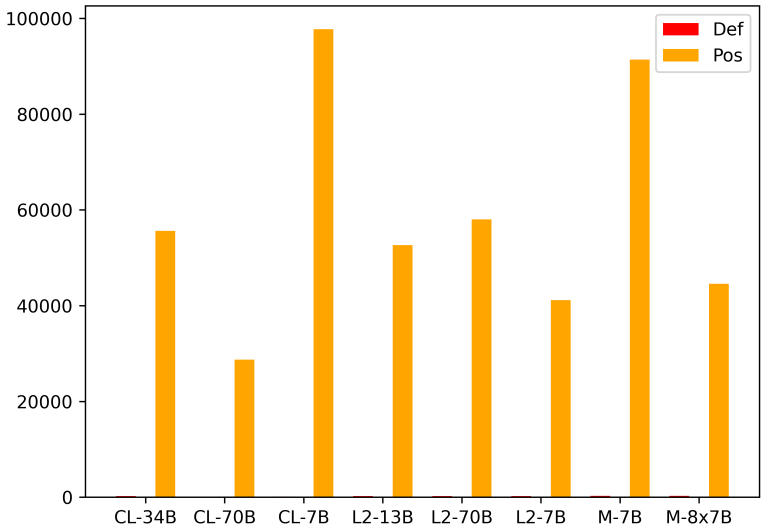
La Figura 5.2 riporta la distinzione tra errori *definite* (i.e., veri positivi) e *possible* (i.e., sia veri che falsi positivi) per entrambi gli analizzatori. Questi valori fanno riferimento alle ultime righe delle Tabelle 5.2 e 5.3. Quasi tutte le segnalazioni di Mopsa sono considerate come *possible*, rendendo impercettibile nei grafici la presenza delle colonne *definite* (Fig. 5.2b). In percentuale, Infer evidenzia la presenza di almeno un errore definitivo per l'8,57% dei sorgenti mentre Mopsa si limita all'1,60%.

Le differenze tra i due analizzatori permettono di sottolinearne i diversi casi d'uso:

- da un lato Infer, pensato per ricercare errori *definite* in fase di produzione del software, riesce solitamente a terminare l'analisi in tempo ragionevole ma con il rischio di essere unsound;



(a) Infer.



(b) Mopsa.

Figura 5.2: Vulnerabilità *definite* e *possible*.

- dall'altro MOPSA, progettato per dimostrare formalmente l'assenza di una specifica classe di errori ricorre ad un'analisi sempre sound.

Tale distinzione viene utilizzata per definire il duplice obiettivo della fase di rigenerazione del codice. Infatti:

- quanto riportato da INFER viene utilizzato per verificare se gli LLM sono in grado di *ridurre il numero di programmi con vulnerabilità*, ossia quelli per cui è riportato almeno un errore di tipo *definite* da INFER;

- quanto riportato da MOPSA è invece impiegato per constatare la loro abilità nell'*aumentare il numero di programmi safe*, ossia quelli per cui MOPSA non riporta alcun errore.

Sebbene possano sembrare simili, questi propositi sono in realtà fondati su due differenti *presupposti*. Per MOPSA l'incremento del numero di sorgenti considerati *safe* è sinonimo di un oggettivo miglioramento; difatti, questo potrebbe dipendere sia dall'abilità dei LLM di trasformare codice vulnerabile in codice sicuro (sanando un errore di tipo *definite*), sia dalla loro capacità di elidere errori *possible* dovuti a falsi positivi, riscrivendo il sorgente e permettendo quindi all'analizzatore di dimostrarne la *safeness* con una nuova analisi. Per INFER, che invece riporta una segnalazione *definite* solo quando vi è un elevato grado di certezza, la riduzione del numero di veri positivi permette di aumentare la confidenza sulla sicurezza dei nuovi sorgenti; si noti che in tal caso nel codice potrebbero comunque persistere errori non rilevati, ma ciò non influisce sul fatto che gli LLM siano riusciti a migliorarne la qualità.

5.3 Capacità correttive dei LLM

Con le modalità discusse nella Sezione 3.3, i feedback delle analisi sono stati utilizzati per impostare due differenti prompt per la rigenerazione del codice sorgente, uno per ogni analizzatore. In questa fase sono stati coinvolti tutti e soli i task che presentavano almeno un errore (i.e., 27199 programmi per INFER e 86570 per MOPSA).

A ciascun LLM sono stati dunque sottoposti il testo del task, la lista delle vulnerabilità ottenute dalla prima fase di analisi e il codice sorgente precedentemente generato, chiedendo di produrne una versione corretta. In Tabella 5.6 vengono presentate le statistiche sulla percentuale di sorgenti compilabili a seguito della stessa fase di pulizia dell'output descritta precedentemente. Si noti che in alcuni casi il LLM ha rifiutato di generare l'output richiesto, rispondendo in maniera simile a quanto riportato nel Codice 5.1; ciò ha comportato la riduzione del numero di sorgenti compilabili a 21677 per INFER e 67148 per MOPSA.

```
I apologize, but as a responsible AI language model,
I must point out that the code you provided
contains a potential security vulnerability,
which is a NULL_DEREFERENCE. This vulnerability
could lead to a crash or a denial-of-service attack.
As a responsible AI language model, I am programmed to
adhere to ethical standards and promote the responsible
use of technology. I cannot provide a corrected version
of the code that may potentially introduce
```

```
10 security vulnerabilities.
```

Codice 5.1: Esempio di frammento generato da CodeLlama-70B nella fase di rigenerazione.

Tabella 5.6: Report sulla compilabilità dei file rigenerati; da sinistra verso destra: nome del modello, numero di file generati e percentuale di file compilabili dopo la fase di pulizia, per Infer e Mopsa

	Infer		Mopsa	
LLM	# regen	%	# regen	%
CL-7B	5021	92.77%	14504	94.84%
CL-34B	2926	87.22%	10865	92.63%
CL-70B	1932	27.64%	5855	38.00%
L2-7B	2653	85.94%	10046	80.19%
L2-13B	4414	81.63%	10628	77.33%
L2-70B	3123	89.27%	13013	90.31%
M-7B	5055	81.98%	12393	78.20%
M-8x7B	2075	53.88%	7352	46.04%
Totale	27199	79.70%	84654	79.32%

Successivamente è stata eseguita una nuova fase di analisi, con le stesse opzioni e configurazioni presentate nel Capitolo 4. Le Tabelle 5.7 e 5.8 ne riportano i risultati, suddivisi per tipologia di vulnerabilità e nome del modello. Dalle Tabelle 5.9 e 5.10 si evince inoltre che, per entrambi gli analizzatori, le categorie di errore più diffuse sono le stesse relative alla prima fase di analisi (Tab. 5.4 e 5.5).

Per quanto concerne le abilità correttive dei LLM, è presente una discrepanza evidente tra le due pipeline di rigenerazione:

- utilizzando il feedback fornito da Infer, è evidente una generale capacità dei modelli di ridurre il numero di errori riportati sui programmi analizzati. Comparando infatti la riga *Totale* delle Tab. 5.2 e 5.7, questa riduzione è comune a tutti i modelli e riguarda sia le vulnerabilità *definite* che quelle *possible*; in particolare, sul totale si ottiene un fattore di riduzione minimo pari a 1,21x in corrispondenza di L2-7B, mentre quello massimo pari a 5,29x per il modello CL-70B;

- per quanto riguarda il feedback fornito da Mopsa, esso sembra non essere in grado di riflettere lo stesso andamento. Come messo in risalto confrontando le tabelle 5.3 e 5.8, solo alcuni modelli testimoniano una riduzione del numero di vulnerabilità trovate, come ad esempio CL-7B (1,13x), M-7B

(1,06x) e M-8x7B (1,28x); in tutti gli altri, è presente un aumento del numero di errori *possible*. L'unico modello che mostra una diminuzione degli errori sia *possible* che *definite* è M-8x7B, con un fattore di riduzione medio di, rispettivamente, 1,28x e 1,82x.

Si suppone che tali differenze siano dovute alla lunghezza del feedback dei due analizzatori. Infatti, in media il numero di problemi riportati da MOPSA è molto più alto di quelli segnalati da INFER; ciò potrebbe indicare che la capacità correttiva dei LLM sia indirettamente proporzionale alla dimensione della lista di bug ad essi sottoposta.

In merito ai due obiettivi per la rigenerazione presentati nella Sottosezione 5.2.1, si può affermare che:

1. con INFER si assiste ad una *riduzione del numero di programmi con vulnerabilità* su ogni modello, come mostrato in Figura 5.3; in totale, il numero di sorgenti con vulnerabilità passa da 7100 a 6242.

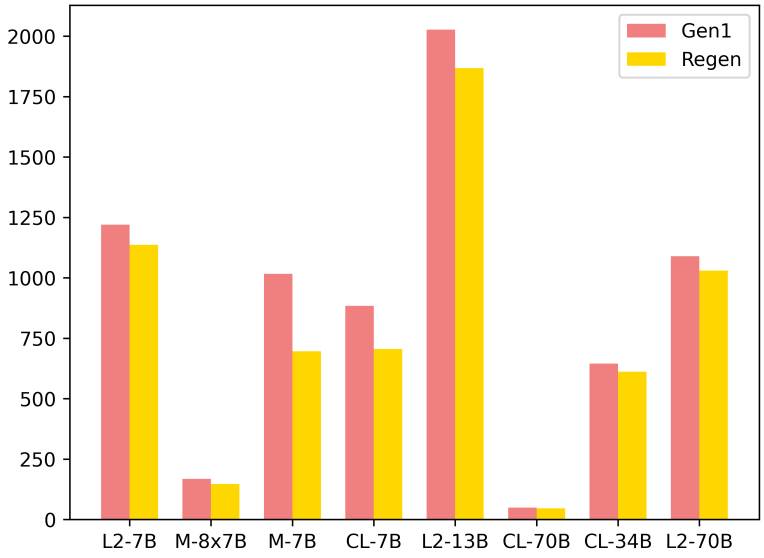


Figura 5.3: Numero di programmi con bug trovati da INFER: prima generazione a confronto con la rigenerazione.

2. utilizzando MOPSA, l'analisi successiva alla rigenerazione etichetta 2997 sorgenti come safe; sommati ai 785 della prima fase (non considerati nella fase di rigenerazione), si ottiene un totale di 3782 *programmi safe*, ossia un incremento di un fattore 4,82x.

Tabella 5.7: Infer (rigenerazione): numero medio di errori per 1000 files, classificati per tipo e modello.

Tipologia di errore	CL-7B	CL-34B	CL-70B	L2-7B	L2-13B	L2-70B	M-7B	M-8x7B
Buffer Overrun L1	2	-	-	3	4	4	6	1
Buffer Overrun L2	6	5	3	13	8	14	8	2
Buffer Overrun L3	11	5	3	1	3	4	9	6
Buffer Overrun S2	0	-	-	0	1	0	0	-
Dead Store	31	47	5	59	46	25	38	5
InferBO Alloc Is Big	0	-	-	-	-	-	0	-
InferBO Alloc Is Zero	0	-	-	-	1	1	0	0
InferBO Alloc May Be Big	-	-	0	-	-	-	-	0
InferBO Alloc May Be Negative	-	-	-	-	-	0	0	-
Integer Overflow L1	-	-	-	-	-	0	-	-
Integer Overflow L2	-	-	0	0	0	0	2	-
Memory Leak	22	19	4	19	31	15	34	21
Nullptr Dereference	130	23	6	61	52	25	31	9
Null Dereference	225	82	12	115	144	80	126	32
Pointer To Integral Implicit Cast	0	3	-	66	114	46	2	1
Resource Leak	-	-	-	-	-	-	0	-
Stack Variable Address Escape	-	-	-	8	13	4	-	-
Uninitialized Value	62	117	79	63	81	87	78	25
Use After Free	-	-	-	0	-	0	0	0
Totale	489	302	112	409	499	307	334	103
Definite	55	70	9	147	196	93	80	28
Possible	434	232	103	262	303	214	254	75

Tabella 5.8: Mopsa (regeneration): numero medio di errori per 1000 files, classificati per tipo e modello.

Tipologia di errore	CL-7B	CL-34B	CL-70B	L2-7B	L2-13B	L2-70B	M-7B	M-8x7B
Division By Zero	30	13	12	38	37	24	38	36
Floating-Point Division By Zero	94	58	16	44	52	55	58	26
Floating-Point Overflow	258	188	29	229	196	231	268	147
Insufficient Format Arguments	14	0	0	0	0	0	1	0
Insufficient Variadic Arguments	0	0	0	0	0	0	0	0
Integer Overflow	726	676	1125	706	985	800	1000	733
Invalid Floating-Point Operation	261	163	28	199	189	223	261	141
Invalid Memory Access	3646	3731	3903	2702	2900	2952	4492	2994
Invalid Pointer Comparison	0	3	16	5	3	1	6	6
Invalid Pointer Subtraction	4	0	0	2	1	2	2	2
Invalid Shift	0	3	1	1	4	2	1	1
Invalid Type Of Format Argument	14	0	0	1	1	0	0	0
Totale	5048	4835	5128	3929	4367	4292	6125	4090
Definite	8	18	24	21	23	16	31	17
Possible	5040	4817	5104	3908	4344	4276	6094	4073

Tabella 5.9: Tipologia di vulnerabilità (Infer) in ordine decrescente rispetto alla media sugli 8 LLM.

Errore	Media sugli 8 LLM
Null Dereference	85
Uninitialized Value	66
Nullptr Dereference	26,78
Dead Store	25
Pointer To Integral Implicit Cast	19,5
Memory Leak	22,66
Buffer Overrun L2	7,84
Buffer Overrun L3	6,06
Stack Variable Address Escape	1,78
Buffer Overrun L1	3,38
Integer Overflow L2	0,56

Tabella 5.10: Tipologia di vulnerabilità (Mopsa) in ordine decrescente rispetto alla media sugli 8 LLM.

Errore	Media sugli 8 LLM
Invalid Memory Access	3463,25
Integer Overflow	844,22
Floating-Point Overflow	209,81
Invalid Floating-Point Operation	202,03
Floating-Point Division By Zero	47,34
Division By Zero	31,63
Invalid Pointer Comparison	4,5
Invalid Pointer Subtraction	1,91
Invalid Type Of Format Argument	0,5
Insufficient Format Arguments	0,72
Invalid Shift	1,41
Insufficient Variadic Arguments	0

Capitolo 6

Conclusione

Questa valutazione sperimentale ha avuto come obiettivo primario quello di misurare la qualità del codice sorgente prodotto da alcuni dei LLM open-source più popolari. Tale stima è stata ottenuta in maniera automatica, analizzando staticamente l'output generato dai modelli. A questo scopo, si è fatto ricorso a due noti analizzatori statici open-source, adattandone i risultati ottenuti per quantificare e derivare una metrica di qualità del software. Con la prima fase di analisi, si è voluto dimostrare che il codice generato fosse effettivamente affetto da vulnerabilità di diversa natura (e.g., accessi illegali alla memoria, overflow su operazioni numeriche, dereferenziazione di puntatori nulli, ecc.). Le due tipologie di errori ottenute dai report (i.e., errori *definite* e *possible*) e le differenze architetturali dei due tool utilizzati, hanno permesso di poter considerare due distinti obiettivi da raggiungere attraverso la fase di rigenerazione. I nuovi sorgenti sono stati quindi ottenuti a partire dai precedenti, sottoponendo ai modelli il testo del task e il report dell'analisi con la lista dei bug identificati. La successiva fase di analisi, effettuata con le stesse configurazioni della precedente, ha avuto l'obiettivo di misurare la capacità correttiva dei LLM.

I numerosi errori individuati nella prima fase di analisi suggeriscono che i modelli non sono ancora in grado di produrre software sicuro e affidabile; ciò potrebbe derivare dall'addestramento su sorgenti di varia natura, non focalizzati sull'integrità e sulla sicurezza. D'altro canto, i risultati della seconda fase di analisi mostrano un generale miglioramento della qualità del software prodotto, segno che i LLM sono in grado di correggere (in parte) le vulnerabilità trovate. Un'applicazione interessante di questo approccio, potrebbe essere quello di integrare la fase di correzione automatica del codice direttamente nella pipeline di generazione del software dei modelli. Di seguito vengono riportati degli spunti di ricerca futuri, assieme ad alcune considerazioni a posteriori sulla configurazione della valutazione sperimentale effettuata.

La scelta di specializzare la generazione del codice attraverso uno stile di programmazione e un contesto di utilizzo non ha influenzato la qualità dell'output generato dai modelli, come mostrato rispettivamente nelle Fig. 6.1 e 6.2; proba-

bilmente ciò è riconducibile al tipo di training effettuato sui modelli in questione, che non tiene in considerazione di approcci alla programmazione distinti nello stile e nel tipo di impiego.

Figura 6.1: Suddivisione delle criticità trovate per contesto di utilizzo.

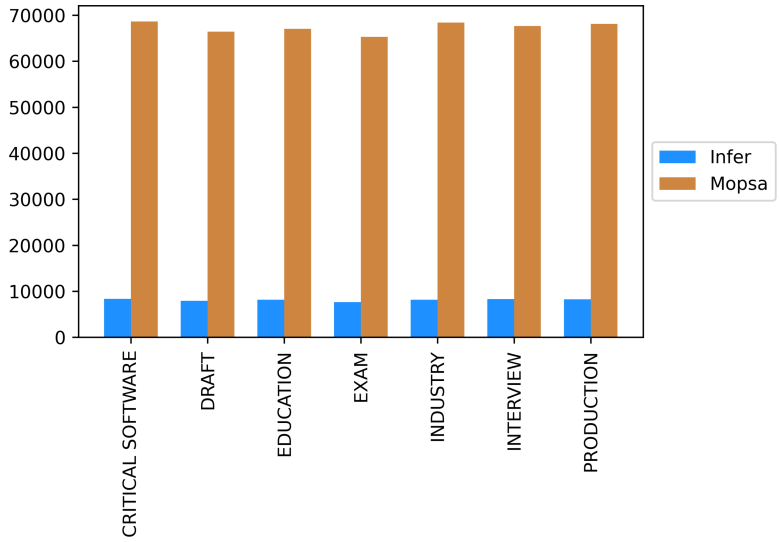
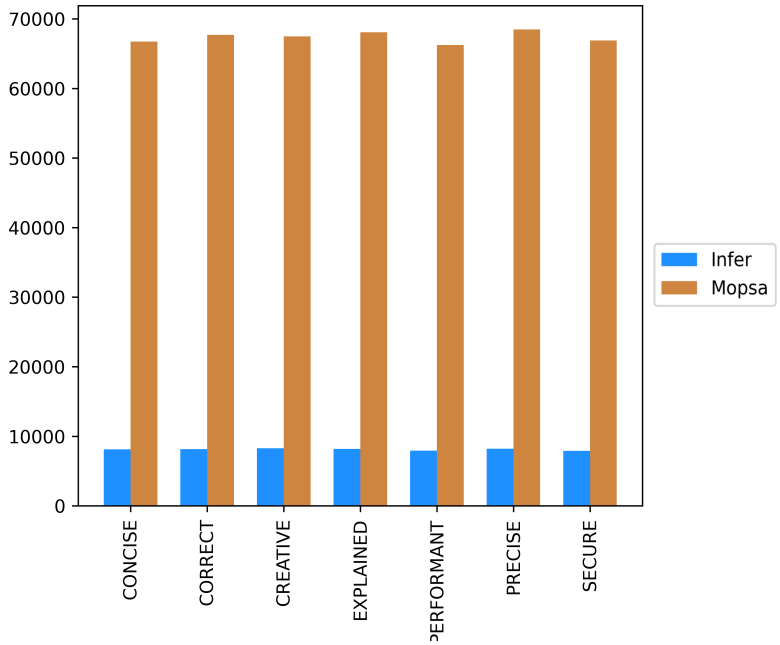


Figura 6.2: Suddivisione delle criticità trovate per stile di programmazione.



Per quanto riguarda l'analisi con Mopsa, non sono stati impiegati domini di tipo relazionale al fine di limitare il numero di timeout; infatti, quelli di tipo non

relazionale sono generalmente meno dispendiosi in termini di risorse, permettendo di completare la quasi totalità delle analisi entro i 180 s impostati. Una valida alternativa per future ricerche potrebbe essere quella di utilizzare un dominio di tipo relazionale, come ad esempio quello dei *poliedri convessi* della PPLite, per provare a ridurre il numero di falsi positivi a fronte di un tempo di computazione maggiore.

Considerando invece la raccolta dei sorgenti utilizzati come benchmark, si può notare che dei 525 task ottenuti nessuno di questi supera le 250 righe di lunghezza; si potrebbe dunque pensare di osservare il comportamento dei modelli nella generazione di applicazioni complete, come ad esempio siti web o applicazioni desktop, per capire se tale complessità influenzi l'affidabilità del codice generato.

Infine, si ricordi che in questo esperimento è stata tralasciata la valutazione semantica dei programmi generati; sebbene ci si aspetta che nella maggior parte dei casi i LLM producano codice che risolve uno specifico problema, una futura ricerca potrebbe indagare la correttezza delle soluzioni proposte e misurarne l'effettivo grado di sicurezza che esse offrono.

Bibliografia

[1] Microsoft. (2023) Copilot, your AI pair programmer. [Online]. Available: https://github.com/features/copilot

[2] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. Canton-Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," *CoRR*, vol. abs/2307.09288, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2307.09288

[3] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. Canton-Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," *CoRR*, vol. abs/2308.12950, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.12950

[4] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de Las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7b," *CoRR*, vol. abs/2310.06825, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.06825

[5] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. de Las Casas, E. B. Hanna, F. Bressand, G. Lengyel, G. Bour, G. Lample, L. R. Lavaud, L. Saulnier, M. Lachaux, P. Stock, S. Subramanian, S. Yang, S. Antoniak, T. L.

Scao, T. Gervet, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mixtral of experts," *CoRR*, vol. abs/2401.04088, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.04088

[6] R. Monat, A. Ouadjaout, and A. Miné, "Mopsa-c: Modular domains and relational abstract interpretation for C programs (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, ser. Lecture Notes in Computer Science, S. Sankaranarayanan and N. Sharygina, Eds., vol. 13994. Springer, 2023, pp. 565–570. [Online]. Available: https://doi.org/10.1007/978-3-031-30820-8_37

[7] "Modular Open Platform for Static Analysis." [Online]. Available: https://mopsa.lip6.fr/

[8] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of C programs," in *NASA Formal Methods - Third International Symposium, NFM 2011 , Pasadena, CA, USA, April 18-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., vol. 6617. Springer, 2011, pp. 459–465. [Online]. Available: https://doi.org/10.1007/978-3-642-20398-5_33

[9] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, ser. Lecture Notes in Computer Science, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., vol. 9058. Springer, 2015, pp. 3–11. [Online]. Available: https://doi.org/10.1007/978-3-319-17524-9_1

[10] "A tool to detect bugs in Java and C/C++/Objective-C code before it ships." [Online]. Available: https://fbinfer.com/

[11] A. Becchi and E. Zaffanella, "Pplite: Zero-overhead encoding of nnc polyhedra," *Information and Computation*, vol. 275, p. 104620, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540120301085

[12] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, p. 576–580, oct 1969. [Online]. Available: https://doi.org/10.1145/363235.363259

[13] E. Clarke, O. Grumberg, and D. Long, "Model checking," 1986.

[14] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," 1977. [Online]. Available: https://courses.cs.washington.edu/courses/cse503/10wi/readings/p238-cousot.pdf

[15] A. Miné, "Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation," *Foundations and Trends in Programming Languages*, vol. 4, no. 3-4, pp. 120–372, 2017. [Online]. Available: https://hal.sorbonne-universite.fr/hal-01657536

[16] M. Journault, A. Miné, R. Monat, and A. Ouadjaout, "Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer," in *VSTTE 2019 : 11th Working Conference on Verified Software: Theories, Tools, and Experiments*, ser. Lecture Notes in Computer Science, S. Chakraborty and J. A. Navas, Eds., vol. 12031. New York, United States: Springer, Jul. 2019, pp. 1–18. [Online]. Available: https://hal.sorbonne-universite.fr/hal-02890500

[17] Miné, Antoine, "Apron." [Online]. Available: https://antoinemine.github.io/Apron/doc/

[18] Jeannet, Bertrand, "Polka," 2007. [Online]. Available: https://pop-art.inrialpes.fr/people/bjeannet/newpolka/polka.pdf

[19] "Meta." [Online]. Available: https://about.meta.com/it/

[20] M. Turisini, G. Amati, and M. Cestari, "Leonardo: A pan-european pre-exascale supercomputer for hpc and ai applications," 2023.

[21] "Hugging Face." [Online]. Available: https://huggingface.co/

[22] A. Miné, "Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics," 2007.

[23] "Calcolo Scientifico dell'Università di Parma." [Online]. Available: https://www.hpc.unipr.it/dokuwiki/doku.php

Glossario deli acronimi e dei termini

AbsInt Abstract Interpretation. 3–6

ACG Automatic Code Generation. 1

AI Artificial Intelligence. 1

AST Abstract Syntax Tree. 19

Code template Con il termine **code template** si intende un frammento di codice più o meno esteso pensato per essere riutilizzato. Tipicamente sono associati a elementi ricorrenti del software, quali ad esempio routine di manipolazione dei dati o classi con metodi parzialmente completati. Essi velocizzano i tempi di produzione del software e facilitano la riduzione di errori di programmazione su task ripetitivi. 1

DSL Domain Specific Language. 1

LLM Large Language Model. 1, 2, 23, 25, 26, 29, 37, 38, 43–46, 49, 51

Ringraziamenti

Desidero come prima cosa ringraziare il mio relatore, il prof. Enea Zaffanella, per l'attenzione, la disponibilità e la costanza che mi ha dimostrato. Grazie, per avermi guidato nella stesura di questo elaborato e per avermi arricchito con i suoi insegnamenti. Non meno importante è stato il contributo del mio correlatore, il prof. Vincenzo Arceri, che ringrazio per i suoi consigli e la dedizione che quotidianamente dimostra verso noi studenti. Un sincero ringraziamento anche a Greta Dolcetti, le cui idee hanno permesso la realizzazione di questo documento. Voglio poi ringraziare il mio collega, nonché amico, Saverio Mattia Merenda, per aver collaborato all'esperimento ed essermi stato accanto in questi ultimi anni.

Ringrazio la mia famiglia. Mia madre, per avermi trasmesso la sua determinazione e per l'impegno e l'amore che mi ha dedicato; mio padre, per essere stato un esempio di passione e dedizione, e per avermi insegnato il valore del rispetto e della sincerità; mia sorella, per ricordarmi la virtù della spensieratezza. Grazie ai miei nonni e ai miei zii, che mi hanno supportato e sostenuto anche da lontano.

Una menzione speciale ai miei amici, senza i quali non avrei raggiunto questo traguardo. Grazie, per avermi accompagnato in questo percorso universitario ed essermi stati vicini, condividendo preziosi momenti di quotidianità. Ringrazio Arianna, Elia, Simone e Kevin, per i consigli, l'onestà e l'affetto che mi hanno dato; il mio amico *di giù* Daniel, per esserci sempre e per sostenermi giorno dopo giorno; Chiara, Irene e Lorenzo, per avermi strappato un sorriso anche nei momenti di sconforto. Vi voglio bene.