Compilation and Optimization with Security Annotations

Exploring the expression, use and propagation of functional and non-functional properties across the compilation flow

Son Tuan Vu¹, Karine Heydemann¹, Arnaud de Grandmaison², Albert Cohen³

¹Sorbonne University, ²ARM, ³Google
son-tuan.vu@lip6.fr

1 Problemsatics

- Annotation languages have been proposed to specify properties, usually functional, in the source programs to provide additional information [1]. However, for the purpose of implementing secure code, there has been little effort to support non-functional properties about side-channels or faults.
- Securing code involves enforcing and checking such properties on the program binary representation. We thus need an automated approach to carry-source-level annotations across the compilation flow, interacting safely with optimizations and lowering steps, and to capture them at binary level.

2 Objectives

A complete workflow using annotations:

(1) Annotated source code
(2) Compiler
(3) Binary + Annotations
(4) Binary analysis tool
(5) Source code analysis tool
(6) Binary annotation tool

This comprises:
1. An annotation language that allows expressing security-related properties
2. An optimizing, annotation-aware C compiler able to propagate source-level annotations, controlling their interaction with compilation passes, and to emit them into the executable binary
3. A representation of the annotations at the binary level

3 Annotation Language

Source-level language
- Based on ANSI-C Specification Language (ACSL) [1], designed to specify functional properties to be verified by source code analyzers
- Extended with semantic predicate and semantic variables to capture side-effects of the code
- Annotation representation:
  - Function: Annotated Entity ⊔ Predicate ⊔ Referenced Variables
  - Annotated Entity = Function ⊔ Predicate ⊔ Semantic Predicate

```
#define ANNOT(s) __attribute__((annotate(s)))
```

Example:
```
int verifyPIN(char *cardPin, char *userPin) {
  int i, diff = 0;
  for (i = 0; i < PIN_SIZE; i++)
    if (userPin[i] != cardPin[i])
      diff = 1;
  return BOOL_FALSE;
}
```

4 Annotations in LLVM

Binary-level representation
- Based on DWARF debugging information format [2] which provides mapping from source-level entities to their representation in the binary
- Introduced new tags and attributes to represent annotations

4.1 Annotations in LLVM

1. Binary-level representation
- Annotated Entity: new metadata node containing the predicate
- Annotated entity
  - Function or variable: debug information metadata
  - Statement: region delimited by so-called annotation markers
- Variables referenced in the annotation predicate: debug information metadata
- Annotated C source code
- Source code analysis tool
- Binary + Annotations
- Binary analysis tool

5 Preliminary Results

- Annotations found in DWARF section
- Code with protection inserted at source level: the protection may be removed by the compiler
- Traditionally, programmers compile the protected code without optimization or use fragile programming tricks to outwit the compiler
- SSA barriers prevent optimizations from removing the protection
- Tested on 2 different protections for the PIN authentication code: CFI [3] and loop protection [4]
- Simulated for ARM Cortex-M3: code generated using SSA barriers has about 50% less executed instructions than code generated without optimization and 30% less executed instructions than generated optimized code with programming tricks, while still preserving the protection

5.1 Test results

```
```n

6 Future Work

- Annotation correctness verification mechanism
- Pre-region optimization mechanism
- Rules for transforming annotations in the optimizer

References