Towards Provably Secure and Correct Systems

Avik Chaudhuri
Systems we rely on

Operating systems
Storage systems

Web application frameworks
Mobile device platforms

Clouds

These systems run our code.
What security and correctness guarantees do they provide?
Aim of my research

*Develop foundations of secure and correct systems*

**Analysis** of systems
(specification + verification)

**Construction** of systems
(design + implementation)

*Exploit programming languages as lens principles, techniques*
This talk

Overview of past and ongoing work on system analysis

- Storage systems (Plutus, OSD, PCFS)
- Operating systems (Windows Vista, Asbestos)
- Mobile device platforms (Android)
- Web application frameworks (Ruby on Rails)

Systems as programming languages

Program analyses for system guarantees

Similar ideas can guide system design
Analogy: Communication

Model cryptography with equations

\[ \text{decrypt} (\text{encrypt} (x, k), k) = x \]

*Programming language to describe protocols*

- create **new names** (keys)
- build terms with **function symbols** (hashing, encryption)
- communicate terms over **channels**
- compose **parallel processes**

[applied pi calculus]
Specification and Verification

**Adversary** *(arbitrary unspecified context)*

*Language semantics defines the power of the adversary*

**Specification** *(types, logical assertions)*

**Proof techniques** *(type systems, abstract interpretation)*

**Tools** *(ProVerif, F7)*

*Can verify protocols, their implementations, and their uses*

This approach can be extended to other systems!
Storage systems

- **Analysis of secure file sharing on untrusted storage**
  [with B. Blanchet: S&P’08]
  - various attacks, precise guarantees

- **Correct implementation of distributed access control**
  [with M. Abadi: FMSE’05, FORTE’06; FCS’08]
  - pitfalls, general design principles

- **Language support for proof-carrying authorization**
  [with D. Garg: ESORICS’09]
  - automatic proof management
Operating systems

- **Analysis of operating system security models**
  [with S. Rajamani and others: CCS’08]
  - decidability results, tool for system design

- **Type system for enforcing security on an operating system**
  [with S. Rajamani and others: PLAS’08]
  - formalization of design intentions and best practices
Mobile device platforms

- Security verification of mobile device applications
  
  [PLAS’09; ongoing work with J. Foster and others]
  
  - certified installation with security guarantees
Web application frameworks

- Security and correctness verification of web applications
  [with J. Foster and others: ASE’09, ongoing work]
  - elimination of various classes of attacks and bugs
Storage

Reductionist: Storage is, after all, communication
- write/read a file
  ≈ send/receive on a channel
- storage on untrusted disks
  ≈ communication over insecure networks

Should be able to leverage previous work on communication

Pragmatist: Careful! Why do we need dynamic access control?
Expect dynamic specifications (e.g., dynamic trust assumptions)
Storage on Untrusted Disks

Clients: some trusted, some untrusted

Server: untrusted
Storage on Untrusted Disks

**Files:** encrypted/signed

**Server:** untrusted

**Clients:** some trusted, some untrusted
Storage on Untrusted Disks

**Reader** (knows read key)

**Writer** (knows write key)

**Alice**

**Bob**

**Server:** untrusted

**Files:** encrypted/signed

**Clients:** some trusted, some untrusted
Storage on Untrusted Disks

**Reader** (knows read key)

**Writer** (knows write key)

**Clients:** some trusted, some untrusted

**Server:** untrusted

**Files:** encrypted/signed
Storage on Untrusted Disks

**Reader** (knows read key)

**Owner** (creates/distributes keys)

**Writer** (knows write key)

Files: encrypted/signed

Server: untrusted

Clients: some trusted, some untrusted
Storage on Untrusted Disks

Clients: some trusted, some untrusted

Server: untrusted

Files: encrypted/signed
Storage on Untrusted Disks

**Files:** encrypted/signed

**Server:** untrusted

**Clients:** some trusted, some untrusted

**Reader** (knows new read key)

**Owner** (creates/distributes new keys)

**Writer** (knows new write key)
Storage on Untrusted Disks

**Files:** encrypted/signed

**Server:** untrusted

**Clients:** some trusted, some untrusted
Storage on Untrusted Disks

Reader
(knows new read key)

Owner
(creates/distributes new keys)

Writer
(knows new write key)

Files: encrypted/signed

Server: untrusted

Clients: some trusted, some untrusted
Optimizations in Plutus

• Are files immediately re-secured with new keys?
  No! Files secured with new keys on subsequent writes
  “lazy revocation”

• How do readers decide which keys to use?
  Throw away old keys, derive from new keys as required
  “key rotation”

Does the protocol implement these optimizations correctly?
What are the security implications of these optimizations?
Specification and Verification

Automated security analysis of Plutus with **ProVerif**

- Applied pi calculus programs + expected properties
- Horn logic clauses + queries

**Results**

- **Weaker secrecy than claimed**
  - *writers can act for readers, secrets may eventually leak*

- **Serious attack on integrity** (clever exploit of subtle bug)
  - *adversary can collude with readers to become writers*
Key idea: Model Corruption

Specify code for each role in the protocol

- **owners** trusted
- **readers** trusted or corrupt
- **writers** trusted or corrupt

(trusted = follows protocol, corrupt = leaks keys)

Adversary unspecified, controls any run of the protocol
(e.g., chooses which principals to corrupt and when)

Specify security despite corruption

Security violation ➔ Precise lower bounds on corruption
Distributed Access Control

Networked Storage (e.g., OSD)

**Capability:** unforgeable/verifiable *authorization* certificate

*Revocation is tricky*

How do we specify and verify correctness of implementations?
Ideal Storage

Study preservation of trace properties, equivalences

Results

Various ways to break full abstraction
Nevertheless, correct implementations exist

General design principles:

Distributed implementation of stateful computations
Proof-Carrying Authorization

File system requires **authorization proofs**
*Tremendous burden on user*

**Solution**

**PCAL:** **Language/compiler support** for PCA systems

*user writes script, compiler manages proofs & instruments script*

Tricky! (e.g., proofs at compile time may be invalid at run time)
Various levels of trust (ordered)

Process
Location
System  Admin  User  Web

Protection mechanisms (e.g., labels for access control) intend to restrict information flow across levels of trust but often do not! (too restrictive)
Access Control for Integrity

“no write up”
“no execute down”

Browser security

Web data cannot overwrite User location
Web code cannot be run by Admin process

Browser functionality?

User needs to download data from Web (e.g., email attachment)
Admin needs to install code from Web (e.g., game application)
Dynamic Control of Labels

Windows Vista

**Admin can install code from Web**
- spawn untrusted process (execute less trusted code)

**User can save data from Web**
- trust untrusted location (verify trust by other means)
- read untrusted location (only overwrite untrusted data)
Dynamic Control of Labels

Windows Vista

**Admin** can install code from Web
- spawn untrusted process (execute less trusted code)

**User** can save data from Web
- trust untrusted location (verify trust by other means)
- read untrusted location (only overwrite untrusted data)

*Implicit design intentions, best practices?*
(Similar ideas in other operating systems, e.g., Asbestos)
System Designer: *What guarantees can the system provide?*
- **model behaviors** (dynamic semantics)
- **analyze restrictions** (static semantics)

**ProVerif**

✗ *undecidable query evaluation can be a problem!*

**Solution**

**Eon:** *dynamic logic programming*

✓ *expressive enough* with

✓ *decidable query evaluation*
Eon

\[
Eon = \text{Datalog} + \text{new} + \text{next} \\
+ \text{some syntactic restrictions} \\
(\text{unary dynamic predicates, monotonicity})
\]

\text{new: create constants initialized with some predicates} \\
\text{next: update predicates for such constants} \\
\text{Datalog: enforce constraints}

Query evaluation: reduction to Datalog query satisfiability \\
(new = existential quantifier, next = state transformer)
Automated Analysis with Eon

Experiments

- model behaviors (dynamic semantics)
- analyze restrictions (static semantics)

Results

Windows Vista

Attacks can be blamed on trusted processes
(interesting design principle)

Sound monitoring of trusted processes to eliminate attacks

Explicit design intentions, best practices!
Code Analysis for Security

Windows Vista
Attacks can be eliminated by restricting trusted code

Idea

*Enforce restrictions on code via a security type system*

**types** = security invariants

**soundness** = type preservation
Access Control + Security Types

*Programming language to describe code running on system*

- create **processes/locations** that are protected with labels
- update **labels** (access control)
- pack **code** as data (compilation)
- **read/write/execute** contents at locations

**Access control** encoded in **dynamic semantics**

**Security types** enforce **static semantics**

*Hybrid typechecking*  
(soundness, precision, optimization)
Access Control + Security Types

Security type $T^L$ (data of type $T$, trusted statically at level $L$)

Type Loc $T^L$ (location that contains data of security type $T^L$)

If loc : Loc $T^L$, then dynamic label of loc $\geq L$

By access control, levels $< L$ cannot
- write to loc
- control dynamic label of loc

By typechecking, ensure that levels $\geq L$ always
- write data that flows from levels $\geq L$ to loc
- maintain dynamic label of loc $\geq L$
Examples

```
let cmd.exe = new(... # System) in
let url = new(... # Web) in
let binIE = pack(let x = !url in exec x) in
let ie.exe = new(binIE # System) in

let virus.exe = new(... # Web) in
let url := virus.exe ↟
...
```

- **[System]** initialize `cmd.exe`
- `cmd.exe` contains code:
- reads `url` & executes contents
- **[User]** executes `ie.exe` at `Web`
- `ie.exe` contains code:
- overwrites `cmd.exe`
- **[Web]** `url` contains `virus.exe`

*Code typechecks*

**[Web] cannot write to `cmd.exe`**
Examples

```
let home = new(... # User) in
let url = new(... # Web) in
let setup.exe = new(... # Web) in
let binIE = pack(let z = !url in
    let x = !z in setup.exe := x) in
let ie.exe = new(binIE # System) in

[Admin] (...) ↑
    let _ = (Admin) setup.exe in
    exec setup.exe) ↑
[Web] (let binVirus = pack(home := ...) in
    let virus.exe = new(binVirus # Low) in
    url := virus.exe ↑
...)
```

- initialize **home**
- initialize **url**
- initialize **setup.exe**
- **ie.exe** contains code: reads **url** & copies contents to **setup.exe**
- trusts **setup.exe** & executes it
- executes **ie.exe** at **Web**
- **virus.exe** contains code: overwrites **home**
- **url** contains **virus.exe**

*Code does not typecheck*[ **Admin** can write to **home**]
Mobile Device Platforms

Android
- operating system + core apps
- SDK: Java APIs to develop new apps

Apps can share components
Sharing controlled statically (at install time)

Can Alice know whether Bob’s app is safe?
Can Bob convince Alice that the app is safe?
Certified Installation (PCC)

Bob constructs a proof that his app is safe
Alice verifies the proof before installing the app
(proofs may be implemented as certificates)

Requirements

- **Operational semantics** for apps running on Android
  *formal specifications of APIs provided by SDK*
  
  *Separate verification of APIs for efficiency*

- **Static analysis** of Android apps for safety
  *formalized as a security type system*
  
  *Soundness of analysis provides necessary proofs*
Android Applications

code may belong to other apps run with permissions of those apps

components
inherited classes, overridden methods

manifest
declare necessary permissions specify access controls for other apps

static

stack of windows

dynamic

pool of listeners
data
Security specifications derived from manifests of installed apps. Typechecking guarantees that access controls enforce security.

Suppose that reading my contacts list requires permission P.
Then only apps installed with permission P can know my contacts.

**Droid:** certified installer for Android apps (in progress)
Web Application Frameworks

Ruby on Rails
- Ruby: dynamically typed OO scripting language
- Rails: Ruby APIs for automating development of web apps

Models, Views, Controllers
- **models** connected to database tables
- **views** described by code-embedded markup
- **controllers** route requests to responses

Convention over Configuration
Static Analysis

Rails makes extensive use of **meta-programming** in Ruby

*Direct analysis is difficult*

**DRails:** translate to explicit Ruby code that is easier to analyze

Typechecking guarantees that method calls succeed
But almost everything is a method call in Ruby!

**Results**

Bugs that crash existing apps (confirmed by developers)
Simple typechecking cannot catch logical errors/security attacks
Advanced type systems difficult to build for Ruby

**Rubyx:** *symbolic verification* of Ruby scripts (in progress)
Key idea: *executable specifications*
Systems of the Future

*Influence of programming languages*

**Principles**
- Core, expressive abstractions
- Strong guarantees of security and correctness
- Implementations as refinements

**Techniques**
- Rich types/invariants for specification and verification
- Combinations of static and dynamic mechanisms
Systems of the Future

Influence of programming languages

Revisit academic programming languages
“strong foundations reduce complexity, spur inventions”

View system design as language design
“generalizations provide insight, set trends”

Opportunity to make lasting contributions