Self-protecting Systems

Nikolaos Michalakis
Motivation

Automating protection of content in an edge-side platform in untrusted environment.

Approach: detect malicious nodes via peer surveillance.

Peer surveillance is a form of self-protection.

What is the larger picture?

"Civilization advances by extending the number of important operations which we can perform without thinking about them."

- Alfred N. Whitehead
Self protection

Metaphysical approach
- self-awareness
- “smart” systems

Realistic approach
- System goal: system solves a problem (service, storage etc.).
- Protection goal: must achieve system goal under threat model.
- Solution: couple protection with functionality and automate.
- Eliminate threat.
Pessimistic self-protection

- Dependability of storage systems, hardware, critical services.
- Why self-protection?
  - Components faulty or untrusted.
- Integrate protection at performance cost
  - Secure, but now not enough...
Optimistic self-protection

Distributed service trend:
- scalable, available, fast
- business, collaboration

Increased complexity

Increased vulnerabilities

Large hacking computing base

Automate protection
- Admins too slow
- Manageability cost
Classification

- Replication (Practical BFT)
- Voting (LOCKSS)
- Reputation (EigenTrust)
- Detection + Reboot Recovery (SSM)
- Checkpoint + Roll-back Recovery (Rx)

Focus: (a) threat model, (b) protection coupling, (c) protection automation (d) key (knack) of workable solution
Criteria of evaluation:

- Protection: does solution work against threat model?
- Threat model: realistic, limited?
- Scalability: can it scale?
- Manageability: easy to manage?
- Cost: price for self-protection?
Practical BFT

Goal: provide correct service out of faulty, malicious components

Threat model: Byzantine faults.

Solution: replicated state machine with primary.

3f+1 replicas can handle f faulty nodes.

Algorithm: (1) Client sends request to primary. (2) Primary forwards to backups. (3) Primary+backups reply to client. (4) Client accepts if f+1 replies match.
Practical BFT

Assumptions: asynchronous, independent failures
Replicas mask failures
Non-faulty replicas execute client requests in same order
3 phase commit for serializability.
View change for liveness.
Coupling+automation via agreement.

Key: made practical by limiting effects of faulty primary

Performance: Byzantine NFS made practical by communication and crypto optimizations (only 3% slower).
Practical BFT

Protection: theoretical guarantee when less than \((n-1)/3\) nodes are Byzantine.

Threat Model: clear model but in practice errors accumulate.

Scalability: pessimistic, all non-faulty replicas must participate in 3-phase commit. Service partitioning?

Manageability: no intervention. Failure masked.

Cost: high. Multi-version programming, OS etc. to ensure independence.
Goal: Data preservation across the globe.

Threat model: natural damage at slow rate. Intentional damage through repair offers by malign nodes.

Lots Of Copies Keep Stuff Safe
- multiple copies at a set of peers
- use peers for repairs

Solution: randomly sampled opinion polls for safe repairs
LOCKSS data preservation

Opinion Poll
- pick random peers from personal list
- collect votes of data correctness.
- If votes disagree, repair local copy.
- alarm if something horrible happened.

Coupling: repair through opinion poll.

Automation: repair by download.

Key: evades adversary using randomness.

Performance: anti-goal. Preservation must maintain inertia.
LOCKSS data preservation

**Protection**: Preservation for up to 35% subversion. No theoretical guarantees. Only simulation.

**Threat model**: strong adversary active for 30 year span.

**Scalability**: polls instead of agreement.

**Manageability**: If alarm -> manual repair.

**Cost**: implementation cost to avoid common exploits
Eigentrust reputation

Goal: identify peers that serve inauthentic files in p2p sharing.

Threat model: malicious peers act rationally and form collectives.

Assumes a set of pretrusted peers known by all

Solution: Compute global reputation for each peer from local ratings to detect malicious peers.
Eigentrust reputation

Global Reputation computed by local trust values of others weighted by their global reputation.

Distributed+Secure: random score managers compute global trust for each peer.


Coupling: good/bad downloads fed back to local trust.

Key: “crowd is wiser than the individual”

Eigentrust reputation

**Protection:** Even with pre-trusted 30% corruption.
- Rational adversary can game pre-trusted.
- Suffers from one-time attacks.

**Threat model:** weak. In practice, malicious file replication.

**Scalability:** fully-distributed, fast convergence, robust.

**Manageability:** local ratings must be manually input.

**Cost:** update client software.
SSM: Manage Session state

- Goal: Simplify session state management in web services.
- Threat model: transient failures.
  - node failures are independent
  - no network partitions
  - reads followed by writes
  - otherwise catastrophic failure -> not dealt with
SSM: Manage Session state

Solution: detect failure + reboot + replicate to mask reboot.

R/W session state from replicas (bricks). Stub talks to bricks.

Pinpoint: Failure detection subsystem.

statistical model of “good behavior”

Coupling: send statistics to Pinpoint.

Automation: Pinpoint reboots brick.

State recovered from writes using client cookie.

Key: active (search) detection + cheap recovery.

Performance: No downtime. Failures are fully masked.
SSM session state

**Protection**: rebooting recovers. Good detection.

**Threat model**: independent brick failures not realistic (proven by DDS).

**Scalability**: more bricks, servers -> higher throughput.

**Manageability**: self-managed, but persistent faults need attention.

- Human intervention assisted by Pinpoint data.

**Cost**: hardware: need to buy multiple bricks for one server.
**Rx: bug recovery**

- **Goal:** increase service availability by fixing persistent failures.
- **Challenge:**
  - Deterministic software bugs persistent even after reboots.
  - Persistent after testing: memory, timing, user requests.
  - Bug-free code costs resources and money.
- **Threat model:** persistent failures caused by external factors (timing, mem management, clients).
**Rx: bug recovery**

Solution: (1) rollback to checkpoint before bug, (2) change the execution environment based on failure symptoms, and (3) re-execute buggy code in new environment.

Uses sensors to detect failure: OS exceptions, bug-specific.

Environment change examples: zero filling, change timing, drop malformed requests.

**Key:** non-determinism in the execution environment

Performance: Proxy masks failure, rollback from clients. Outperforms reboots by an order of magnitude.
Rx: bug recovery

- **Protection**: recovers correctly.
  - Suffers from false negatives (sensors).

- **Threat model**: realistic, correct assumptions about persistent faults.

- **Scalability**: Each node independent. Can scale under functional homogeneity (Porcupine).

- **Manageability**: automates service recovery.
  - Reports bugs.
  - Worst case reboot.

- **Cost**: software installation, proxy configuration.
What went wrong

- Wrong threat model.
  - Tricky: Eigentrust
  - Unrealistic: SSM

- Masking failures from system
  - causes long-term failure.
  - PBFT vs LOCKSS (corrupt nodes)
  - SSM vs Rx (bugs)
What went well

- **Recovery:**
  - repair or lose. Cannot bound failures over time.
  - PBFT alone does not help.
  - SSM recovers aggressively.

- **Redundancy:**
  - Survives failures. Availability. (LOCKSS)
  - Detects attacks. (SSM+Pinpoint)

- **History:**
  - remember healthy states. (Rx)
  - Detect attacks. (SSM)
What went well

- Non-determinism: have many paths to get somewhere.
  - key to survive failures.
  - LOCKSS random polls.
  - Randomness->non-determinism. Not other way around.
  - Rx environment changes.
  - Active non-determinism: adversary is part of the equation for choosing a path.
  - PBFT appears deterministic, but not really.
  - Multiple version programming...
What we need

- detection + resolution -> automation -> manageability
- More comprehensive failure/attack detection
  - Rx: false negatives.
  - semantic attack detection.
- Better problem resolution
  - improved assistance (Rx, SSM)
  - But humans are still in the loop (LOCKSS, Eigentrust)
What we need

Composition of solutions
- solutions still domain specific.
- non-determinism and recovery are key design principles.
- Are they both always possible?
- What if SSM detection + Rx recovery + LOCKSS replication?
- Need to find common ground
- Need to prove protection guarantees
Outline

- Background
- Classification
  - class representative
  - evaluation
- Systems comparison
- Future directions
- Conclusion
Future directions

- Open systems: exploit scalability in untrusted environments.
- Complex Composition: self-protection as a building block.
- Protection guarantees under non-deterministic model.
- Can we model protection + threat model?
- Threat sensing: Better failure + attack sensors
Conclusions

- Demand for scalability, availability, manageability
- Self-protection simplifies management. Helps evolution.
- Threat model can be tricky.
- Recovery is essential for the long-term.
- Non-determinism provides advantage over adversary.
- Need solution convergence to build SuperSystem!
THE END