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**: Nurse can obtain patient status only with Doctor’s authorization, and N’s ID is logged
Information flow policies

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

- **accountability**: Nurse can obtain patient status only with Doctor’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**: Nurse can obtain patient status only with Doctor’s authorization, and N’s ID is logged

- **access control** to implement accountability:
  - D grants read permission, \( \text{readP} \), to N
  - N invokes \( \text{readP} \) and calls \( \text{readStatus} \)
  - \( \text{readStatus} \) checks for \( \text{readP} \), writes log
    (simplified for exposition)

Java and .NET CLR have support for implementing fine-grained application-specific policies.
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 $\text{readP}$ to N; or fail to revoke after read — misuse of access control.
- $\text{nurseTryRead}$ could write bogus log entries — here prevented using access control.
- A $\text{readP}$ object is forged — prevent using *crypto, encapsulation* (private, immutable, final method).
- A $\text{readP}$ object is obtained through pointer leak — *data flow leads to access control failure*. 

Naumann - NJITES Symposium, March 2002 – p.5/13
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 $\text{readP}$ to N; or fail to revoke after read — misuse of access control.
- $\text{nurseTryRead}$ could write bogus log entries — here prevented using access control.
- A $\text{readP}$ object is forged — prevent using *crypto, encapsulation* (private, immutable, final method).
- A $\text{readP}$ object is obtained through pointer leak — *data flow leads to access control failure*.
- Bug in implementation of $\text{checkPriv}$.
• 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 ≤ Nurse ≤ Doc, L≤H).

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

- 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.

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

- Lattice of levels (Recept ≤ Nurse ≤ Doc, L≤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 \( x \).
  - \( \text{if (} E \text{) S1 else S2} \) — 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

- Lattice of levels (Recept ≤ Nurse ≤ Doc, L ≤ 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 \( x \).
  - \( \text{if } (E) \ S_1 \text{ else } S_2 \) — if \( E \) is H then \( S_1, S_2 \)
    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 \textit{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
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 }`

Challenges include: privilege subsumption & genericity; application-specific semantics of privileges; complexity of language and of policies. (Wallach, Appel & Felten '00)
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)
• 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 \( \text{readP} \) 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: Representation independence, confinement and access control, POPL 2002.
• Clarke,Noble&Potter: Simple ownership types for object containment, ECOOP 2001.
• Skalka&Smith: Static enforcement of security with types, ICFP 2000.

Naumann - NJITES Symposium, March 2002 – p.13/13