IC637 Program Analysis

Lecture 9: Context Tunneling

Minseok Jeon

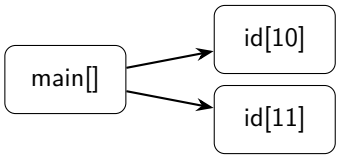
2025 Fall

Review: 1-Call-Site Sensitivity

- 1-**call-site sensitive** analysis can prove the castings are always safe.
- call-site sensitivity uses the call sites as context elements.

```
1  class A{}
2  class B{}
3  class C{
4    static Object id(Object v) {
5      return v;
6    }
7    void main(String[] args) {
8      A a = new A();//l1
9      B b = new B();//l2
10     A a2 = (A)id(a);//query1
11     B b2 = (B)id(b);//query2
12   }
13 }
```

1-**call-site** sensitive analysis



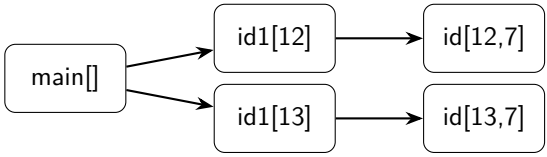
$$a[] \rightarrow \{l_1[]\}\ b[] \rightarrow \{l_2[]\}$$
$$v[10] \rightarrow \{l_1[]\}\ v[11] \rightarrow \{l_2[]\}$$
$$a2[] \rightarrow \{l_1[]\}\ b2[] \rightarrow \{l_2[]\}$$

Review: 2-Call-Site Sensitive Analysis

- conventional 2-**call-site sensitive** uses the caller methods' contexts.

```
1  class A{} class B{}
2  class C{
3    static Object id(Object v) {
4      return v;
5    }
6    Object id1(Object v) {
7      return this.id(v);
8    }
9    void main(String[] args) {
10     A a = new A();//l1
11     B b = new B();//l2
12     A a2 = (A)id1(a);//query1
13     B b2 = (B)id1(b);//query2
14   }
15 }
```

2-**call-site** sensitive analysis



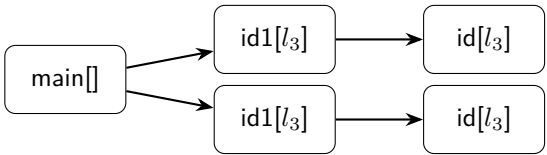
$$a[] \rightarrow \{l_1[]\}\ b[] \rightarrow \{l_2[]\}$$
$$v[10] \rightarrow \{l_1[]\}\ v[11] \rightarrow \{l_2[]\}$$
$$a2[] \rightarrow \{l_1[]\}\ b2[] \rightarrow \{l_2[]\}$$

Review: 1-Object Sensitivity

- Object sensitivity uses the receiver objects as context elements.

```
1  class A{} class B{}
2  class C{
3    Object id(Object v1) {
4      return v1;
5    }
6    Object id1(Object v2) {
7      return this.id(v2);
8    }
9  }
10 class D{
11   void main(String[] args) {
12     A a = new A();//l1
13     B b = new B();//l2
14     C c = new C();//l3
15     A a3 = (A)c.id1(a);//query3
16     B b3 = (B)c.id1(b);//query4
17   }
18 }
```

1-**object** sensitive analysis



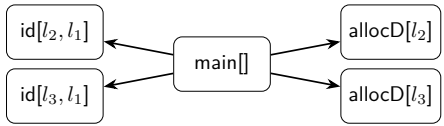
$$a[] \rightarrow \{l_1[]\}\ b[] \rightarrow \{l_2[]\}$$
$$v1[l_3] \rightarrow \{l_1[]\}\ v1[l_3] \rightarrow \{l_2[]\}$$
$$v2[l_3] \rightarrow \{l_1[]\}\ v2[l_3] \rightarrow \{l_2[]\}$$
$$a2[] \rightarrow \{l_1[]\}\ b2[] \rightarrow \{l_2[]\}$$

Review: 2-Object Sensitivity

- 2-**object sensitive** analysis uses the heap contexts of the receiver objects.

```
1  class C{
2    D allocD() {
3      return new D();}//l1
4  }
5  class D{
6    Object id(Object v) {
7      return v;}
8  }
9  class E{
10   void main(String[] args) {
11     C c1 = new C();//l2
12     C c2 = new C();//l3
13     D d1 = c1.allocD();
14     D d2 = c2.allocD();
15     A a = (A)d1.id(new A());//l4
16     B b = (B)d2.id(new B());//l5
17 }}
```

2-**object** sensitive analysis



$$c1[] \rightarrow \{l_2[]\}\ c2[] \rightarrow \{l_3[]\}$$
$$d1[] \rightarrow \{l_1[l_2]\}\ d2[] \rightarrow \{l_1[l_3]\}$$
$$v[l_2, l_1] \rightarrow \{l_4[]\}\ v[l_3, l_1] \rightarrow \{l_5[]\}$$
$$a[] \rightarrow \{l_4[]\}\ b[] \rightarrow \{l_5[]\}$$

Limitation of Conventional K-Context Sensitivity

- In the following example, conventional k-context sensitivity fails to prove the queries.

```
1  class A{} class B{}
2  class C{
3    Object id(int i, Object v) {
4      if (i > 0) {
5          return id(i-1,v);
6      } else {
7          return v;}}}
8  public class D {
9    void main(String[] args) {
10     int i = input();
11     A a = new A();//l1
12     B b = new B();//l2
13     C c = new C();//l3
14     A a2 = (A)c.id(i,a);//query1
15     B b2 = (B)c.id(i,b);//query2
16   }}
```

k-call-site sensitivity

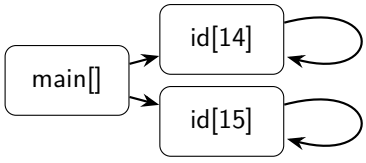
$$a[] \to \{l_1[]\}\ b[] \to \{l_2[]\}\ c[] \to \{l_3[]\}$$
$$v[14] \to \{l_1[]\}\ v[15] \to \{l_2[]\}$$
$$v[14,5] \to \{l_1[]\}\ v[15,5] \to \{l_2[]\}$$
$$\dots$$
$$v[5,\dots,5] \to \{l_1[], l_2[]\}$$
$$a2[] \to \{l_1[], l_2[]\}\ b2[] \to \{l_1[], l_2[]\}$$

Necessity of Context Tunneling

- With context tunneling, even 1-call-site sensitivity can prove the queries.

```
1  class A{} class B{}
2  class C{
3    Object id(int i, Object v) {
4      if (i > 0) {
5          return id(i-1,v);
6      } else {
7          return v;}}}
8  public class D {
9    void main(String[] args) {
10     int i = input();
11     A a = new A();//l1
12     B b = new B();//l2
13     C c = new C();//l3
14     A a2 = (A)c.id(i,a);//query1
15     B b2 = (B)c.id(i,b);//query2
16   }}
```

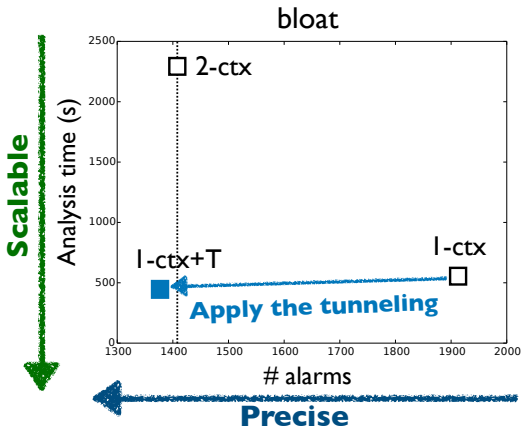
1-call-site sensitivity + tunneling
Important: $\{14, 15\}$



$$a[] \to \{l_1[]\}\ b[] \to \{l_2[]\}$$
$$v[14] \to \{l_1[]\}\ v[15] \to \{l_2[]\}$$
$$a2[] \to \{l_1[]\}\ b2[] \to \{l_2[]\}$$

Necessity of Context Tunneling

- Context tunneling can improve the performance of static analysis.



Analysis Rule (Call-Site Sensitivity with Tunneling)

- Rule for method calls (e.g., $VCall$) needs to be updated as follows:

$$\begin{aligned}
&\textbf{Merge}(\textbf{important}, heap, hctx, invo, callerCtx) = calleeCtx,\\
&VarPointsTo(this, calleeCtx, heap, hctx),\\
&CallGraph(invo, callerCtx, toMeth, calleeCtx) \leftarrow\\
&\quad VCall(base, sig, invo, inMeth), CallGraph(_, _, inMeth, callerCtx),\\
&\quad VarPointsTo(base, callerCtx, heap, hctx),\\
&\quad HeapType(heap, heapT), LookUp(heapT, sig, toMeth),\\
&\quad ThisVar(toMeth, this), \textbf{Importance}(\textbf{invo}, \textbf{important}).
\end{aligned}$$

- Key difference 1: $\textbf{Importance}(\textbf{invo}, \textbf{important})$
- Key difference 2: **Merge** takes another parameter $\textbf{important}$

- Define **Merge** for k-call-site sensitivity with context tunneling:

Analysis Rule

- **Merge** for 1-call-site sensitivity with context tunneling:

$$\textbf{Merge}(important, heap, hctx, invo, ctx) = \begin{cases} invo & \text{if } important = \textit{true} \\ ctx & \text{if } important = \textit{false} \end{cases}$$

- **Merge** for 2-call-site sensitivity with context tunneling:

$$\textbf{Merge}(important, heap, hctx, invo, ctx) = \begin{cases} \lceil ctx +\!\!+ invo \rceil_2 & \text{if } important = \textit{true} \\ \lceil ctx \rceil_2 & \text{if } important = \textit{false} \end{cases}$$

$$\ldots$$

- **Merge** for k-call-site sensitivity with context tunneling:

$$\textbf{Merge}(important, heap, hctx, invo, ctx) = \begin{cases} \lceil ctx +\!\!+ invo \rceil_k & \text{if } important = \textit{true} \\ \lceil ctx \rceil_k & \text{if } important = \textit{false} \end{cases}$$

Analysis Rule (Object Sensitivity with Tunneling)

- Rule for method calls (e.g., $VCall$) needs to be updated as follows:

$$\begin{aligned}
&\textbf{Merge}(\textbf{important}, heap, hctx, invo, callerCtx) = calleeCtx,\\
&VarPointsTo(this, calleeCtx, heap, hctx),\\
&CallGraph(invo, callerCtx, toMeth, calleeCtx) \leftarrow\\
&\quad VCall(base, sig, invo, inMeth), CallGraph(_, _, inMeth, callerCtx),\\
&\quad VarPointsTo(base, callerCtx, heap, hctx),\\
&\quad HeapType(heap, heapT), LookUp(heapT, sig, toMeth),\\
&\quad ThisVar(toMeth, this), \textbf{Importance}(\textbf{heap}, \textbf{important})
\end{aligned}$$

- Key difference 1: $\textbf{Importance}(\textbf{heap}, \textbf{important})$
- Key difference 2: **Merge** takes another parameter $\textbf{important}$

- Define **Merge** for k-object sensitivity with context tunneling:

Analysis Rule

- **Merge** for 1-object sensitivity with context tunneling:

$$\textbf{Merge}(important, heap, hctx, invo, ctx) = \begin{cases} hctx & \text{if } important = \textit{true} \\ heap & \text{if } important = \textit{false} \end{cases}$$

- **Merge** for 2-object sensitivity with context tunneling:

$$\textbf{Merge}(important, heap, hctx, invo, ctx) = \begin{cases} \lceil hctx +\!\!+ heap \rceil_2 & \text{if } important = \textit{true} \\ \lceil hctx \rceil_2 & \text{if } important = \textit{false} \end{cases}$$

$$\dots$$

- **Merge** for k-object sensitivity with context tunneling:

$$\textbf{Merge}(important, heap, hctx, invo, ctx) = \begin{cases} \lceil ctx +\!\!+ invo \rceil_k & \text{if } important = \textit{true} \\ \lceil ctx \rceil_k & \text{if } important = \textit{false} \end{cases}$$

Modeling Static Analyzer

- Given a program P, a parametric static analyzer F_P is modeled as follows.

$$F_P : \mathcal{A}_P \to \mathcal{P}(\mathbb{Q}_P) \times \mathbb{N}.$$

where $\mathcal{A}_P : \mathbb{C}_P \to \{true, false\}$ denotes mappings from each program component (e.g., call-site or object) to a boolean value, $\mathbb{Q}_P$ denotes sets of proven queries, and $\mathbb{N}$ denotes analysis costs.

- Given a mapping $\mathbf{a} \in \mathcal{A}$, so-called abstraction, we can generate input facts

$$Importance(component : C, importance : b)^1$$

For example, if a component c is mapped to $true$ in the abstraction $\mathbf{a}$, we generate an input fact $Importance(c, true)$.

[1] C: the set of program components, b: the set of boolean values.

Modeling Static Analyzer

- Given a program P, a parametric static analyzer F_P is modeled as follows.

$$F_P : \mathcal{A}_P \to \mathcal{P}(\mathbb{Q}_P) \times \mathbb{N}.$$

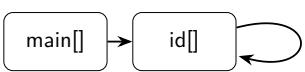
where $\mathcal{A}_P : \mathbb{C}_P \to \{true, false\}$ denotes mappings from each program component (e.g., call-site or object) to a boolean value, $\mathbb{Q}_P$ denotes sets of proven queries, and $\mathbb{N}$ denotes analysis costs.

- For convenience, we define two projection functions: $proved(F_P(a))$ returns the set of proven queries, and $cost(F_P(a))$ returns the analysis cost ($a \in \mathcal{A}_P$).

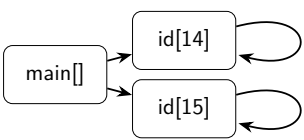
Non-Monotonicity of Analysis

- Unlike selective context sensitivity, classifying more program components as important (or unimportant) does not guarantee better (or worse) analysis precision.

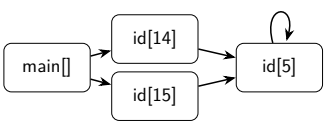
Important: $\{\}$



Important: $\{14, 15\}$



Important: $\{14, 15, 5\}$

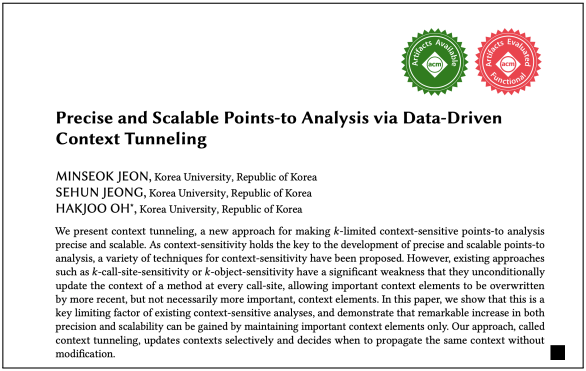


Open Challenge

- How can we find whether each program component is important or not?

Machine Learning-based Context Tunneling

- Machine learning-based approaches can be used:



Precise and Scalable Points-to Analysis via Data-Driven Context Tunneling

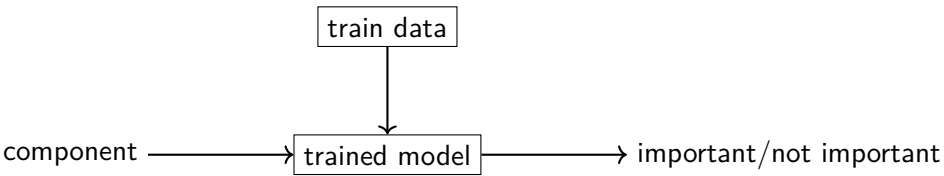
MINSEOK JEON, Korea University, Republic of Korea
SEHUN JEONG, Korea University, Republic of Korea
HAKJOO OH*, Korea University, Republic of Korea

We present context tunneling, a new approach for making k-limited context-sensitive points-to analysis precise and scalable. As context-sensitivity holds the key to the development of precise and scalable points-to analysis, a variety of techniques for context-sensitivity have been proposed. However, existing approaches such as k-call-site-sensitivity or k-object-sensitivity have a significant weakness that they unconditionally update the context of a method at every call-site, allowing important context elements to be overwritten by more recent, but not necessarily more important, context elements. In this paper, we show that this is a key limiting factor of existing context-sensitive analyses, and demonstrate that remarkable increase in both precision and scalability can be gained by maintaining important context elements only. Our approach, called context tunneling, updates contexts selectively and decides when to propagate the same context without modification.

- Idea: learn a machine learning model that maps each program component to a boolean value.

Machine Learning-based Context Tunneling

- Goal: learn a machine learning model that classifies whether each program component is important or not.



- Assumption, if a model is effective for the training data, it is also effective for the test data.

Disjunctive Model

- Disjunctive model: We adapt the disjunctive model to describe context tunneling technique (i.e., classifier) using boolean formulas.

Let $\mathbb{A} = \{a_1, a_2, \ldots, a_m\}$ be a set of atomic features, where each atomic feature is a predicate over program components:

$$a_i(P) : \mathbb{C}_P \to \{\mathit{true}, \mathit{false}\}.$$

Given a program P, a formula f represents a set of program components as follows:

$$\begin{array}{rcl} [\![\mathit{true}]\!]_P & = & \mathbb{C}_P \\ [\![\mathit{false}]\!]_P & = & \emptyset \\ [\![a_i]\!]_P & = & \{c \in [\![\mathit{true}]\!]_P \mid a_i(P)(c) = \mathit{true}\} \end{array} \qquad \begin{array}{rcl} [\![\neg f]\!]_P & = & [\![\mathit{true}]\!]_P \setminus [\![f]\!]_P \\ [\![f_1 \wedge f_2]\!]_P & = & [\![f_1]\!]_P \cap [\![f_2]\!]_P \\ [\![f_1 \vee f_2]\!]_P & = & [\![f_1]\!]_P \cup [\![f_2]\!]_P \end{array}$$

Disjunctive Model

- Let f be a tunneling formula. Then, the model $\mathcal{M}^{(f)}$ classifies each program component as follows:

$$\mathcal{M}^{(f)}(P) = \lambda c \in \mathbb{C}_P. \begin{cases} \textit{false} & \text{if } c \in [\![f]\!]_P \\ \textit{true} & \text{otherwise} \end{cases}$$

Learning Problem

Given a training set $\mathbf{P} = \{P_1, P_2, \ldots, P_n\}$, the learning problem is as follows:

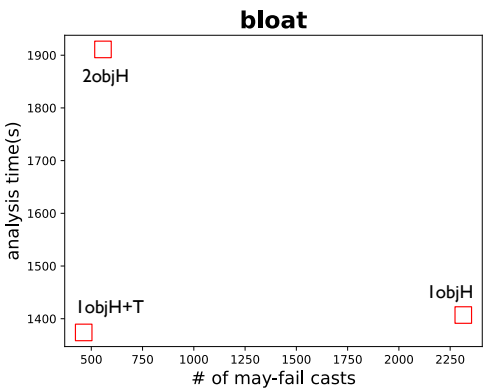
Given a codebase $\mathbf{P}$, find f that maximizes $\sum_{P \in \mathbf{P}} proved(F_P(\mathcal{M}^f(P)))$. (1)

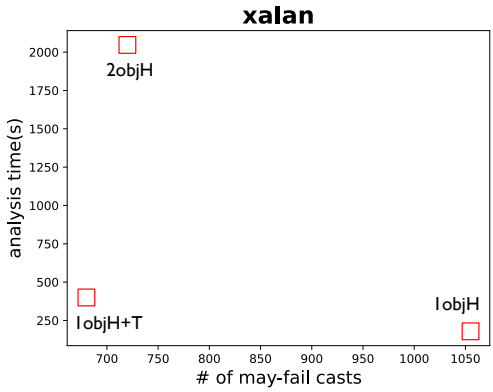
Overall Learning Procedure

- We can use standard learning algorithms to learn such f.

Evaluation Results

- Our context tunneling significantly enhances the precision.





Wrap-Up: Context Tunneling

- **Motivation:** Conventional k-context sensitivity usually loses important context elements and it degrades the precision.
- **Key Idea:** Context tunneling enables the analysis to keep important context elements for improving the precision.
- **Challenge:** Determining which program components are important
  - Non-monotonic: more important $\neq$ better precision
- **Benefits:** Significantly improves precision
  - 1-obj with tunneling is even more precise than the conventional 2-obj