Java, Access Control, and Static Analysis

David Naumann

naumann@cs.stevens-tech.edu

Stevens Institute of Technology

Supported by NJCST and NSF

Joint work with Anindya Banerjee, Kansas State Univ.

Outline

- Information flow policies and their enforcement using access control
- Violations due to preventable bugs
- Recent progress in static analysis for assurance and performance
  - info flow checking
  - optimizing transformations
- Summary and research directions

Information flow policies

- *confidentiality*: receptionist can obtain patient phone number but not HIV status

Information flow policies

- *confidentiality*: receptionist can obtain patient phone number but not HIV status
- *accountability*: <u>N</u>urse can obtain patient status only with <u>D</u>octor's authorization, and N's ID is logged

Information flow policies

- *confidentiality*: receptionist can obtain patient phone number but not HIV status
- *accountability*: <u>N</u>urse can obtain patient status only with <u>D</u>octor's authorization, and N's ID is logged
- *access control* to *implement accountability*:
  - D grants read permission, `readP`, to N
  - N invokes `readP` and calls `readStatus`
  - `readStatus` checks for `readP`, writes log

  (simplified for exposition)

Information flow policies

- *confidentiality*: receptionist can obtain patient phone number but not HIV status
- *accountability*: <u>N</u>urse can obtain patient status only with <u>D</u>octor's authorization, and N's ID is logged
- *access control* to *implement accountability*:
  - D grants read permission, `readP`, to N
  - N invokes `readP` and calls `readStatus`
  - `readStatus` checks for `readP`, writes log

  (simplified for exposition)

Java and .NET CLR have support for implementing fine-grained application-specific policies.

Implementation sketch

```
void nurseTryRead(){
  doPriv readP { readStatus(self,"Joe"); }
}
void readStatus(String ID, String patientNam){
  checkPriv readP;
  doPriv writeLogP
    { writeLog(ID,patientNam); }
  doPriv sysReadP
    { print( sysReadStatus(patientNam) ); }
}
String sysReadStatus(String patientNam){
  checkPriv sysReadP;
  ... read file and return...
}
```

Violations due to program bugs

- Status appears on N’s screen because buffers not cleared — *info via direct data flow*.

Violations due to program bugs

- Status appears on N's screen because buffers not cleared — *info via direct data flow*.
- N deduces status from presence of "Schedule counselling?" option — *info via control flow*.

Violations due to program bugs

- Status appears on N's screen because buffers not cleared — *info via direct data flow*.
- N deduces status from presence of "Schedule counselling?" option — *info via control flow*.
- Access matrix initialized to grant `readP` to N; or fail to revoke after read — misuse of access control.

Violations due to program bugs

- Status appears on N's screen because buffers not cleared — *info via direct data flow*.
- N deduces status from presence of "Schedule counselling?" option — *info via control flow*.
- Access matrix initialized to grant `readP` to N; or fail to revoke after read — misuse of access control.
- `nurseTryRead` could write bogus log entries — here prevented using access control.

Violations due to program bugs

- Status appears on N's screen because buffers not cleared — *info via direct data flow*.
- N deduces status from presence of "Schedule counselling?" option — *info via control flow*.
- Access matrix initialized to grant `readP` to N; or fail to revoke after read — misuse of access control.
- `nurseTryRead` could write bogus log entries — here prevented using access control.
- A `readP` object is forged — prevent using *crypto*, *encapsulation* (private, immutable, final method).

Violations due to program bugs

- Status appears on N's screen because buffers not cleared — *info via direct data flow*.
- N deduces status from presence of "Schedule counselling?" option — *info via control flow*.
- Access matrix initialized to grant `readP` to N; or fail to revoke after read — misuse of access control.
- `nurseTryRead` could write bogus log entries — here prevented using access control.
- A `readP` object is forged — prevent using *crypto*, *encapsulation* (private, immutable, final method).
- A `readP` object is obtained through pointer leak — *data flow leads to access control failure*.

Violations due to program bugs

- Status appears on N's screen because buffers not cleared — *info via direct data flow*.
- N deduces status from presence of "Schedule counselling?" option — *info via control flow*.
- Access matrix initialized to grant `readP` to N; or fail to revoke after read — misuse of access control.
- `nurseTryRead` could write bogus log entries — here prevented using access control.
- A `readP` object is forged — prevent using *crypto*, *encapsulation* (private, immutable, final method).
- A `readP` object is obtained through pointer leak — *data flow leads to access control failure*.
- Bug in implementation of `checkPriv`.

Static analysis

- Ordinary type checking *assures basic invariants* of underlying security mechanisms: pointers not forged, no jumps to data, no calls to private methods, no overriding of final methods.

Static analysis

- Ordinary type checking *assures basic invariants* of underlying security mechanisms: pointers not forged, no jumps to data, no calls to private methods, no overriding of final methods.
- Extended types or model checking for info flow via data, control, pointer aliasing. . .
  Enforcement of info flow policy and *assurance for underlying mechanisms.*

Static analysis

- Ordinary type checking *assures basic invariants* of underlying security mechanisms: pointers not forged, no jumps to data, no calls to private methods, no overriding of final methods.
- Extended types or model checking for info flow via data, control, pointer aliasing. . .
  Enforcement of info flow policy and *assurance for underlying mechanisms.*
- Access control *invalidates standard optimizations*, and requires *new optimizations* for acceptable performance.

Secure flow by typing

- Lattice of levels (Recept $\leq$ Nurse $\leq$ Doc, L$\leq$H).

Secure flow by typing

- Lattice of levels (Recept $\leq$ Nurse $\leq$ Doc, L$\leq$H).
- Label input/output channels to express policy; annotate or infer labels for program variables etc.

Secure flow by typing

- Lattice of levels (Recept $\leq$ Nurse $\leq$ Doc, L$\leq$H).
- Label input/output channels to express policy; annotate or infer labels for program variables etc.
- Syntax-directed rules; extend typing to include labels.
  - `x:=E` — data type of `E` is $\subseteq$ type of `x`; level of `E` is $\leq$ level of `x`.
  - `if (E) S1 else S2` — if `E` is H then `S1,S2` do not assign to L variables.

Secure flow by typing

- Lattice of levels (Recept $\leq$ Nurse $\leq$ Doc, L$\leq$H).
- Label input/output channels to express policy; annotate or infer labels for program variables etc.
- Syntax-directed rules; extend typing to include labels.
  - `x:=E` — data type of `E` is $\subseteq$ type of `x`; level of `E` is $\leq$ level of `x`.
  - `if (E) S1 else S2` — if `E` is H then `S1,S2` do not assign to L variables.
- Volpano,Smith,Irvine '96 prove noninterference: if two initial states are identical on L variables then so are the corresponding final states. (A *simulation relation* is preserved [Abadi et al'99].)

Secure flow by typing - Java

- Myers'99 extends rules from simple imperative code to large fragment of Java, adds selective *declassification*, e.g., Doc to Nurse.

Secure flow by typing - Java

- Myers'99 extends rules from simple imperative code to large fragment of Java, adds selective *declassification*, e.g., Doc to Nurse.
- Banerjee&Naumann'02a prove noninterference; but simpler, less expressive rules. Key advances:
  - tractable semantic model
  - *pointer confinement* so that simulation is sound

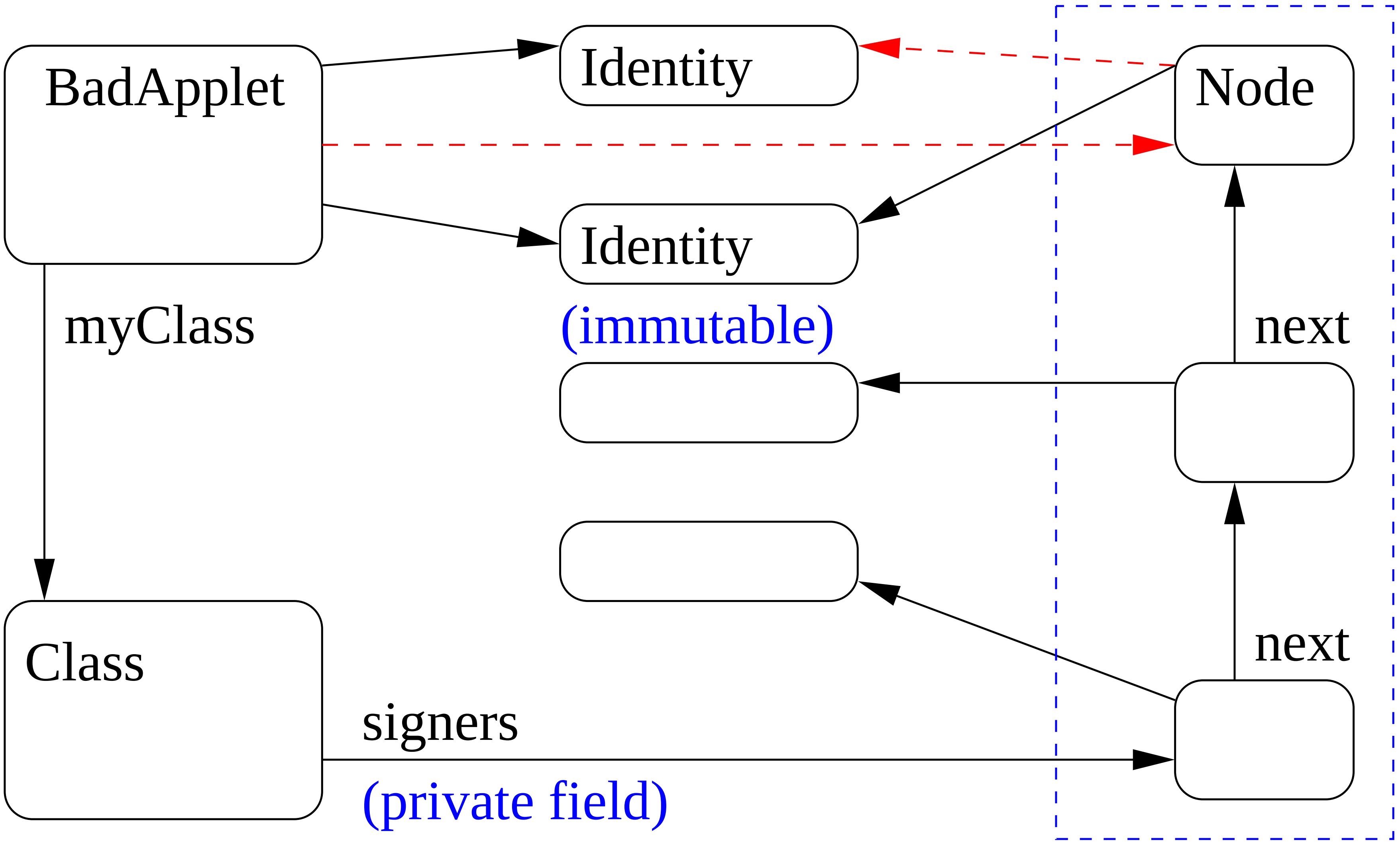
Secure flow by typing - Java

- Myers'99 extends rules from simple imperative code to large fragment of Java, adds selective *declassification*, e.g., Doc to Nurse.
- Banerjee&Naumann'02a prove noninterference; but simpler, less expressive rules. Key advances:
  - tractable semantic model
  - *pointer confinement* so that simulation is sound
- Simulation and data abstraction: *encapsulated* representations cannot be distinguished.
  - collection represented using array vs. tree
  - security context — marked stack vs. privilege set

checkPriv bug / simu. unsound

```
public class Class { // meta class
  private Identity[] signers; //crypto auth.
  public Identity[] getSigners() {
    return signers; }
...}
public class System {
  public Identity[] getKnownSigners(){...}
...}
class BadApplet {
  void bad() {
    Identity[] s = getSigners(); //leak
    s[0] = System.getKnownSigners()[0];
    doPriv {... something bad...} }
...}
```

Pointer confinement



Clarke,Noble&Potter'01, Smith,Walker&Morrisett'00, B&N'02, Vitek&Bokowski'01

Optimization

- Runtime overhead for stack inspection.
- *Call inlining not valid*.
- `doPriv readP { checkPriv readP; S }`

Optimization

- Runtime overhead for stack inspection.
- *Call inlining not valid*.
- `doPriv readP { checkPriv readP; S }`
- *Global analysis* for superfluous checks:

Can `S` throw exception if `readP` enabled? if not?
(Smith&Skalka'00, idealized language)

Optimization

- Runtime overhead for stack inspection.
- *Call inlining not valid*.
- `doPriv readP { checkPriv readP; S }`
- *Global analysis* for superfluous checks:

Can `S` throw exception if `readP` enabled? if not?
(Smith&Skalka'00, idealized language)

- Valid *transformations* (Fournet&Gordon '02,

applicative bisimulation for idealized language)

Optimization

- Runtime overhead for stack inspection.
- *Call inlining not valid*.
- `doPriv readP { checkPriv readP; S }`
- *Global analysis* for superfluous checks:

Can `S` throw exception if `readP` enabled? if not? (Smith&Skalka'00, idealized language)

- Valid *transformations* (Fournet&Gordon '02, applicative bisimulation for idealized language)
- Banerjee,Naumann'01, succinct proofs for idealized lang, *feasible validation of transformations for Java*.

Optimization

- Runtime overhead for stack inspection.
- *Call inlining not valid*.
- `doPriv readP { checkPriv readP; S }`
- *Global analysis* for superfluous checks:

Can `S` throw exception if `readP` enabled? if not? (Smith&Skalka'00, idealized language)

- Valid *transformations* (Fournet&Gordon '02,

applicative bisimulation for idealized language)

- Banerjee,Naumann'01, succinct proofs for idealized

lang, *feasible validation of transformations for Java*.

- Challenges include: privilege subsumption &

genericity; application-specific semantics of privileges; complexity of language and of policies. (Wallach,Appel&Felten'00)

Summary & ongoing research

- Access control used *for* flow policy, but control mechanisms *depend on* flow properties.
- (For specialists:) tractable models for complex real-world languages [B&N'02, Naumann'01].

Summary & ongoing research

- Access control used *for* flow policy, but control mechanisms *depend on* flow properties.
- (For specialists:) tractable models for complex real-world languages [B&N'02, Naumann'01].

⋆ Extend noninterference [B&N'02a] to declassification and rest of Java. Crypto types.

⋆ Flexible pointer confinement using access control and static analysis. Modifies clauses.

⋆ Modular analysis:
  - Behavioral subclassing
  - Proof-carrying code
  - Case study: extensible middleware for wireless

- Abadi,Banerjee,Heintze&Riecke: A core calc. of dependency, POPL 1999.
- Banerjee&Naumann: *A simple semantics and static analysis for Java security*, Stevens Tech Report 2001-1.
- Banerjee&Naumann: *Representation independence, confinement and access control*, POPL 2002.
- Banerjee&Naumann: *Secure information flow and pointer confinement in a Java-like language*, CSFW 2002, to appear.
- Clarke,Noble&Potter: *Simple ownership types for object containment*, ECOOP 2001.
- Fournet&Gordon: *Stack inspection: theory & variants*, POPL 2002.
- Myers: *JFlow: Practical mostly-static info. flow control*, POPL 1999.
- Naumann: *Soundness of data refinement for a higher order imper. lang.*, Theoretical Comp. Sci., 2001, to appear.
- Skalka&Smith: *Static enforcement of security with types*, ICFP 2000.
- Smith,Walker&Morrisett: *Alias types*, ESOP 2000.
- Vitek&Bokowski: *Confined types in Java*, Softw. Prac. Exper., 2001.
- Volpano,Smith&Irvine: *A sound type system for secure flow analysis*, Journal of Computer Security, 1996.