Abstract Interpretation of Symbolic Execution for Information Flow Analysis

Reiner Hähnle

joint work with: Richard Bubel & Benjamin Weiß

Chalmers University of Technology, Gothenburg, Sweden

23 October 2008

Mobius

http://mobius.inria.fr

Work in Progress Warning

Overview

Mobius: Mobility, Ubiquity and Security

Proof-carrying code for Java on mobile devices

FP6 Integrated Project developing novel technologies for trustworthy global computing, using proof-carrying code to give users independent guarantees of the safety and security of Java applications for mobile phones and PDAs

- Innovative trust management, digital evidence of program behavior
- Static enforcement, checking code before it starts
- Modularity, building trusted applications from trusted components

Overview

Mobius: Mobility, Ubiquity and Security

Proof-carrying code for Java on mobile devices

FP6 Integrated Project developing novel technologies for trustworthy global computing, using proof-carrying code to give users independent guarantees of the safety and security of Java applications for mobile phones and PDAs

- Innovative trust management, digital evidence of program behavior
- Static enforcement, checking code before it starts
- Modularity, building trusted applications from trusted components

This talk

Integration of the two Mobius approaches for PCC basis

- Type Systems, type checking
- Program Logics, theorem proving

Type Systems vs. Program Logics

Type Systems

- Automatic, decidable
- Low precision
- Fixed precision
- Scaling to JAVA?

Program Logics

- Interactive systems
- High precision
- Formal specification
- JAVA CARD+ (byte/source)

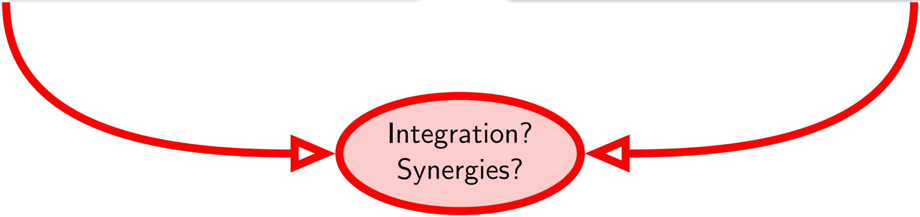
Type Systems vs. Program Logics

Type Systems

- Automatic, decidable
- Low precision
- Fixed precision
- Scaling to JAVA?

Program Logics

- Interactive systems
- High precision
- Formal specification
- JAVA CARD+ (byte/source)



Integration of a Type System into a Program Logic

Security properties often guaranteed by dedicated type systems

Non-Interference	Low (public) variables depend not on High (secret) ones
Declassification	Non-interference relativized to common knowledge

Integration of a Type System into a Program Logic

Security properties often guaranteed by dedicated type systems

Non-Interference: Low (public) variables depend not on High (secret) ones

Declassification: Non-interference relativized to common knowledge

Hähnle et al., Integration of a Security Type System into a Program Logic, TCS 402(2/3), pp172–189, 2008

- Translate Hunt-Sands flow-sensitive type system into program logic
- Type derivation = sequent calculus proof = symbolic execution
- Common semantics and calculus for type/deductive analysis

Integration of a Type System into a Program Logic

Security properties often guaranteed by dedicated type systems

Non-Interference: Low (public) variables depend not on High (secret) ones

Declassification: Non-interference relativized to common knowledge

Hähnle et al., Integration of a Security Type System into a Program Logic, TCS 402(2/3), pp172–189, 2008

- Translate Hunt-Sands flow-sensitive type system into program logic
- Type derivation = sequent calculus proof = symbolic execution
- Common semantics and calculus for type/deductive analysis

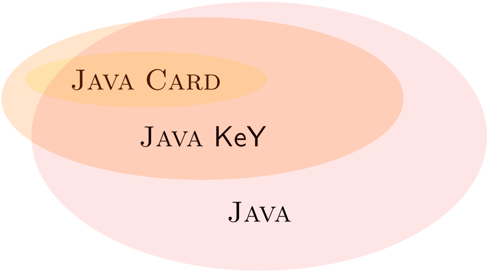
Achieved integration, but at price of some drawbacks

- Adaptation to other type systems remains non-trivial effort
- Toy language, incompatible to KeY's JAVA CARD program logic

Basis for Reasoning about Java Card Programs

KeY System: Java Program Logic & Verifier

- Sequent calculus for Java program logic
- Sequent calculus proof = symbolic execution + invariant rule
- Interactive prover with high degree of automation, e.g.:
  - Correctness of Mondex reference implementation (1 interaction)
  - Correctness of Java Card API reference implementation



Symbolic Execution in a Program Logic

Symbolic execution of conditional

$$\text{if}\ \frac{\Gamma, \mathtt{b} \doteq \mathbf{true} \Longrightarrow [\mathtt{p;\ \ rest}]\phi, \Delta \qquad \Gamma, \mathtt{b} \doteq \mathbf{false} \Longrightarrow [\mathtt{q;\ \ rest}]\phi, \Delta}{\Gamma \Longrightarrow [\mathbf{if}\ \mathtt{(b)\ \{\ p\ \}}\ \mathbf{else}\ \mathtt{\{\ q\ \};\ \ rest}]\phi, \Delta}$$

May require case split into different symbolic execution branches

Symbolic Execution in a Program Logic

Symbolic execution of conditional

$$\text{if}\ \frac{\Gamma, \mathtt{b} \doteq \mathbf{true} \Longrightarrow [\mathtt{p;\ rest}]\phi, \Delta \qquad \Gamma, \mathtt{b} \doteq \mathbf{false} \Longrightarrow [\mathtt{q;\ rest}]\phi, \Delta}{\Gamma \Longrightarrow [\mathbf{if}\ \mathtt{(b)\ \{\ p\ \}}\ \mathbf{else}\ \mathtt{\{\ q\ \};\ rest}]\phi, \Delta}$$

May require case split into different symbolic execution branches

Symbolic execution of loops:

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow [\mathbf{if}\ \mathtt{(b)\ \{\ p;}\ \mathbf{while}\ \mathtt{(b)\ p\};\ r}]\phi, \Delta}{\Gamma \Longrightarrow [\mathbf{while}\ \mathtt{(b)\ \{p\};\ r}]\phi, \Delta}$$

No termination if no fixed loop bound can be determined

The Challenge

Modular integration of (security) type system with (JAVA) program logic

The Challenge

Modular integration of (security) type system with (JAVA) program logic

```
x = (x % 2 * y)* z - 327;
```

Program logic:
precise symbolic execution

The Challenge

Modular integration of (security) type system with (JAVA) program logic

```
x = (x % 2 * y)* z - 327;
```

Program logic:
precise symbolic execution

```
x = (x, y, z);
```

Hunt-Sands type system viewed as bookkeeping of variable dependencies

The Challenge

Modular integration of (security) type system with (Java) program logic

```
x = (x % 2 * y)* z - 327;
```

↓ Abstraction

```
x = (x, y, z);
```

Program logic:
precise symbolic execution

Hunt-Sands type system viewed as
bookkeeping of variable dependencies

The Challenge

Modular integration of (security) type system with (Java) program logic

```
x = (x % 2 * y)* z - 327;
```

↓ Abstraction

```
x = (x, y, z);
```

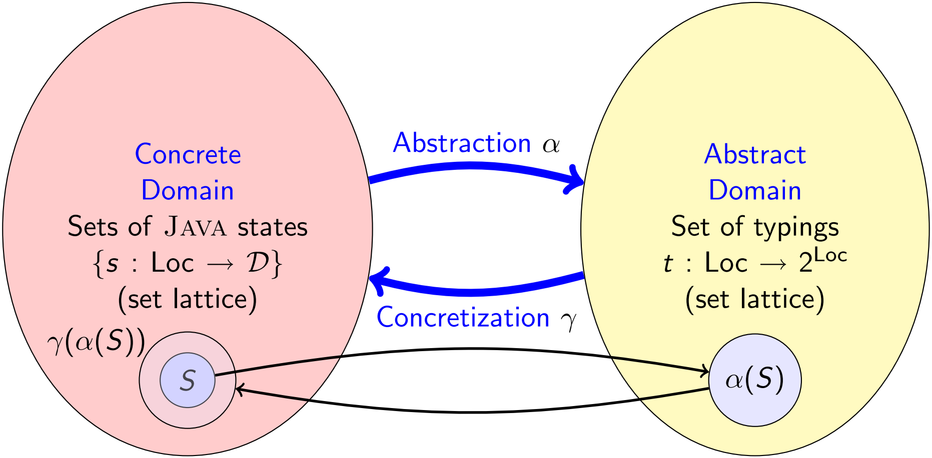
Program logic:
precise symbolic execution

Hunt-Sands type system viewed as bookkeeping of variable dependencies

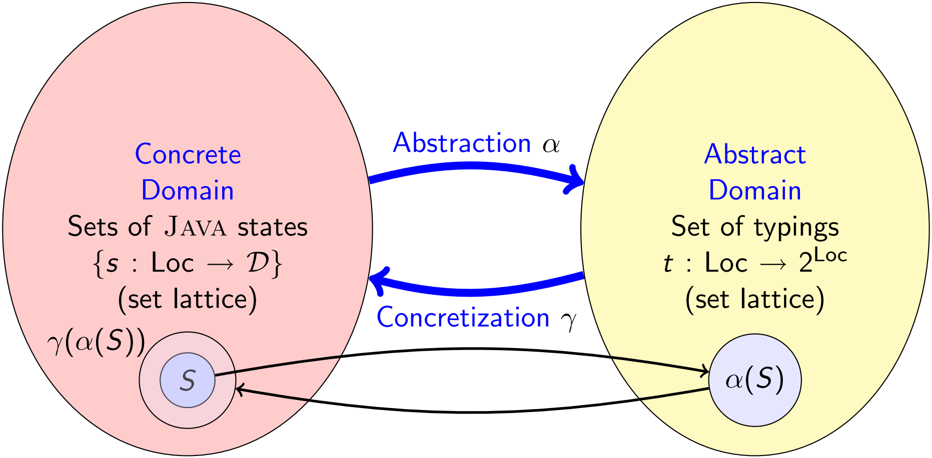
Our Idea

View type derivation as abstract interpretation of symbolic computation

Abstraction from Symbolic Execution



Abstraction from Symbolic Execution



Symbolic execution as concrete domain in abstract interpretation

Program Logic vs. Abstract Interpretation

Symbolic execution as concrete domain in abstract interpretation

	Program Logic	Abstract Interpretation
Program representation	abstract syntax tree	control flow graph
Merging execution paths	unusual, but possible	yes
Computation states	implicit	explicit
Value Computation	symbolic	concrete
Node semantics	single path	collecting
Loop treatment	invariant from user	fixed point
Termination	in general, no	if no ∞ chains

Program Logic vs. Abstract Interpretation

Symbolic execution as concrete domain in abstract interpretation

	Program Logic	Abstract Interpretation
Program representation	abstract syntax tree	control flow graph
Merging execution paths	unusual, but possible	yes
Computation states	implicit	explicit
Value Computation	symbolic	concrete
Node semantics	single path	collecting
Loop treatment	invariant from user	fixed point
Termination	in general, no	if no ∞ chains

Unwind control flow graph or permit sequent proof dag
(Leino InfProL'05, Schmitt & Weiß VERIFY'07)

Program Logic vs. Abstract Interpretation

Symbolic execution as concrete domain in abstract interpretation

	Program Logic	Abstract Interpretation
Program representation	abstract syntax tree	control flow graph
Merging execution paths	unusual, but possible	yes
Computation states	implicit	explicit
Value Computation	symbolic	concrete
Node semantics	single path	collecting
Loop treatment	invariant from user	fixed point
Termination	in general, no	if no ∞ chains

Identify symbolic expression (formula) with set of its models

Symbolic execution converges against collecting semantics

Program Logic vs. Abstract Interpretation

Symbolic execution as concrete domain in abstract interpretation

	Program Logic	Abstract Interpretation
Program representation	abstract syntax tree	control flow graph
Merging execution paths	unusual, but possible	yes
Computation states	implicit	explicit
Value Computation	symbolic	concrete
Node semantics	single path	collecting
Loop treatment	invariant from user	fixed point
Termination	in general, no	if no ∞ chains

Remaining issues: state representation and loop treatment

Explicit Computation States

Abstract Interpretation of JAVA is problematic

Computation on abstract domain using approximations of concrete ops:

$$\alpha(\mathtt{x} * \mathtt{y}) = \alpha(\mathtt{x})\,\alpha(*)\,\alpha(\mathtt{y}) = \alpha(\mathtt{x}) \cup \alpha(\mathtt{y})$$

- JAVA has dozens of operators (reducable to very few in program logic)
- Inter-procedurality
- Complex datatypes
- Complex operational semantics (dynamic dispatch, exceptions, . . .)

Explicit Computation States

Abstract Interpretation of JAVA is problematic

Computation on abstract domain using approximations of concrete ops:

$$\alpha(\texttt{x * y}) = \alpha(\texttt{x})\,\alpha(*)\,\alpha(\texttt{y}) = \alpha(\texttt{x}) \cup \alpha(\texttt{y})$$

- JAVA has dozens of operators (reducable to very few in program logic)
- Inter-procedurality
- Complex datatypes
- Complex operational semantics (dynamic dispatch, exceptions, ...)

Separate symbolic execution machinery from state representation

Explicit Computation States

Abstract Interpretation of JAVA is problematic

Computation on abstract domain using approximations of concrete ops:

$$\alpha(\mathtt{x\ *\ y}) = \alpha(\mathtt{x})\,\alpha(*)\,\alpha(\mathtt{y}) = \alpha(\mathtt{x}) \cup \alpha(\mathtt{y})$$

- JAVA has dozens of operators (reducable to very few in program logic)
- Inter-procedurality
- Complex datatypes
- Complex operational semantics (dynamic dispatch, exceptions, . . .)

Separate symbolic execution machinery from state representation

Needed: syntactic representation of symbolic computation states

- Describe symbolic state change in concise way
- Simple semantics, small set of operators
- Our solution: KeY updates (other options: Why, B gen. subst.,. . .)

KeY Updates

Definition (Update)

Let $\mathtt{l}$, $\mathtt{l_i}$ be JAVA program locations and $\mathtt{v}$, $\mathtt{v_i}$ first-order terms

- $\{\mathtt{l} := \mathtt{v}\}$ is an atomic update
- $\{\mathtt{l}_1 := \mathtt{v}_1\}\{\mathtt{l}_2 := \mathtt{v}_2\}$ is a sequential update
- $\{\mathtt{l}_1 := \mathtt{v}_1 | \cdots | \mathtt{l}_n := \mathtt{v}_n\}$ is a (bounded) parallel update (last-win)
- For $\mathtt{T}$ well-ordered type: quantified (parallel) update (minimal-win)

```
{\for T x; \if P; l := v}
```

KeY Updates

Definition (Update)

Let $\mathtt{l}$, $\mathtt{l_i}$ be Java program locations and $\mathtt{v}$, $\mathtt{v_i}$ first-order terms

- $\{\mathtt{l} := \mathtt{v}\}$ is an atomic update
- $\{\mathtt{l}_1 := \mathtt{v}_1\}\{\mathtt{l}_2 := \mathtt{v}_2\}$ is a sequential update
- $\{\mathtt{l}_1 := \mathtt{v}_1 | \cdots | \mathtt{l}_n := \mathtt{v}_n\}$ is a (bounded) parallel update (last-win)
- For $\mathtt{T}$ well-ordered type: quantified (parallel) update (minimal-win)

```
{\for T x; \if P; l := v}
```

Usage of updates

- KeY symbolic execution engine renders state change embodied by loop-free Java program in terms of updates
- Updates have normal form, are aggressively simplified

Update Abstraction

Update abstraction for non-interference analysis

Low (public, insecure) values can't depend on High (secret, secure) ones

- $\alpha(\{\mathtt{l} := \mathtt{v}\}) = \{\mathtt{l}^\alpha := \mathsf{Locations}(\mathtt{v})\}$
- Need to approximate semantics of update combinators (usually $\cup$)
- Semantics of abstract update $\{\mathtt{l}^\alpha := \{\mathtt{l}_1, \ldots \mathtt{l}_n\}\}$:
  Update of $\mathtt{l}$ with first-order term that depends at most on $\mathtt{l}_1, \ldots, \mathtt{l}_n$

Update Abstraction

Update abstraction for non-interference analysis

Low (public, insecure) values can't depend on High (secret, secure) ones

- $\alpha(\{\mathtt{l} := \mathtt{v}\}) = \{\mathtt{l}^\alpha := \mathsf{Locations}(\mathtt{v})\}$
- Need to approximate semantics of update combinators (usually $\cup$)
- Semantics of abstract update $\{\mathtt{l}^\alpha := \{\mathtt{l}_1, \ldots \mathtt{l}_n\}\}$:
  Update of $\mathtt{l}$ with first-order term that depends at most on $\mathtt{l}_1, \ldots, \mathtt{l}_n$

```
int h1, h2, t;
t=h1; h1=h2; h2=t;
t=t-h2;
```

Update Abstraction

Update abstraction for non-interference analysis

Low (public, insecure) values can't depend on High (secret, secure) ones

- $\alpha(\{\mathtt{l} := \mathtt{v}\}) = \{\mathtt{l}^\alpha := \mathsf{Locations}(\mathtt{v})\}$
- Need to approximate semantics of update combinators (usually $\cup$)
- Semantics of abstract update $\{\mathtt{l}^\alpha := \{\mathtt{l}_1, \ldots \mathtt{l}_n\}\}$:
  Update of $\mathtt{l}$ with first-order term that depends at most on $\mathtt{l}_1, \ldots, \mathtt{l}_n$

$$\{\mathtt{h1} := \mathtt{h2} \,|\, \mathtt{h2} := \mathtt{h1} \,|\, \mathtt{t} := 0\}$$

Symbolic ↑ Execution

```
int h1, h2, t;
t=h1; h1=h2; h2=t;
t=t-h2;
```

Update Abstraction

Update abstraction for non-interference analysis

Low (public, insecure) values can't depend on High (secret, secure) ones

- $\alpha(\{\mathtt{l} := \mathtt{v}\}) = \{\mathtt{l}^{\alpha} := \mathsf{Locations}(\mathtt{v})\}$
- Need to approximate semantics of update combinators (usually $\cup$)
- Semantics of abstract update $\{\mathtt{l}^{\alpha} := \{\mathtt{l}_1, \ldots \mathtt{l}_n\}\}$:
  Update of $\mathtt{l}$ with first-order term that depends at most on $\mathtt{l}_1, \ldots, \mathtt{l}_n$

$\{\mathtt{h1} := \mathtt{h2} \,|\, \mathtt{h2} := \mathtt{h1} \,|\, \mathtt{t} := 0\}$

Symbolic ↑ Execution

```
int h1, h2, t;
t=h1; h1=h2; h2=t;
t=t-h2;
```

Simplification before abstraction!

Update Abstraction

Update abstraction for non-interference analysis

Low (public, insecure) values can't depend on High (secret, secure) ones

- $\alpha(\{\mathtt{l} := \mathtt{v}\}) = \{\mathtt{l}^\alpha := \text{Locations}(\mathtt{v})\}$
- Need to approximate semantics of update combinators (usually $\cup$)
- Semantics of abstract update $\{\mathtt{l}^\alpha := \{\mathtt{l}_1, \ldots \mathtt{l}_n\}\}$:
  Update of $\mathtt{l}$ with first-order term that depends at most on $\mathtt{l}_1, \ldots, \mathtt{l}_n$

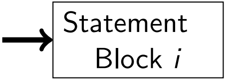
$\{\mathtt{h1} := \mathtt{h2} \,|\, \mathtt{h2} := \mathtt{h1} \,|\, \mathtt{t} := 0\}$ —Abstraction→ $\{\mathtt{h1}^\alpha := \{\mathtt{h2}\} \,|\, \mathtt{h2}^\alpha := \{\mathtt{h1}\} \,|\, \mathtt{t}^\alpha := \{\}\}$

Symbolic ↑ Execution

```
int h1, h2, t;
t=h1; h1=h2; h2=t;
t=t-h2;
```

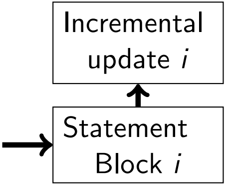
Simplification before abstraction!

Schema of Symbolic Execution with Update Abstraction



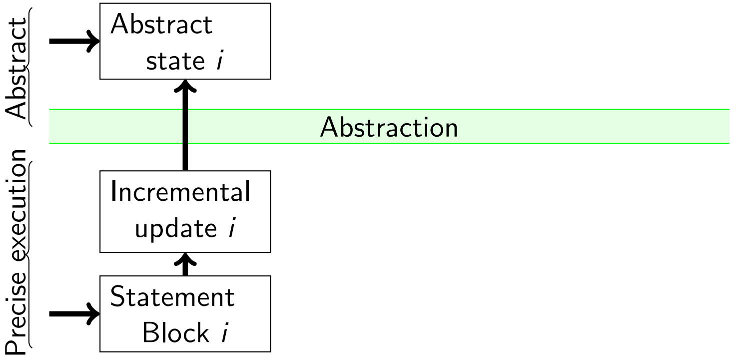
Symbolically execute JAVA statement in program logic — precise

Schema of Symbolic Execution with Update Abstraction



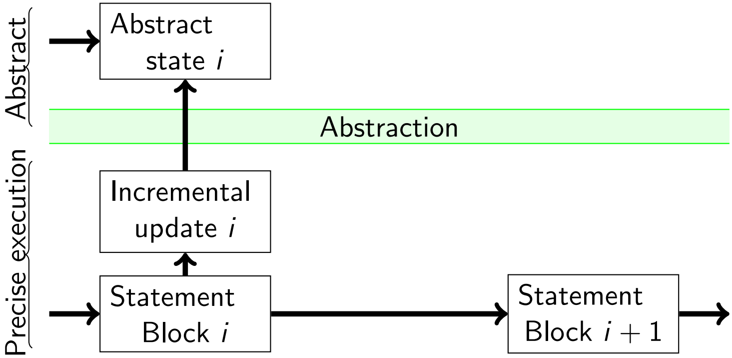
Compute resulting state change in terms of updates

Schema of Symbolic Execution with Update Abstraction



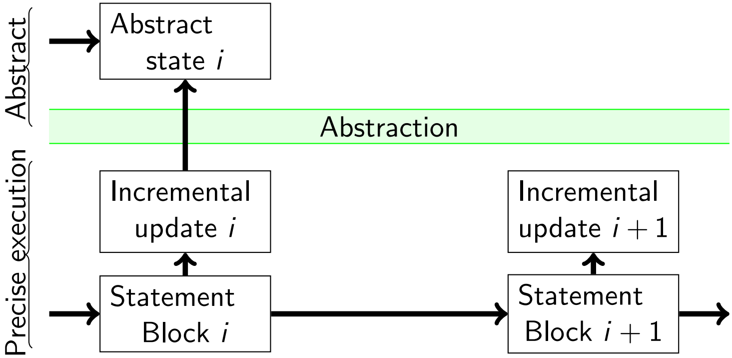
Abstraction of state update

Schema of Symbolic Execution with Update Abstraction

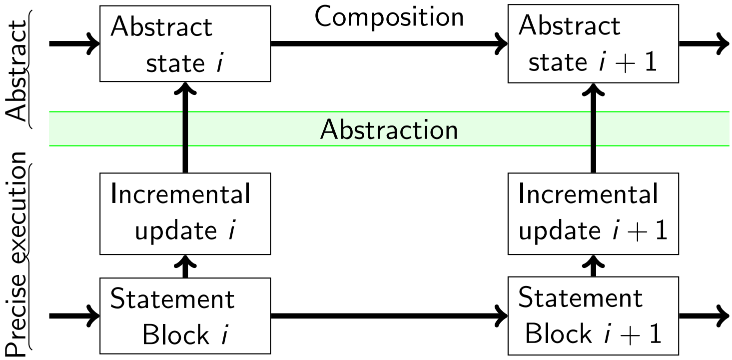


Continue symbolic execution of JAVA in program logic

Schema of Symbolic Execution with Update Abstraction



Compute incremental state change since Block i as an update



Abstract state update and compose with previous — abstract interpretation

Lazy Abstraction

Abstracting all locations is wasteful!

Lazy Abstraction

Abstracting all locations is wasteful!

- Precise values of static fields, initial values, system constants,. . .
  - Specify non-**null** assumptions, array bounds, . . .
  - Essential for termination-sensitive analyses, alias resolution

Lazy Abstraction

Abstracting all locations is wasteful!

- Precise values of static fields, initial values, system constants,. . .
  - Specify non-**null** assumptions, array bounds, . . .
  - Essential for termination-sensitive analyses, alias resolution
- Start execution in concrete domain for all locations
- Make program locations abstract one at a time by need
  - When encountering loops, user input, etc.
- When a location becomes abstract so do the locations depending on it

Lazy Abstraction

Abstracting all locations is wasteful!

- Precise values of static fields, initial values, system constants,. . .
  - Specify non-**null** assumptions, array bounds, . . .
  - Essential for termination-sensitive analyses, alias resolution
- Start execution in concrete domain for all locations
- Make program locations abstract one at a time by need
  - When encountering loops, user input, etc.
- When a location becomes abstract so do the locations depending on it

Example (Aliasing)

```
class C { public int a; public static N=2; }
C o = new C();
o.a=1; u.a=C.N; o.a=C.N; if (o.a!=u.a) l=h else h=l;
```

At first `l` is abstract.

Lazy Abstraction

Abstracting all locations is wasteful!

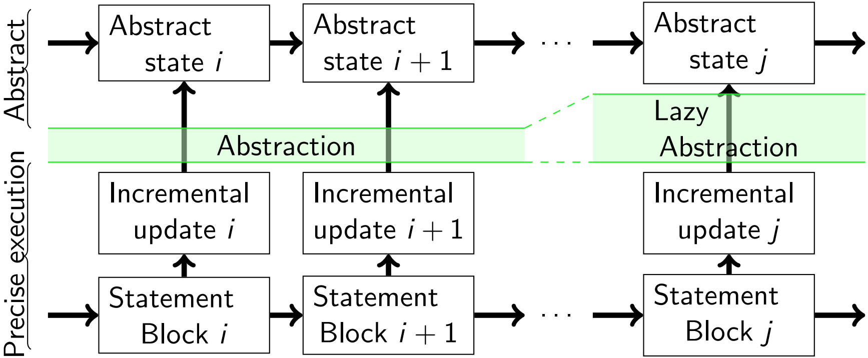
- Precise values of static fields, initial values, system constants,. . .
  - Specify non-**null** assumptions, array bounds, . . .
  - Essential for termination-sensitive analyses, alias resolution
- Start execution in concrete domain for all locations
- Make program locations abstract one at a time by need
  - When encountering loops, user input, etc.
- When a location becomes abstract so do the locations depending on it

Example (Aliasing)

```
class C { public int a; public static N=2; }
C o = new C();
o.a=1; u.a=C.N; o.a=C.N; if (o.a!=u.a) l=h else h=l;
```

At first l is abstract. Symbolic execution: $\{\mathtt{o.a} := 2 \,|\, \mathtt{u.a} := 2 \,|\, \mathtt{h} := \mathtt{l}\}$
Then abstraction only of h: $\{\mathtt{o.a} := 2 \,|\, \mathtt{u.a} := 2 \,|\, \mathtt{h}^{\alpha} := \{\mathtt{l}\}\}$

Schema of Lazy Abstraction



Search for Invariants Drives Abstraction

When encountering a loop . . .

```
while (guard) { body }
```

1. Save current abstract state in $\mathtt{s}_{old}$
2. Unwind loop once, execute `guard`, `body`, and obtain `s`
3. Compute point-wise $\sqcup$ on locations in $\mathtt{s}_{old}$, `s`:
   1. Different concrete values of $\mathtt{l}_{old}$, `l`: abstract both
   2. One of $\mathtt{l}_{old}$, `l` concrete: make it abstract
   3. Both $\mathtt{s}_{old}$, `s` abstract: $\sqcup = \cup$
4. Repeat until $\mathtt{s}_{old}$ equal to `s`
5. Conjoin `s` with `!guard`

- Terminates: finite number of locations, finite abstract domain
- Abstraction is driven by search for invariant

Proving Non-Interference

Low variables depend not on High variables

Proving Non-Interference

Low variables depend not on High variables

Formulating Non-Interference in Program Logic (Darvas et al. 2003)

Location $\mathtt{l}$ depends at most on locations $\mathtt{h}_1, \ldots, \mathtt{h}_n$ in program $\mathtt{p}$
Let $\mathtt{l}_1, \ldots, \mathtt{l}_m$ be remaining locations in $\mathtt{p}$ that $\mathtt{l}$ may depend on

$$\text{Validity of:} \qquad \forall \mathtt{l}_1, \ldots, \mathtt{l}_m . \exists r . \forall \mathtt{h}_1, \ldots, \mathtt{h}_n \, wp(\mathtt{p}, \mathtt{l} \doteq \mathtt{r})$$

Can be expressed in KeY's program logic (but also Coq, Isabelle, etc.)

Proving Non-Interference

Low variables depend not on High variables

Formulating Non-Interference in Program Logic (Darvas et al. 2003)

Location $\mathtt{l}$ depends at most on locations $\mathtt{h}_1, \ldots, \mathtt{h}_n$ in program $\mathtt{p}$

Let $\mathtt{l}_1, \ldots, \mathtt{l}_m$ be remaining locations in $\mathtt{p}$ that $\mathtt{l}$ may depend on

Validity of: $\forall \mathtt{l}_1, \ldots, \mathtt{l}_m. \exists r. \forall \mathtt{h}_1, \ldots, \mathtt{h}_n \, wp(\mathtt{p}, \mathtt{l} \doteq \mathtt{r})$

Can be expressed in KeY's program logic (but also Coq, Isabelle, etc.)

Soundness

1. Soundness of underlying symbolic execution of JAVA
2. Soundness of abstraction (from first-order terms to dependency sets)
3. Soundness of composition of abstract updates

Summary of Important Points

- Symbolic execution viewed as syntactic rendering of collecting semantics of concrete domain within AI
- Incremental computation of syntactic JAVA state representation (updates)
- Precise symbolic execution/first-order simplification before abstraction

  No need to handle complex language concepts at level of type system
  - Aliasing analysis
  - Exception handling
- Dynamic and lazy change of degree of abstraction during execution
- Direction of search for abstraction: precise $\Rightarrow$ abstract
  - Exploit information gained from precise symbolic execution