MirChecker: Detecting Bugs in Rust Programs via Static Analysis

Nel paper "MirChecker: Detecting Bugs in Rust Programs via Static Analysis" viene proposto un framework per la scoperta di bug nel linguaggio di programmazione Rust che sfrutta static analysis applicata sulla MIR (Mid-Level Intermediate Representation) di Rust.

Rust è un linguaggio di programmazione noto per permettere lo sviluppo di programmi veloci e sicuri, ma presenta dei problemi collegati all'utilizzo della keyword *unsafe,* la quale viene adottata per concedere al programmatore la libertà di mandare in esecuzione programmi che normalmente verrebbero rifiutati dal compilatore Rust.

In particolare, i bug presi in considerazione sono di due categorie:

- **Runtime panics:** i check di alcune condizioni come i boundaries degli array oppure integer overflow detection sono posticipati a runtime in Rust, questo vuol dire che in alcuni casi verrà abortita l'esecuzione per evitare memory corruption, causando verosimilmente DoS.
- **Lifetime corruption:** l'utilizzo della keyword *unsafe* potrebbe causare puntatori invalidi o shared mutable aliases ed in seguito, il meccanismo di ownership, potrebbe eseguire drop di tali valori causando use-after-free o double-free.

L'approccio svolto è stato quello di combinare *numerical static analysis* e s*ymbolic execution* sul Rust MIR, il quale ha un language core molto più semplice del linguaggio Rust pur preservando type information ed altre informazioni utili al fine della static analysis. L'integer bound analysis è stata eseguita utilizzando un dominio numerico astratto, adatto per modellare applicazioni low-level e security-critical spesso integrate in sistemi embedded; tale dominio però, non è capace di rappresentare funzionalità complesse che sono presenti nel MIR come references e array, per questo motivo si è deciso di adottare anche la s*ymbolic execution*.

Il processo di analisi segue le tre fasi canoniche di design di uno static analyzer:

1) *User interface:* è realizzata come un subcommand di *Cargo* (Package manager di Rust) che accetta in input una Rust crate e delle opzioni di configurazione che servono all'utente per specificare quali file sorgente devono essere analizzati ed altre specifiche sull'analisi.
2) *Static analyzer:* lo static analyzer è stato realizzato come una versione modificata del Rust compiler il quale esegue degli step di verifica aggiuntivi.
3) *Bug detection:* quando l'analisi è completata vengono invocati molteplici bug detector atti a scoprire eventuali vulnerabilità.

Il tutto termina con l'emissione di messaggi di diagnostica strutturati che forniscono informazioni sul tipo di errore che potrebbe verificarsi e dove potrebbe verificarsi.

Come già anticipato, il dominio astratto adottato è tale da catturare gli aspetti relativi a valori numerici e valori simbolici durante l'esecuzione. Per questo motivo è stato definito un linguaggio astratto e sulla base di tale lingauggio sono stati definiti il dominio astratto e le *transfer functions*; il linguaggio è stato anche utlizzato per costruire le *symbolic expression* che servono a MirChecker per emulare gli accessi in memoria.

Dato che il principio su cui si basa MirChecker è quello di applicare numerical static analysis su programmi Rust l'insieme dei data types è stato semplificato e ridotto al solo insieme degli interi (il quale include boolean e interi signed ed unsigned di dimensione variabile). La sintassi di nostro interesse comprende:

- **p = r** : viene eseguito l'assignment di p con il valore espresso da r, p rappresenta un **place,** il quale può essere sia una variabile sia un path che rappresenta l'accesso ad un campo e r è un **Rvalue,** il quale esprime il risultato delle varie operazioni contemplate nel linguaggio.
- $op1 \oplus op2$ **:** viene applicata un operazione binaria $\oplus$ tra due operandi che possono essere costanti o place. Il tipo degli operandi e del risultato è intero.
- $op1 \otimes op2$ **:** viene applicata una operazione di confronto $\otimes$ tra due operandi. Il risultato è boolean e viene spesso utilizzato come guard delle branch conditions.
- **Call(*f*,[op1,op2, …],(p, b))) :** viene chiamata la funzione *f* con lista di argomenti [op1,op2, …] ed il valore di ritorno viene assegnato a p nel basic block b.
- **Drop(p) :** viene esplicitamente deallocata la memoria di p, allocata a runtime.
- **Assert(op) :** l'esecuzione continua se la condizione dell'assert è true, altrimenti viene attivato il meccanismo di runtime panic.
- **Goto(b) :** jump incondizionato al basic block b.
- **SwitchInt(op,[b1,b2, …]) :** istruzione di branch condizionale che trasferisce il CF ad uno dei basic block della target list, l'ultimo block rappresenta la scelta di default.

Il modello di memoria utilizzato sfrutta la rappresentazione di memoria del MIR Rust in cui la memoria viene trattata come una collezione di oggetti strutturati, ognuno identificato da una place expression. Quando viene acceduto un place viene costruita un'espressione simbolica che verrà utilizzata come abstract memory address.

In memoria viene mantenuta una lookup table σb **: P → V** che serve, dato un basic block b, a tenere traccia di tutti i valori astratti V che può assumere un place P dichiarato all'interno del block. Vengono anche definiti due valori speciali $\bot \in$ **V** e $\top \in$ **V** che rappresentano rispettivamente un valore non inizializzato e tutti i possibili valori. I valori astratti V possono appartenere a due categorie disgiunte: **NV** *numerical values* e **SV** *symbolic values*, i primi sono utilizzati per descrivere gli interi ed i loro bounds, i secondi sono utilizzati per definire abstract memory address e branch conditions.

Viene definito un *abstract state* **AS** come una map lattice che consiste del set di tutti i mapping P→V, ovvero di tutte le lookup table per ogni basic blocks. Similmente l'*abstract domain* **AD** viene definito come una map lattice che consiste di tutti i mapping B → AS dove B rappresenta il set di tutti i basic blocks del CFG. Basandosi sulle precedenti definizioni vengono definite le **TF** *transfer function*, le quali modellano ogni statement come un *abstract state transformer,* il quale prende in input l'AS del programma prima dello statement e ritorna l'AS del programma immediatamente dopo lo statement.

L'algoritmo implementato attraversa l'intero CFG ed iterativamente esegue static analysis finché non raggiunge un fixed point. Per ogni funzione si ha un CFG, il quale, oltre le espressioni da eseguire, contiene anche le relazioni di dipendenza tra i vari basic blocks. Per poter attraversare il CFG nel miglior modo possibile si utilizza la strategia di *weak topological ordering* WTO, una versione generalizzata di topological ordering che, a differenza dell'originale, funziona anche in presenza di loop.

Il CFG in input viene pre-processato ordinando tutti i basic blocks con strategia WTO ed il risultato è una lista di topological ordered *strongly connected components* **SCC** dove ogni SCC rappresenta o un singolo basic block (nel caso si tratti di una esecuzione sequenziale) oppure una lista di SCC (nel caso si tratti di un loop).

Rispettando l'ordine definito in precedenza, un *MIR visitor* attraversa ogni SCC del CFG e nel caso in cui si tratti di un'espressione sequenziale semplicemente analizza ogni statement e conseguentemente aggiorna gli *abstract values*. Nel caso in cui si tratti di un loop, il *visitor* attraversa gli statement ripetutamente finchè non si raggiunge un fixed point. Per assicurarsi che l'algoritmo termini sempre e non vada in loop è stata

adottata la tecnica standard di *widening and narrowing*. Alla fine di ogni basic block c'è uno statement speciale chiamato terminator, il quale gestisce il control flow tra i vai basic blocks; branch condizionali nel control flow sono rappresentati da uno SwitchInt terminator.

Quando il *visitor* incontra una funzione, se tale funzione è dichiarata staticamente nel crate sotto analisi verrà analizzata, altrimenti, se ha dependencies esterne viene semplicemente skippata per evitare path explosion. Per quanto riguarda le funzioni ricorsive il MirChecker evita ricorsioni infinite mantenendo una lista che simula il call stack, se viene eseguito il push di un function name già presente nella lista avviene un return in quanto è stata individuata una chiamata ricorsiva.

Una volta che l'algoritmo è terminato si vanno ad analizzare i risultati per scoprire possibili bug attraverso dei *bug detector* che analizzano sia il *numerical* sia il *symbolic* domain e risolvono delle condizioni attraverso SMT solving engine; i detector generano dei messaggi nel momento in cui una security condition potrebbe essere violata. Le security condition da verificare per ogni tipologia di bug sono:

- **Runtime panics:** il runtime panic detector attraversa il CFG e raccoglie tutte le condizioni nei terminator di tipo Assert(cond) e in seguito traduce gli integer bounds dal numerical abstract domain e i valori simbolici dal symbolic abstract domain in formule SMT che vengono prese in input da un SMT solver.
- **Lifetime Corruption:** per scoprire lifetime corruption il MirChecker mantiene una lista interna di funzioni *unsafe*. Durante la symbolic analysis, la funzione di trasferimento raccoglie la transizione di ownership legata alla funzione unsafe ed il lifetime corruption detector verifica se l'owner originale viene utilizzato dopo che l'ownership del valore è stata trasferita.

Uno dei vantaggi nell'utilizzare MIR è che la lifetime di ogni variabile viene codificata esplicitamente dagli statement *StorageLive* e *StorageDead.* Quando il MirChecker ha meccanismo di dead variable cleaning attivo ed incontra uno statement *StorageDead*, cerca tutte le dead variables in entrambi i domini e le elimina, liberando memoria e riducendo i tempi di esecuzione.

Per poter implementare il MirChecker sono state utilizzate diverse librerie C (MIRAI per i meccanismi di symbolic evaluation, Apron per gestire gli abstract numerical domain, MST solver Z3 per risolvere i sistemi di constraint), integrate attraverso Rust Foreign Function Interface (FFI) e riadattate al caso di utilizzo. Durante l'analisi vengono utilizzati diversi *converter*: a partire dagli statement, MirChecker genera un set di linear constraint e li converte in un formato compatibile con Apron; una volta che l'algoritmo è terminato i constraint vengono ulteriormente tradotti in espressioni Z3 in modo da poter sfruttare il solving engine Z3.

Il sistema realizzato è stato poi testato su più di 1000 crates ottenute da crates.io e GitHub, filtrando per repository in cui è presente la keyword *unsafe*. Per ogni crate sono state estratte le funzioni pubbliche e i metodi e sono stati utilizzati come entry function della static analysis.

In totale 17 runtime panics e 16 memory-safety issues sono stati individuati in 12 crates ufficiali, ed in seguito è stato eseguito un controllo manuale per verificare che tali warning fossero reali. La maggior parte dei warning generati sono in realtà falsi positivi, le cause sono le seguenti:

- La natura della static analysis, la quale approssima per eccesso il runtime behaviour, garantisce sicuramente soundness ma non precision.
- Molti warning sono dovuti a delle macro realizzate ad-hoc che causano panic mechanism by design. Tali warning non vanno considerati come errori in quanto sono statement intenzionali inseriti dal programmatore.
- Un singolo Bug potrebbe essere triggerato da diversi path di esecuzione, causando molteplici warning.

Per ridurre il numero di falsi positivi sono state introdotte delle opzioni dal subcommand cargo che permettono di sopprimere alcuni warning, ad esempio quelli relativi alla chiamata di macro. Ma, in alcuni dataset questo approccio non ha migliorato i risultati, avendo comunque un rate di Falsi Positivi > 90%.

I risultati sono molto incoraggianti ed il fatto che i bug trovati sono per lo più di runtime panic e non di memory safety dimostra ulteriormente che Rust è un linguaggio affidabile e memory-safe di default.

Pro

MirChecker risulta uno strumento valido in quanto è riuscito ad individuare dei bug mai individuati in precedenza su crates ufficiali Rust (17 runtime panic e 16 memory-safety issues). Si tratta di uno strumento lightweight e performante che può essere utilizzato per individuare e gestire staticamente panic mechanism che vengono solitamente attivati a runtime, evitando in questo modo possibili DoS.

È semplice da utilizzare ed integrare, in quanto definito come subcommand del package manager Rust cargo, quindi senza alterare il normale workflow. L'analisi può essere personalizzata andando a specificare i crates su cui deve essere condotta e/o modificando alcune opzioni di analisi come i parametri di widening e narrowing e l'attivazione o meno di dead variable cleaning.

Contro

L'analisi svolta da MirChecker non è esaustiva in quanto esso lavora su un modello di memoria semplificato basato su symbolic expression che non contempla molte operazioni di memoria complesse.

Molte funzionalità avanzate di Rust come closures, high order function e concorrenza non vengono prese in considerazione. Inoltre, non prende in considerazione funzioni che non sono state dichiarate staticamente nel crate analizzato e funzioni ricorsive, le quali per semplicità vengono skippate.

Possibili Improvements:

Considerando i pro e i contro, MirChecker risulta uno strumento affidabile per l'individuazione e la gestione di runtime panics mechanism, ma non può essere considerato uno strumento affidabile al 100% per l'individuazione di bug. MirChecker non può provare l'assenza di bug in un codice, dato che alcuni bug potrebbero essere causati dalla funzionalità e da aspetti non presi in considerazione dall'analisi.

I possibili improvements da considerare per migliorare il tool potrebbero essere: ridurre il tasso di falsi positivi, magari sopprimendo a prescindere i warning causati da macro utilizzate dal programmatore come *Panic!()* e *Unreachable!()* e rendendo il tool capace di individuare quando uno statement causa più warning in quanto presente in più path di esecuzione. Rendere l'analisi più precisa, introducendo supporto per l'analisi interprocedurale e integrando l'analisi di funzioni presenti in dependencies del crate sotto analisi. Rendere possibile l'analisi di funzioni ricorsive, magari introducendo un meccanismo tale da poter riconoscere ricorsioni infinite basandosi su una certa soglia, come nel caso dei loop.