Compilation and optimization with security annotations

Son Tuan Vu

Advisors: Karine Heydemann, Arnaud de Grandmaison, Albert Cohen

Team Alsoc
Laboratoire d'Informatique de Paris 6

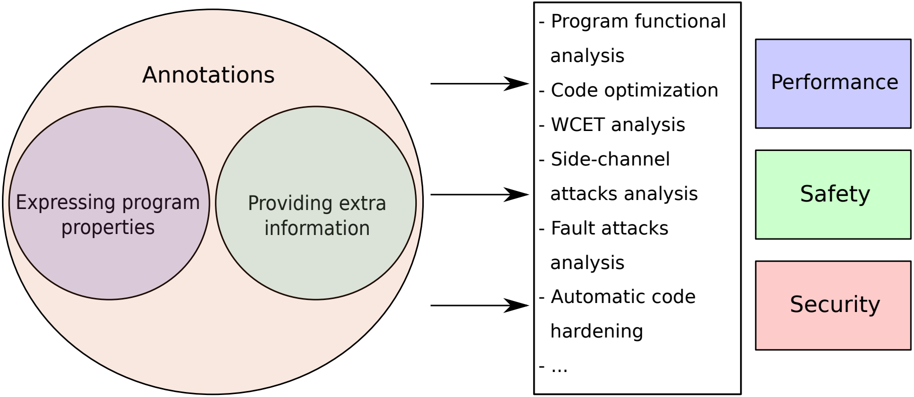
08 April 2019

Outline

1. Introduction
2. Proposed solutions
3. Conclusion
4. References

Background and motivation

- *Annotations = program properties + extra information*
- Applied to security, safety, real-time, optimization



- Annotations are consumed by program analysis or transformation
- Source level to binary level

Related work

- Annotation languages
  - GNU attributes, Microsoft's SAL, JML for Java, ACSL for C, etc.
  - At source-level

⇒ No annotation language covers the wide range of security properties

- Other usages than specifying program behaviors
  - Augment compiler optimizations [NZ13]
  - Automatic code hardening at compilation time [Hil14]
  - Flow information for Worst-Case Execution Time (WCET) analysis at binary level [SCG$^+$18]

⇒ No compiler propagating annotations until the binary other than WCET-aware compilers

Examples of properties: authentication code [DPP+16]

```
int verifyPIN(char *cardPin, char *userPin, int *cnt) {
  int i;
  int diff;
  if (*cnt > 0) {
    diff = 0;

    // Comparison loop
    for (i = 0; i < PIN_SIZE; i++)
      if (userPin[i] != cardPin[i])
        diff = 1;

    // Loop protection against fault attacks
    if (i != PIN_SIZE)
      return BOOL_FALSE;

    if (diff == 0) {
      // PIN codes match
      *cnt = MAX_ATTEMPT;
      return BOOL_TRUE;
    } else {
      // PIN codes differ
      (*cnt)--;
      return BOOL_FALSE;
    }
  }
  return BOOL_FALSE;
}
```

Examples of properties: authentication code [DPP+16]

```
int verifyPIN(char *cardPin, char *userPin, int *cnt) {
  int i;
  int diff;
  if (*cnt > 0) {
    diff = 0;

    // Comparison loop
    for (i = 0; i < PIN_SIZE; i++)
      if (userPin[i] != cardPin[i])
        diff = 1;

    // Loop protection against fault attacks
    if (i != PIN_SIZE)
      return BOOL_FALSE;

    if (diff == 0) {
      // PIN codes match
      *cnt = MAX_ATTEMPT;
      return BOOL_TRUE;
    } else {
      // PIN codes differ
      (*cnt)--;
      return BOOL_FALSE;
    }
  }
  return BOOL_FALSE;
}
```

Functional property:

- `verifyPIN` returns `BOOL_TRUE` only when PIN codes match

Examples of properties: authentication code [DPP+16]

```
int verifyPIN(char *cardPin, char *userPin, int *cnt) {
  int i;
  int diff;
  if (*cnt > 0) {
    diff = 0;

    // Comparison loop
    for (i = 0; i < PIN_SIZE; i++)
      if (userPin[i] != cardPin[i])
        diff = 1;

    // Loop protection against fault attacks
    if (i != PIN_SIZE)
      return BOOL_FALSE;

    if (diff == 0) {
      // PIN codes match
      *cnt = MAX_ATTEMPT;
      return BOOL_TRUE;
    } else {
      // PIN codes differ
      (*cnt)--;
      return BOOL_FALSE;
    }
  }
  return BOOL_FALSE;
}
```

Non-functional property:

- Card PIN code must be kept secret

Examples of properties: authentication code [DPP+16]

```
int verifyPIN(char *cardPin, char *userPin, int *cnt) {
  int i;
  int diff;
  if (*cnt > 0) {
    diff = 0;

    /********** Comparison loop **********/
    for (i = 0; i < PIN_SIZE; i++)
      if (userPin[i] != cardPin[i])
        diff = 1;

    // Loop protection against fault attacks
    if (i != PIN_SIZE)
      return BOOL_FALSE;

    if (diff == 0) {
      // PIN codes match
      *cnt = MAX_ATTEMPT;
      return BOOL_TRUE;
    } else {
      // PIN codes differ
      (*cnt)--;
      return BOOL_FALSE;
    }
  }
  return BOOL_FALSE;
}
```

Non-functional property:

- Comparison loop must be executed exactly `PIN_SIZE` times

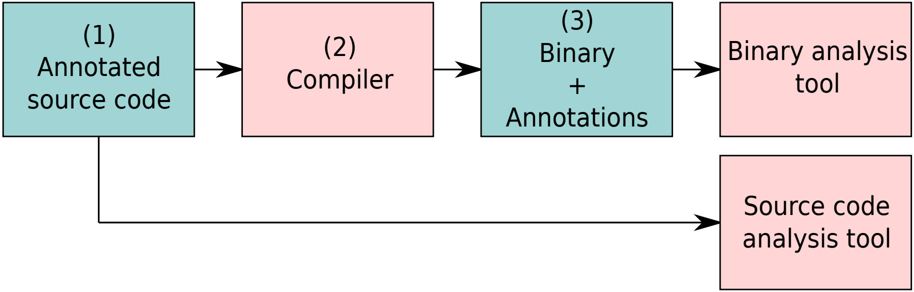
Examples of properties: authentication code [DPP+16]

```
int verifyPIN(char *cardPin, char *userPin, int *cnt) {
  int i;
  int diff;
  if (*cnt > 0) {
    diff = 0;

    // Comparison loop
    for (i = 0; i < PIN_SIZE; i++)
      if (userPin[i] != cardPin[i])
        diff = 1;

    /********** Loop protection against fault attacks **********/
    if (i != PIN_SIZE)
      return BOOL_FALSE;

    if (diff == 0) {
      // PIN codes match
      *cnt = MAX_ATTEMPT;
      return BOOL_TRUE;
    } else {
      // PIN codes differ
      (*cnt)--;
      return BOOL_FALSE;
    }
  }
  return BOOL_FALSE;
}
```

Non-functional property:

- Loop protection should not be removed by compiler optimizations

Problem statement



1. A source-level annotation language to express a wide range of properties
2. An annotation-aware, optimizing, LLVM-based compilation framework which consumes/produces/propagates annotations
3. A binary-level representation for the source-level annotation language

Outline

1. Introduction
2. Proposed solutions
   - Source-level annotation language
   - Binary-level representation of the annotation language
   - Annotations in LLVM: representation and propagation
3. Conclusion
4. References

Outline

1. Introduction

2. Proposed solutions
   - Source-level annotation language
   - Binary-level representation of the annotation language
   - Annotations in LLVM: representation and propagation

3. Conclusion

4. References

Annotation language by example: functional properties

ACSL already allows specifying program functional properties

- `verifyPIN` returns `BOOL_TRUE` only when PIN codes match

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

// Function annotation
ANNOT("\\ensures \\result == BOOL_TRUE &&"
               " \\forall i; 0 <= i < PIN_SIZE: userPin[i] == cardPin[i];"
      "\\ensures \\result == BOOL_FALSE &&"
               " \\exists i; 0 <= i < PIN_SIZE: userPin[i] != cardPin[i];")
int verifyPIN(char *cardPin, char *userPin, int *cnt) {
  ...
}
```

Annotation language by example: non-functional properties

Introduce *semantic predicates* to specify non-functional properties

- Card PIN code must be kept secret

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

// Variable annotation
int verifyPIN(ANNOT("\\invariant \\secret()") char *cardPin,
                                               char *userPin,
                                               int *cnt) {
  ...
}
```

Annotation language by example: non-functional properties

Introduce *semantic predicates* to specify non-functional properties

- Loop protection does not get removed

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

int verifyPIN(char *cardPin, char *userPin, int *cnt) {
  int i;
  int diff;
  if (*cnt > 0) {
    diff = 0;

    for (i = 0; i < PIN_SIZE; i++)
      if (userPin[i] != cardPin[i])
        diff = 1;

    // Statement annotation
    prop1: ANNOT("\\ensures \\sensitive();")
    if (i != PIN_SIZE)
      return BOOL_FALSE;

    if (diff == 0) {
      *cnt = MAX_ATTEMPT;
      return BOOL_TRUE;
    } else {
      (*cnt)--;
      return BOOL_FALSE;
    }
  }
  return BOOL_FALSE;
}
```

Annotation language by example: side-effect properties

Introduce *semantic variables* to capture side-effects of the code

- Comparison loop must be executed exactly `PIN_SIZE` times

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

int verifyPIN(char *cardPin, char *userPin, int *cnt) {
  int i;
  int diff;
  if (*cnt > 0) {
    diff = 0;

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

    if (i != PIN_SIZE)
      return BOOL_FALSE;

    if (diff == 0) {
      *cnt = MAX_ATTEMPT;
      return BOOL_TRUE;
    } else {
      (*cnt)--;
      return BOOL_FALSE;
    }
  }
  return BOOL_FALSE;
}
```

Annotation language summary

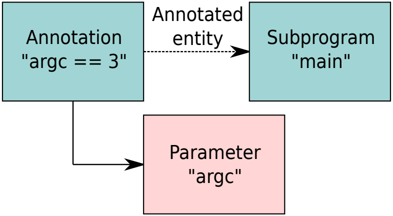
- $Annotation = Annotated\ Entity \wedge Predicate \wedge Predicate\ Variables$
- $Annotated\ Entity = Function \vee Variable \vee Statement$
- $Predicate = Logic\ Predicate \vee Semantic\ Predicate$
- $Predicate\ Variable = Variable\ Referenced\ in\ Predicate$

Outline

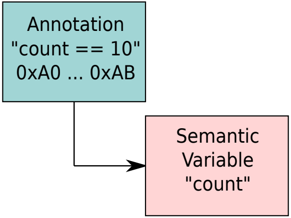
1. Introduction
2. Proposed solutions
    - Source-level annotation language
    - Binary-level representation of the annotation language
    - Annotations in LLVM: representation and propagation
3. Conclusion
4. References

Extending DWARF debugging format

- Executable program = tree of *Debugging Information Entries* (DIEs)
- DIE = tag + attribute(s) + child DIEs (if any)
- Introduce new tags and attributes to represent annotations and semantic variables



Function annotation



Statement annotation

Outline

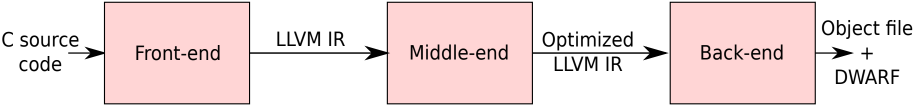
1 Introduction

2 Proposed solutions
- Source-level annotation language
- Binary-level representation of the annotation language
- Annotations in LLVM: representation and propagation

3 Conclusion

4 References

Annotation representation in LLVM



C source code

Front-end

LLVM IR

Middle-end

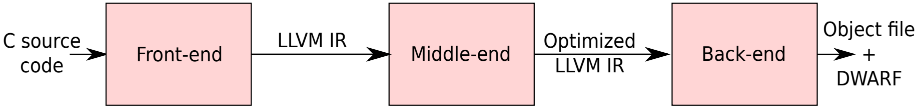
Optimized LLVM IR

Back-end

Object file + DWARF

- Existing metadata mechanism to convey extra information about the code
- Debug info: only metadata preserved and emitted into the binary
  ⇒ used to represent function and variable annotations

Annotation representation in LLVM

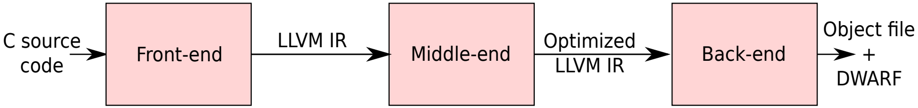


- Existing metadata mechanism to convey extra information about the code

- Debug info: only metadata preserved and emitted into the binary
  ⇒ used to represent function and variable annotations

- Debug info: does have representation for source statements, but too painful to maintain
  ⇒ annotation markers (≈ memory fences) to delimit the region corresponding to an annotated statement

Annotation representation in LLVM



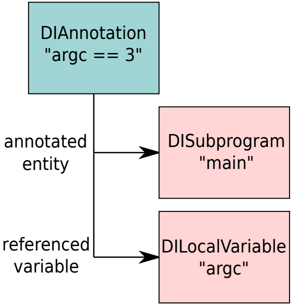
C source code → Front-end → LLVM IR → Middle-end → Optimized LLVM IR → Back-end → Object file + DWARF

- Existing metadata mechanism to convey extra information about the code
- Debug info: only metadata preserved and emitted into the binary
  ⇒ used to represent function and variable annotations
- Debug info: does have representation for source statements, but too painful to maintain
  ⇒ annotation markers (≈ memory fences) to delimit the region corresponding to an annotated statement
  ⇒ inspired by lifetime markers: all instructions from a start marker to a corresponding end marker are annotated

Annotation representation in LLVM: function and variable

Function + variable annotation metadata

- predicate
- reference to debug info metadata for the annotated entity
- reference to debug info metadata for the predicate variables (if any)



DIAnnotation "argc == 3"

annotated entity

DISubprogram "main"

referenced variable

DILocalVariable "argc"

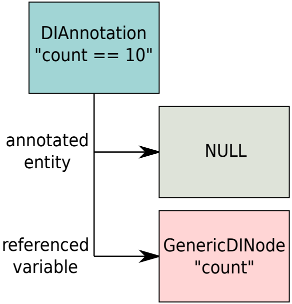
- Emitted by `clang`
- Propagated and emitted to the binary using the same mechanism as debug info metadata

Annotation representation in LLVM: statement

Statement annotation metadata

- predicate
- reference to debug info metadata for the predicate variables (if any)

- Emitted by `clang`
- Propagated and emitted to the binary using the same mechanism as debug info metadata



DIAnnotation "count == 10"

annotated entity

NULL

referenced variable

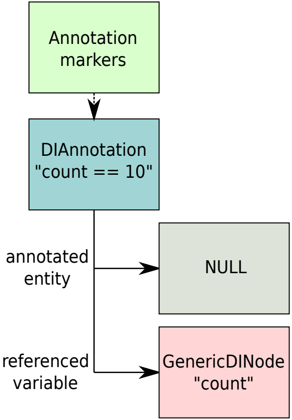
GenericDINode "count"

Annotation representation in LLVM: statement

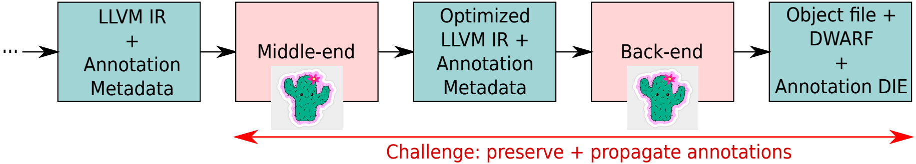
Statement annotation metadata

- predicate
- reference to debug info metadata for the predicate variables (if any)

- Emitted by `clang`
- Propagated and emitted to the binary using the same mechanism as debug info metadata
- Embedded in the annotation markers



Annotation propagation in LLVM: challenge


...

LLVM IR + Annotation Metadata

Middle-end

Optimized LLVM IR + Annotation Metadata

Back-end

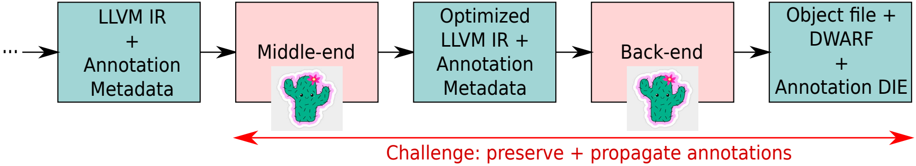
Object file + DWARF + Annotation DIE

Challenge: preserve + propagate annotations

Goal: preserving

1. the annotated entity
2. the predicate variables
3. the annotation metadata itself

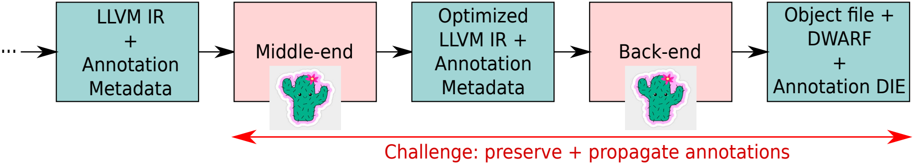
Annotation propagation in LLVM: challenge



Goal: preserving

1. the annotated entity
   ⇒ maintain correct debug info for variable and function annotations
2. the predicate variables
3. the annotation metadata itself

Annotation propagation in LLVM: challenge

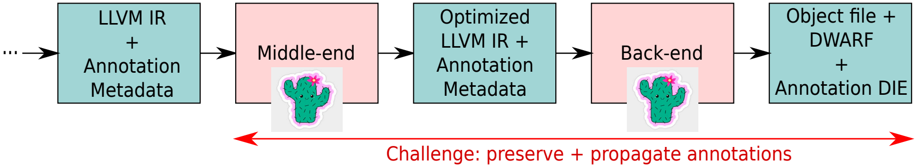


Goal: preserving

1. the annotated entity
   ⇒ maintain correct debug info for variable and function annotations

   ⇒ maintain correct annotated region for statement annotations
2. the predicate variables
3. the annotation metadata itself

Annotation propagation in LLVM: challenge



Challenge: preserve + propagate annotations

Goal: preserving

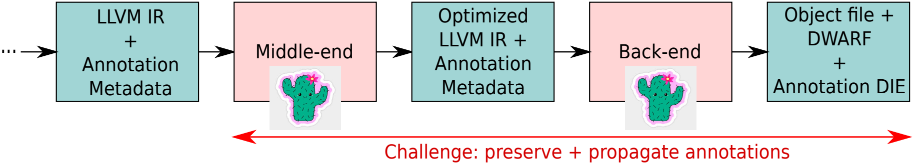
1. the annotated entity
   ⇒ maintain correct debug info for variable and function annotations

   ⇒ maintain correct annotated region for statement annotations

2. the predicate variables
   ⇒ maintain correct debug info for these variables

3. the annotation metadata itself

Annotation propagation in LLVM: challenge



Goal: preserving

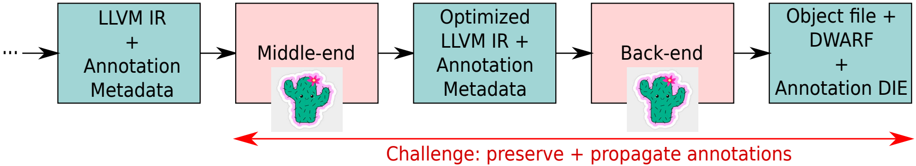
1. the annotated entity
   ⇒ maintain correct debug info for variable and function annotations

   ⇒ maintain correct annotated region for statement annotations

2. the predicate variables
   ⇒ maintain correct debug info for these variables

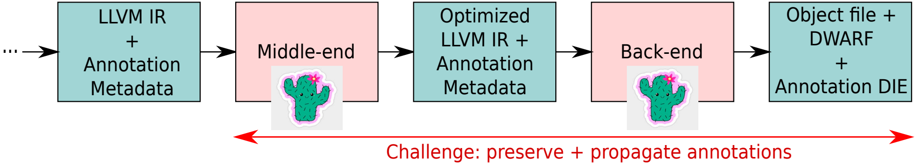
3. the annotation metadata itself
   ⇒ annotation metadata is kept aside from the code and does not interact with optimizations

Annotation propagation in LLVM: problems



Two different types of problems:

Annotation propagation in LLVM: problems



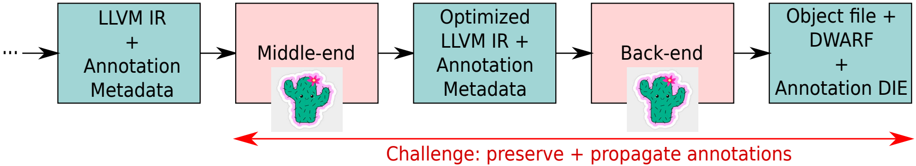
... → LLVM IR + Annotation Metadata → Middle-end → Optimized LLVM IR + Annotation Metadata → Back-end → Object file + DWARF + Annotation DIE

Challenge: preserve + propagate annotations

Two different types of problems:

1. Debug info propagation
   - Maintaining debug info = best-effort, no guarantee
   - Implementation bugs
   - Our biggest hurdle: correct location ranges for auto variables

Annotation propagation in LLVM: problems



... → LLVM IR + Annotation Metadata → Middle-end → Optimized LLVM IR + Annotation Metadata → Back-end → Object file + DWARF + Annotation DIE

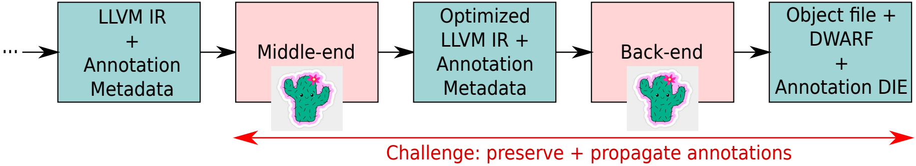
Challenge: preserve + propagate annotations

Two different types of problems:

1. Debug info propagation
   - Maintaining debug info = best-effort, no guarantee
   - Implementation bugs
   - Our biggest hurdle: correct location ranges for auto variables

     ⇒ analysis on the generated binary to recover the information

Annotation propagation in LLVM: problems



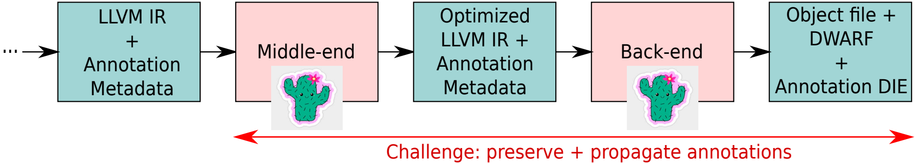
Two different types of problems:

1. Debug info propagation
    - Maintaining debug info = best-effort, no guarantee
    - Implementation bugs
    - Our biggest hurdle: correct location ranges for auto variables

      ⇒ analysis on the generated binary to recover the information

      ⇒ assume that debug info is correct for now

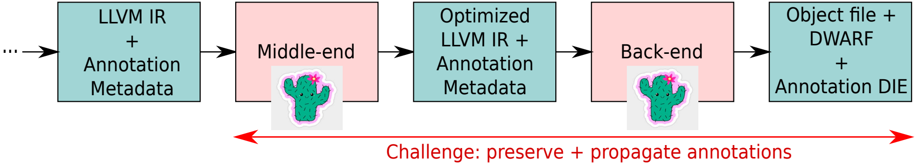
Annotation propagation in LLVM: problems



Two different types of problems:

1. Debug info propagation
2. Statement annotation propagation
    - Annotated instructions removed
    - Annotated instructions merged with not annotated ones, or with ones annotated with a different annotation

Annotation propagation in LLVM: problems

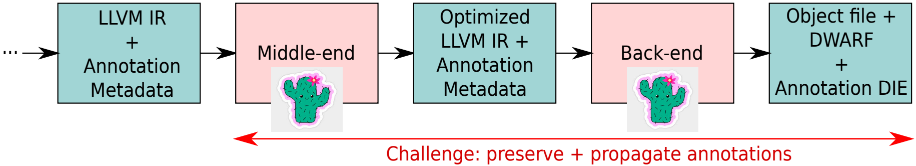


Two different types of problems:

1. Debug info propagation
2. Statement annotation propagation
    - Annotated instructions removed
    - Annotated instructions merged with not annotated ones, or with ones annotated with a different annotation

⇒ How to preserve an annotated region?

Annotation propagation in LLVM: problems



Challenge: preserve + propagate annotations

Two different types of problems:

1. Debug info propagation
2. Statement annotation propagation
   - Annotated instructions removed
   - Annotated instructions merged with not annotated ones, or with ones annotated with a different annotation

⇒ How to preserve an annotated region?

⇒ What does "preserving an annotated region" even mean?

Statement annotation propagation in LLVM: correctness

An annotated region is preserved

1. Isolation conditions (can be relaxed, depending on the annotation's nature)
2. Optimization conditions for the annotated region

Statement annotation propagation in LLVM: correctness

An annotated region is preserved

1. Isolation conditions (can be relaxed, depending on the annotation's nature)
    - no external instructions should get into the region
    - no annotated instructions should get out of the region

Statement annotation propagation in LLVM: correctness

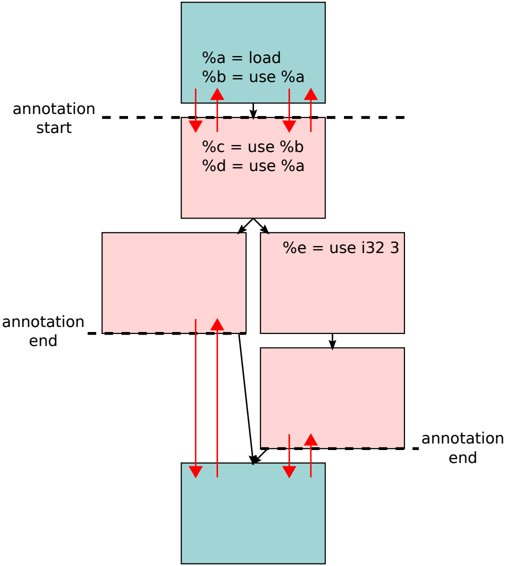
An annotated region is preserved

1. Isolation conditions (can be relaxed, depending on the annotation's nature)
    - no external instructions should get into the region
    - no annotated instructions should get out of the region

   ⇒ annotation markers only guarantee for memory accesses and instructions with side-effects

   What about constants, registers, instructions without side-effects?

Statement annotation propagation in LLVM: SSA barriers

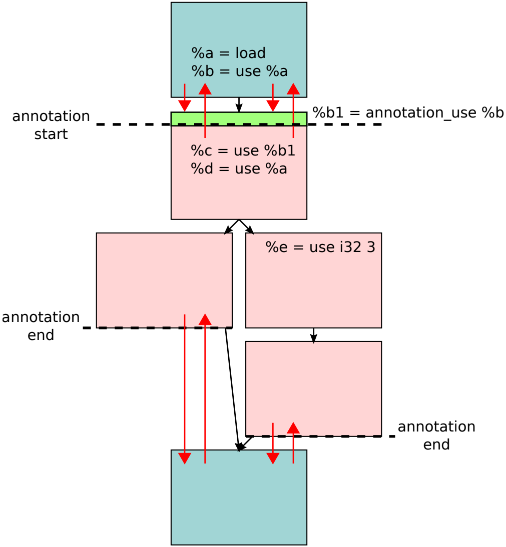


```
%a = load
%b = use %a
```

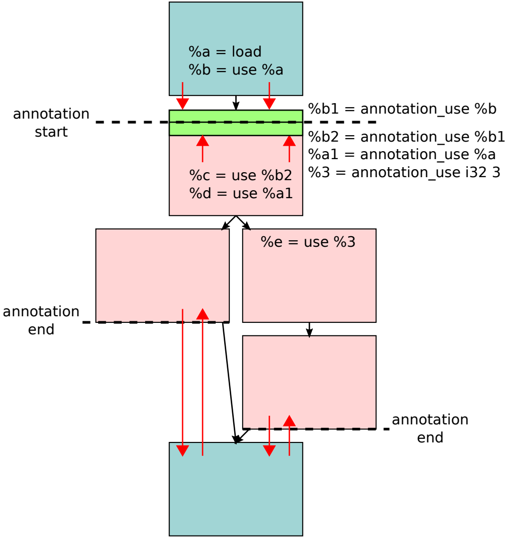
```
%c = use %b
%d = use %a
```

```
%e = use i32 3
```

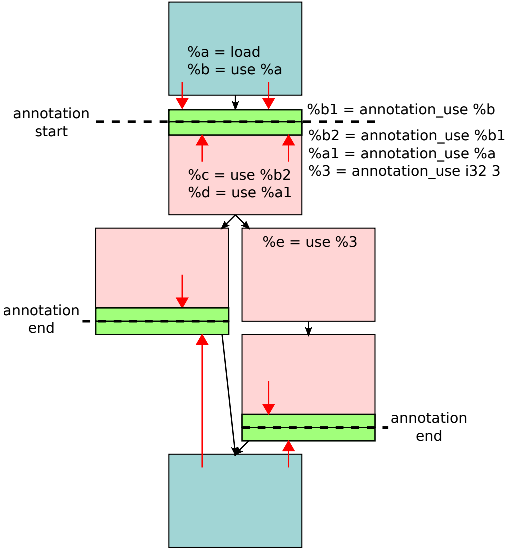
Statement annotation propagation in LLVM: SSA barriers



Statement annotation propagation in LLVM: SSA barriers

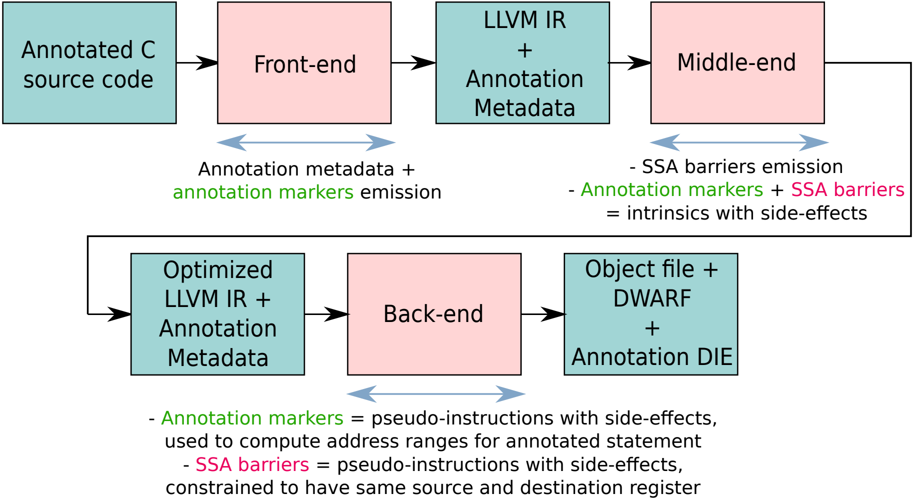


Statement annotation propagation in LLVM: SSA barriers



Annotation propagation in LLVM: complete flow

Current implementation



Annotated C source code

Front-end

LLVM IR + Annotation Metadata

Middle-end

Optimized LLVM IR + Annotation Metadata

Back-end

Object file + DWARF + Annotation DIE

Annotation metadata + annotation markers emission

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

- 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

Statement annotation propagation in LLVM: correctness

An annotated region is preserved

1. Isolation conditions (can be relaxed, depending on the annotation's nature)
   ⇒ guaranteed by annotation markers + SSA barriers

Statement annotation propagation in LLVM: correctness

An annotated region is preserved

1. Isolation conditions (can be relaxed, depending on the annotation's nature)
   ⇒ guaranteed by annotation markers + SSA barriers
2. Optimizations of the annotated region
   - Default to the same level as other regions

Statement annotation propagation in LLVM: correctness

An annotated region is preserved

1. Isolation conditions (can be relaxed, depending on the annotation's nature)

   ⇒ guaranteed by annotation markers + SSA barriers

2. Optimizations of the annotated region
   - Default to the same level as other regions
   - Can be controlled by additional, annotation-specific constraints on the optimizations allowed within the annotated region

Statement annotation propagation in LLVM: correctness

An annotated region is preserved

1. Isolation conditions (can be relaxed, depending on the annotation's nature)

   ⇒ guaranteed by annotation markers + SSA barriers

2. Optimizations of the annotated region
   - Default to the same level as other regions
   - Can be controlled by additional, annotation-specific constraints on the optimizations allowed within the annotated region
     - no optimization

Statement annotation propagation in LLVM: correctness

An annotated region is preserved

1. Isolation conditions (can be relaxed, depending on the annotation's nature)
   $\Rightarrow$ guaranteed by annotation markers + SSA barriers

2. Optimizations of the annotated region
   - Default to the same level as other regions
   - Can be controlled by additional, annotation-specific constraints on the optimizations allowed within the annotated region
     - no optimization

     - less optimizing than the default level

Statement annotation propagation in LLVM: correctness

An annotated region is preserved

1. Isolation conditions (can be relaxed, depending on the annotation's nature)
   ⇒ guaranteed by annotation markers + SSA barriers
2. Optimizations of the annotated region
   - Default to the same level as other regions
   - Can be controlled by additional, annotation-specific constraints on the optimizations allowed within the annotated region
     - no optimization
       ⇒ guaranteed by inserting SSA barriers inside the annotated region
     - less optimizing than the default level

Statement annotation propagation in LLVM: correctness

An annotated region is preserved

1. Isolation conditions (can be relaxed, depending on the annotation's nature)

   ⇒ guaranteed by annotation markers + SSA barriers

2. Optimizations of the annotated region
   - Default to the same level as other regions
   - Can be controlled by additional, annotation-specific constraints on the optimizations allowed within the annotated region
     - no optimization

       ⇒ guaranteed by inserting SSA barriers inside the annotated region
     - less optimizing than the default level

⇒ ideal solution: per-region optimization mechanism

Validation: methodology

1. Annotating the source code
2. Compiling at LLVM *-O2*
3. Verifying manually in the binary

Validation: methodology

1. Annotating the source code
2. Compiling at LLVM *-O2*
3. Verifying manually in the binary

- DWARF section
    - Correct annotation DIE

Validation: methodology

1. Annotating the source code
2. Compiling at LLVM *-O2*
3. Verifying manually in the binary

- DWARF section
  - Correct annotation DIE
  - Correct debug info for annotated function or variable

Validation: methodology

1. Annotating the source code
2. Compiling at LLVM *-O2*
3. Verifying manually in the binary

- DWARF section
    - Correct annotation DIE
    - Correct debug info for annotated function or variable
    - Correct debug info for predicate variables

Validation: methodology

1. Annotating the source code
2. Compiling at LLVM *-O2*
3. Verifying manually in the binary

- DWARF section
  - Correct annotation DIE
  - Correct debug info for annotated function or variable
  - Correct debug info for predicate variables
- `.text` section: code generated for the annotated statement (respecting isolation + optimization conditions)

Validation: benchmarks and results

- Applications tested + annotations considered
  - VerifyPIN without protection: function behavior
  - VerifyPIN + Control Flow Integrity protection [LHB14]: protection
  - VerifyPIN + loop protection [Wit]: protection
  - First-order masked AES [HOM06]: secret + masked variables
  - RSA [DPP$^+$16]: random functions and variables
  - SHA [GRE$^+$01]: random functions and variables

Validation: benchmarks and results

- Applications tested + annotations considered
  - VerifyPIN without protection: function behavior
  - VerifyPIN + Control Flow Integrity protection [LHB14]: protection
  - VerifyPIN + loop protection [Wit]: protection
  - First-order masked AES [HOM06]: secret + masked variables
  - RSA [DPP$^+$16]: random functions and variables
  - SHA [GRE$^+$01]: random functions and variables
- Results
  - annotations found in DWARF section
  - BUT auto variable location ranges might be erroneous

    ⇒ patch submitted to fix the bug
  - protections preserved in machine code

Validation: preserving the protection

- Protection inserted at source level might be removed by optimizations
- Traditionally, 2 solutions to preserve the protections:
    - Compiling without optimization (`-O0`)
    - Using fragile programming tricks (e.g. `volatile`)
- Preliminary comparison: simulated for ARM Cortex-M3

Validation: preserving the protection

- Protection inserted at source level might be removed by optimizations
- Traditionally, 2 solutions to preserve the protections:
  - Compiling without optimization (`-O0`)
  - Using fragile programming tricks (e.g. `volatile`)
- Preliminary comparison: simulated for ARM Cortex-M3

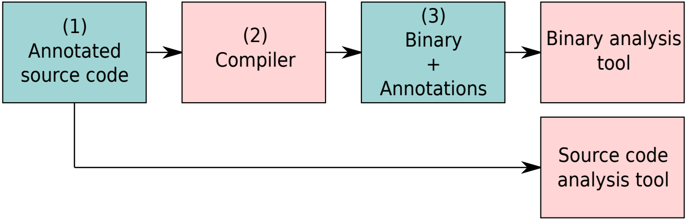
	VerifyPIN + loop protection		VerifyPIN + CFI protection	
	Protection	Exec. instr.	Protection	Exec. instr.
O0	✓	126	✓	1299
O2 + `volatile`	✓	89	✓	890
O2 + annotation	✓	62	✓	629
O2	✗	24	✗	130

- SSA barriers preserve the protections and region isolation while enabling heavy optimizations (`-O2`)

Outline

1. Introduction
2. Proposed solutions
3. Conclusion
4. References

Conclusion



1. ACSL-based source-level annotation language for wide range of properties
2. Mechanisms towards annotation-aware compilation framework
3. DWARF extension for binary-level annotation representation

Perspectives

- Evaluation of the annotation propagation impact on the compiler and the generated executable performance
- Automatic process to validate *annotation correctness*
- Per-region fine-grained optimization control

Perspectives

- Evaluation of the annotation propagation impact on the compiler and the generated executable performance
- Automatic process to validate *annotation correctness*
- Per-region fine-grained optimization control
- PhD graduation

Outline

References I

Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens.
Fissc: A fault injection and simulation secure collection.
pages 3–11, 09 2016.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown.
Mibench: A free, commercially representative embedded benchmark suite.
In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

Christoph Hillebold.
Compiler-assisted integrits against fault injection attacks.
Master's thesis, University of Technology, Graz, December 2014.

Christoph Herbst, Elisabeth Oswald, and Stefan Mangard.
An aes smart card implementation resistant to power analysis attacks.
In *Proceedings of the 4th International Conference on Applied Cryptography and Network Security*, ACNS'06, pages 239–252, Berlin, Heidelberg, 2006. Springer-Verlag.

References II

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.

Kedar S. Namjoshi and Lenore D. Zuck.
Witnessing program transformations.
In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis*, pages 304–323, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

Bernhard Schommer, Christoph Cullmann, Gernot Gebhard, Xavier Leroy, Michael Schmidt, and Simon Wegener.
Embedded Program Annotations for WCET Analysis.
In *WCET 2018: 18th International Workshop on Worst-Case Execution Time Analysis*, volume 63, Barcelona, Spain, July 2018. Dagstuhl Publishing.

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