Securing ARM Binaries with Model-Checking

Kausika Manivannan (kmaniva1@charlotte.edu)
Advisors: Dr. Meera Sridhar (msridhar@charlotte.edu), **Dr. Harini Ramaprasad** (hramapra@charlotte.edu)
College of Computing and Informatics, UNC Charlotte

UNIVERSITY OF NORTH CAROLINA CHARLOTTE | COLLEGE OF COMPUTING AND INFORMATICS

Motivation

- Users of software products consistently face cyber attacks.
- Systems Security personnel are burdened by responding to frequent emergencies.
- Eliminating entire classes of vulnerabilities could reduce potential cyber attacks.
- It would alleviate the burdens carried by the frontline and strengthens users' security

Introduction

Goal: To develop a formal reasoning framework for checking whether a given ARM binary adheres ensure to certain security properties, such as memory safety.

- We achieve this by implementing a model-checker on top of a Prolog interpreter for PCode code.
- PCode code has well established operational semantics, making it more reliable to formal analysis compared to ARM binaries.

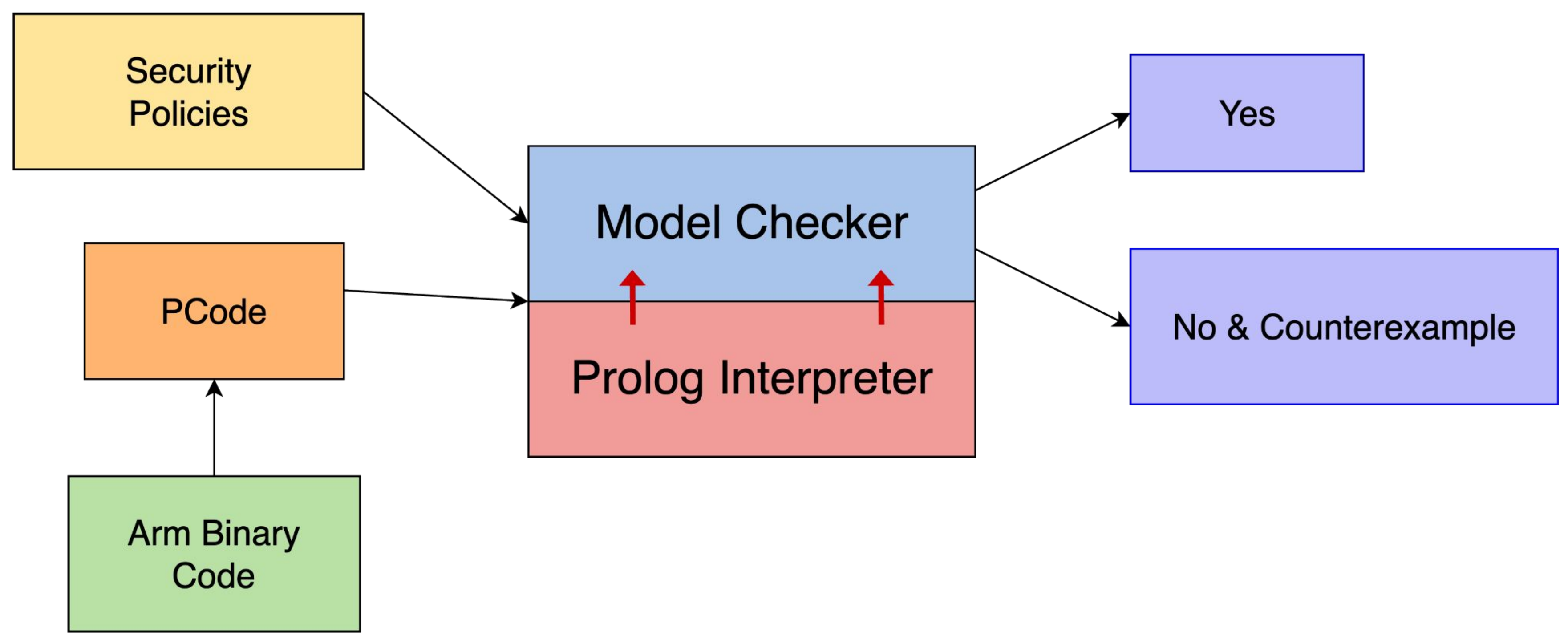
Tools and Technologies

- **Prolog** is a logic based programming language that utilizes formal semantics, consisting of rules and facts.
- **IMP** is a programming language that follows the operation semantic structure. It is a non executable code that represents imperative programming concepts, such as C and Java.
- **PCode** is an intermediate language that is compatible for analysis using the NSA tool Ghidra. Targets (virtual) p-code machine. It allows the same program to be executed on different platforms, making it very versatile.
- **ARM** is a type of machine executable that provides instructions to ARM processors.

Implementing Our Model Checker (MC)

- We convert ARM binary code to PCode to supply as input to MC.
- MC built on Prolog interpreter for PCode code.
- MC identifies policy violations (e.g., violation of memory safety) and path to failure.

ARM / PCode Model Checker



Model Checker Flow

Project Roadmap

- Jan. 2024 — Began the research project
- Mar. 2024 — Developed a Prolog interpreter capable of translating IMP code into an executable format
- Apr. 2024 — Developing a parser in Prolog that allows our interpreter to take PCode programs as input
- Aug. 2024 — Resume the research project
- Complete building a Prolog interpreter for a subset of PCode core instructions
- Construct the MC on top of the Prolog Interpreter for a larger PCode subset
- Dec. 2024 — Have a fully developed MC

Formal Methods Background

- **Operational Semantics:**
  - Provides a formal framework for how programs operate.
- **Static Analysis**
  - Automated process that examines code without executing it.
  - Identifies vulnerabilities such as bugs within code.
  - Prone to making false warnings about flaws in programs.
  - Examples of this tool: FindBugs for Java,
  - Lint for C.
- **Model Checking**
  - Explores all possible program states in a system to check if it adheres to a specific security policy.
  - Provides counterexamples that show developers how their programs' behaviors violate a policy.

Prolog Arithmetic Interpreter

- Interpreters read and execute code line by line, in real-time.
- Our Prolog interpreter translates arithmetic IMP code into executable code.
- It includes predicates that define operations for addition, subtraction, multiplication, and division.
- The **example** and **example2** predicates demonstrate how the interpreter operates by assigning values to variables and setting up environments.

Prolog Arithmetic Interpreter for IMP

```
% predicate for addition
eval_plus(add(A, B), Res, Env) :-
    eval_expr(A, Aval, Env),
    eval_expr(B, Bval, Env),
    Res is Aval + Bval.

% predicate for subraction
eval_min(sub(A, B), Res, Env) :-
    eval_expr(A, Aval, Env),
    eval_expr(B, Bval, Env),
    Res is Aval - Bval.

% predicate for multiplication
eval_mult(mult(A, B), Res, Env) :-
    eval_expr(A, Aval, Env),
    eval_expr(B, Bval, Env),
    Res is Aval * Bval.

% predicate for division
eval_div(div(A, B), Res, Env) :-
    eval_expr(A, Aval, Env),
    eval_expr(B, Bval, Env),
    Res is Aval / Bval.
```

```
eval_expr(Num, Num, _Env) :- number(Num).
eval_expr(Var, Val, Env) :-
    atom(Var),
    get_assoc(Var, Env, Val).

assign(Var, Val, Env, NewEnv) :-
    put_assoc(Var, Env, Val, NewEnv).

example(Result) :-
    empty_assoc(EmptyEnv),
    assign(x, 9, EmptyEnv, Env1),
    assign(y, 10, Env1, Env2),
    eval_plus(add(x, y), Result, Env2).

example2(Result2) :-
    empty_assoc(EmptyEnv2),
    assign(t, 10, EmptyEnv2, Env4),
    assign(q, 2, Env4, Env5),
    eval_mult(mult(t,q), Result2, Env5).
```

```
?- example(Result).
Result = 19

?- example2(Result2).
Result2 = 20
```

References

D'Silva, V., Kroening, D., & Weissenbacher, G. (2008, June 17). A Survey of Automated Techniques for Formal Software Verification. IEEEXplore. https://ieeexplore.ieee.org/document/4544862/authors

Prolog's features. SWI. (n.d.). https://www.swi-prolog.org/features.html

Myers, A. (2013, February 1). Cornell. IMP: Big-Step and Small-Step Semantics. http://www.cs.cornell.edu/courses/cs6110/2013sp/lectures/lec05-sp13.pdf

The White House, 1 Back To The Building Blocks: A Path Toward Secure And Measurable Software (2024). Washington D.C.

Wikimedia Foundation. (2024, April 7). P-Code Machine. Wikipedia. https://en.wikipedia.org/wiki/P-code_machine