Compilation and Optimization with Security Annotations

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

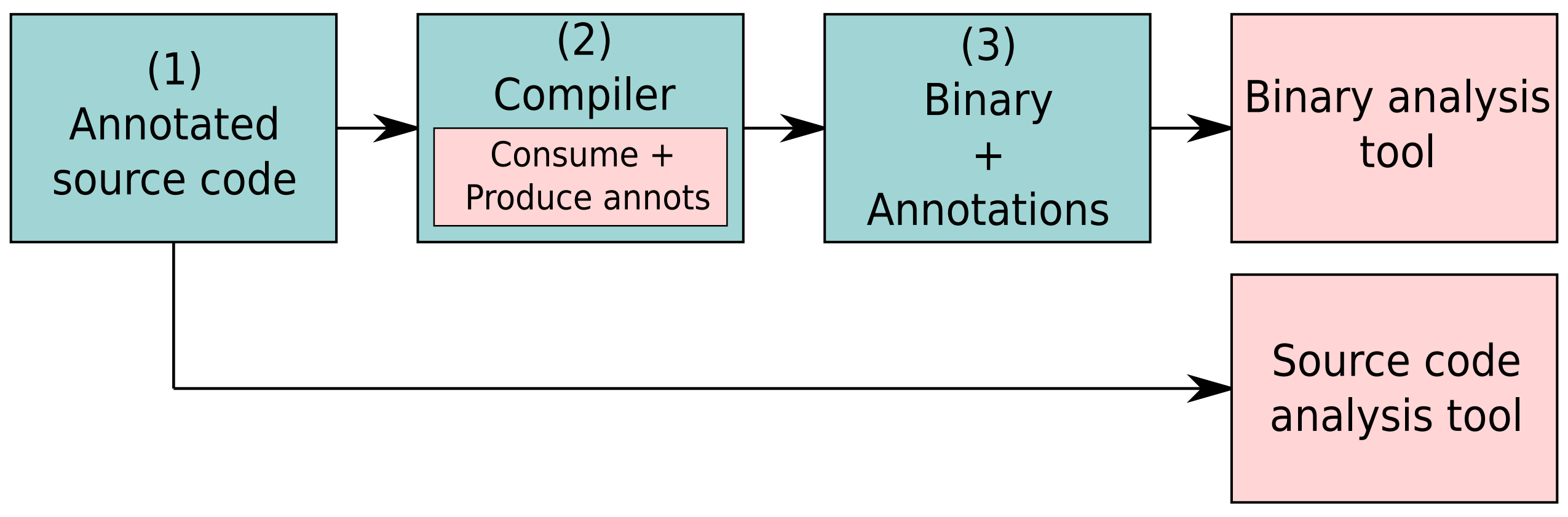
Son Tuan Vu[1], Karine Heydemann[1], Arnaud de Grandmaison[2], Albert Cohen[3]
[1]Sorbonne University, [2]ARM, [3]Google
son-tuan.vu@lip6.fr

1 Problematics

- 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
Consume +
Produce annots

(3)
Binary
+
Annotations

Binary analysis
tool

Source code
analysis 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 predicates* and *semantic variables* to capture side-effects of the code
- Annotation representation:

$$Annotation = Annotated\ Entity \wedge Predicate \wedge Referenced\ Variables$$

$$Annotated\ Entity = Function \vee Variable \vee Statement$$

$$Predicate = Logic\ Predicate \vee Semantic\ Predicate$$

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

// Function annotation: the function returns BOOL_TRUE only when PPIN codes match
ANNOT("\\ensures \\result == 1 &&"
              " \\forall i; 0 <= i < 4: userPin[i] == cardPin[i];"
      "\\ensures \\result == 0 &&"
              " \\exists i; 0 <= i < 4: userPin[i] != cardPin[i];")
int verifyPIN(// Variable annotation: card PIN code should not be leaked
             ANNOT("\\invariant \\secret()") char *cardPin, char *userPin) {
  int i;
  int diff = 0;

  // Statement annotation: loop must be iterated exactly 4 times
  prop1: ANNOT("\\ensures \\count() == 4;")
  for (i = 0; i < PIN_SIZE; i++)
    if (userPin[i] != cardPin[i])
      diff = 1;

  // Statement annotation: the comparison is sensitive so should not be removed
  prop2: ANNOT("\\ensures \\sensitive();")
  if (i != 4)
    return BOOL_FALSE;

  if (diff == 0)
    return BOOL_TRUE;

  return BOOL_FALSE;
}
```

Listing 1: Interesting properties for an authentication code, expressed by the annotation language

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 Annotations in LLVM

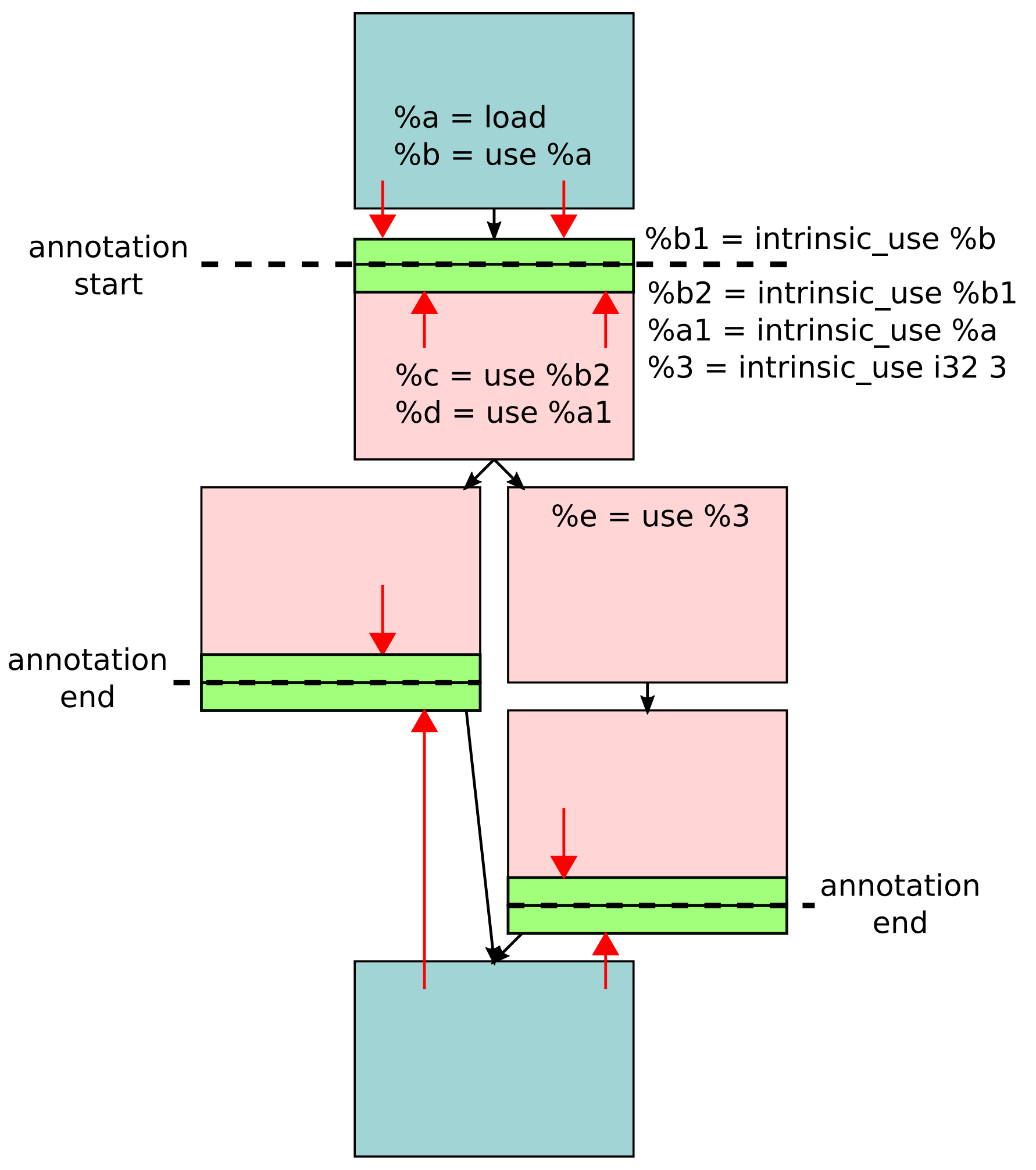
Two different problems: annotation representation and annotation propagation

Annotation representation

- Annotation: 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

Annotation propagation

- The annotation metadata itself is kept aside from the code and is not affected by optimizations
- Major challenges
  - Correctness of debug information for annotated entity and variables referenced by the annotation
  - Correctness of annotated region: SSA barriers to ensure isolation of the annotated region



%a = load
%b = use %a

annotation start

%b1 = intrinsic_use %b

%b2 = intrinsic_use %b1
%a1 = intrinsic_use %a
%3 = intrinsic_use i32 3

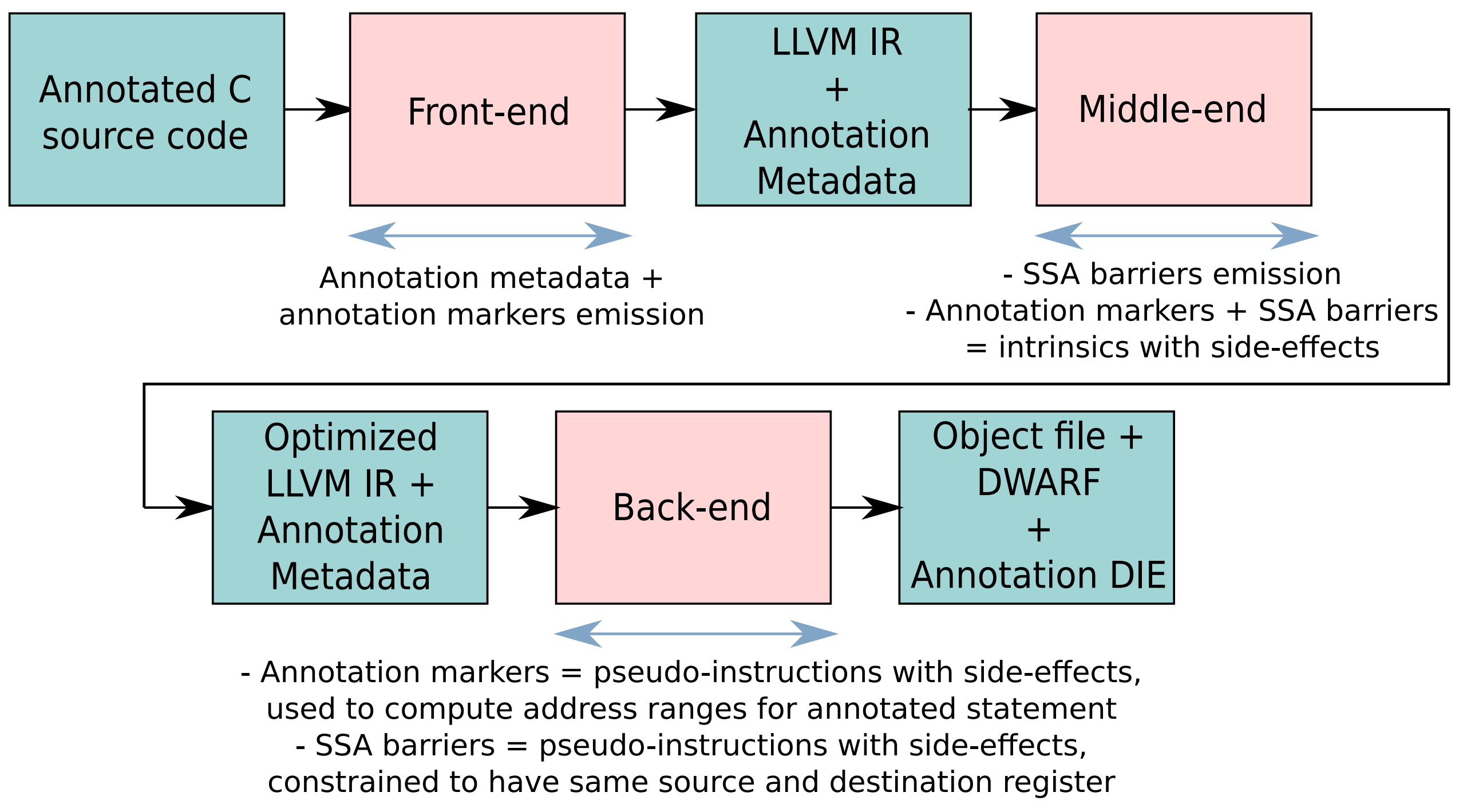
%c = use %b2
%d = use %a1

%e = use %3

annotation end

annotation end

Figure 1: Annotated region isolation by SSA barriers



Annotated C source code

Front-end

LLVM IR
+
Annotation Metadata

Middle-end

Annotation metadata +
annotation markers emission

- SSA barriers emission
- Annotation markers + SSA barriers
= intrinsics with side-effects

Optimized
LLVM IR +
Annotation
Metadata

Back-end

Object file +
DWARF
+
Annotation DIE

- Annotation markers = pseudo-instructions with side-effects,
used to compute address ranges for annotated statement
- SSA barriers = pseudo-instructions with side-effects,
constrained to have same source and destination register

Figure 2: Annotations throughout LLVM compilation flow

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

6 Future Work

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

References

[1] Patrick Baudin, Jean C. Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language Version 1.4*, May 2008.

[2] DWARF Debugging Information Format Commitee. *DWARF Debugging Information Format Version 5*, 2017.

[3] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. Software countermeasures for control flow integrity of smart card C codes. In Miroslaw Kutylowski and Jaideep Vaidya, editors, *ESORICS - 19th European Symposium on Research in Computer Security*, volume 8713 of *Lecture Notes in Computer Science*, pages 200–218, Wroclaw, Poland, September 2014. Springer International Publishing.

[4] Marc Witteman. Secure Application Programming in the Presence of Side Channel Attacks. Technical report.