Sluice
Secure Dissemination of Code Updates in Sensor Networks

Patrick E. Lanigan
Spring 2006
Roadmap

Background
1. Wireless Sensor Networks
2. Network Reprogramming
3. Example Protocol
4. Security Issues

Sluice
1. Design Goals
2. Approach
3. Implementation & Evaluation
4. Future Directions
Wireless Sensor Networks: Overview

Large scale
- 100’s - 1000’s of nodes
- Multi-hop network

Resource constrained
- Limited energy
- Limited computational power

Deeply embedded
- Environmental (low-impact monitoring)
- Infrastructural (poured into concrete)
- Industrial (enclosed within pipelines)

Low cost
- < $100 per node
Wireless Sensor Networks: TinyOS

Provides basic system services sensor networks
- Single threaded, event-based execution
- FIFO task scheduling
- Hardware abstraction
  - Radio stack (sending / receiving packets)
  - Timers
  - Sensors (reading)
  - Flash memory (reading / writing data)

Monolithic application model
- System code & user code compiled together into a single binary - the application image
Wireless Sensor Networks: Maintenance

Software deployment
- Fix bugs, add new functionality
- Update must be deployed to all of the nodes

Traditional method: manual, in-situ updating
- Updates are flashed over a hardwired back-channel
- Requires a physical connection to the node

Often infeasible following network deployment!
- Scale - Who wants to collect 1000’s of nodes?
- Access - Can you even get to the nodes?
Network Reprogramming

Update an entire network of nodes *en masse*

- Send application updates over-the-air
- Epidemic propagation across multi-hop networks

Many existing protocols

- MOAP, Deluge, MNP, Infuse, Sprinkler, etc.

Existing protocols have mainly focused on…

- Efficiency
  - Limit resource usage
  - Reduce propagation time
- Reliability
  - Robust against packet loss
  - Tolerate nodes coming and going
Epidemic Propagation (1)
For example… Deluge

Most widely available protocol
- Developed at UC Berkeley
- Distributed as part of TinyOS
- Complete implementation, includes host-based administration tools

Provides eventual consistency
- Eventually all nodes will receive every byte of the update
- Places no bounds on end-to-end dissemination latency

Leverages various efficiency mechanisms
- Message suppression reduces control-traffic overhead
- Pipelining to reduce end-to-end dissemination time
- Dynamic advertisement period
  - longer period during steady-state operation
  - shorter period during upgrade
Deluge: Data Preparation

CRC block

application data

padding

1104 bytes

pages

48 packets/page

......
Deluge: Epidemic Propagation

Three-phase handshake

- **Advertise**
  - Nodes broadcast version numbers
  - Allows nodes to learn of availability of new updates
- **Request**
  - Nodes request packets from a needed page
  - Missing packets from a particular page are indicated by a bitmap
- **Data**
  - Packets from a requested page are broadcast in round-robin order

Pipelining

- Nodes can forward pages before the entire update has been received
- Pages must be received **sequentially**
What About Security???

No authentication mechanisms!
- No restrictions on the source of an update!
- Assumes correct / trusted operation from all nodes
- Anyone can advertise high version numbers

If you can inject packets, you can inject code
- Battery drain
- Denial-of-service
- Complete control

Local weaknesses affect entire network
- Network reprogramming protocols are epidemic in nature
- All the attacker needs is a single node…
- The protocol does all the work for you!
- Node compromise is a serious liability!
Epidemic Propagation (2)

But what if this node is compromised?
Possible Solutions: Symmetric Crypto

Shared key schemes (i.e. TinySec)
- Protect against packet injection at the link layer
- All nodes share a key (or keys)
  - MAC for access control / integrity
  - Encryption for confidentiality
- Susceptible to node compromise!

Pair-wise key schemes
- Neighboring nodes share a key (or keys) with some probability
- Limits eavesdropping / injection to broadcast neighborhood
- Ineffective with epidemic protocols!
- Still susceptible to node compromise!
Possible Solutions: Authenticated Bcast

Authenticated broadcast schemes ($\mu$TESLA)

- Provides authentication using symmetric primitives
- Delayed key disclosure provides asymmetry
  - Sender keeps symmetric key secret for a period of time
  - Receivers buffer packets until key disclosure
- Security depends on loose time synchronization
  - Once a key is disclosed, anyone can use it
  - Receivers need to know whether a key is “safe”
- Timing assumptions are difficult to guarantee
  - Multi-hop networks
  - Eventual consistency
Possible Solutions: Asymmetric Crypto

Public keys? In sensor networks? No way!
- Asymmetric crypto is **expensive**
- Long thought impractical for sensor networks
- However, recent work (Sizzle, TinyECC, etc) has shown it is **feasible**

OK, sign the entire update then.
- Verification has to wait until completion
- Pipelining? Either turn it off or risk propagating bogus data

So just sign every packet / page.
- It’s still very expensive! (order of seconds for verification)
- Would like to limit expensive operations

Let’s try something smarter…
Roadmap

Overview & Background
1. Wireless Sensor Networks
2. Network Reprogramming
3. Example Protocol
4. Security Issues

Sluice
1. Design Goals
2. Approach
3. Implementation & Evaluation
4. Future Directions
Sluice

sluice (n.) a sliding gate or other device for controlling the flow of water

Authenticity
- Updates should be verified as coming from a trusted source

Integrity
- Any modifications to the update should be detectable

Progressiveness
- Verification should take place in an ongoing fashion

Correctness
- No unverified data should ever be installed or propagated

Compatibility
- The mechanism should require minimal changes to the underlying protocol

Resource sensitivity
- Minimize overhead relative to underlying protocol

Sluice aims for the progressive, resource sensitive verification of code updates in sensor networks.
Sluice

Threat Model

- Insecure wireless medium
  - Packets can be injected, modified, corrupted, captured, etc.
- Base station is trusted
  - Physically hardened against compromise
  - Associated public / private key-pair
- Sensor nodes are untrusted
  - Might exhibit arbitrary behavior
  - Might become compromised
  - Know base station’s public key
- Unbounded number of compromised / malicious nodes

Other assumptions

- Reliable, in-order page delivery
- No need for time synchronization
Sluice

Hash-chain construction
- Hash each page
- Store the digest in the previous page
- Digitally sign the first page payload & digest
- Store the signature in the first page

Progressive verification
- The signature verifies the payload and hash of \( p_0 \)
- Each page’s hash verifies the subsequent page’s payload and hash
Sluice: Implementation

Cryptographic functions

- All crypto libraries are **off-the-shelf** and **unoptimized**
- TinyECC provides ECDSA primitives for sensor nodes
- BouncyCastle JCE provides ECDSA primitives for the base station
- Hash chain uses **full** 160-bit SHA1 digests

Modifications to Deluge

- Reserved space in pages
  - 20 byte hash in each page (except the last)
  - 44 byte signature in the first page
- Buffer pages in RAM until verified
- Host-based Java tools construct hash-chain
Sluice: Informal Evaluation

Authenticity & Integrity
- Pre-image resistance of SHA1
- Unforgeability of ECDSA

Progressiveness
- Hash-chain construction allows pages to be verified as they are received

Completeness
- Pages are not forwarded until they are verified
- Data is not written to RAM until verified

Compatibility
- Uses default page and packet sizes
- No need to disable pipelining or other mechanisms

Efficiency
- Cost of a hash amortized over an entire page
- Cost of a single signature amortized over an entire update
Sluice: Experimental Evaluation

Metrics
- Spatial overhead
  - Memory overhead (ROM and RAM)
  - Transmission overhead (bytes sent over the wire)
- Temporal overhead
  - End-to-end dissemination latency

Testbed
- 12 Tmote Sky nodes (TinyOS)
- Pentium 4 base-station (Linux)
- Single hop topology
- USB back-channel (debugging, data collection)
Sluice: Empirical Evaluation

Spatial overhead
- Memory: 9 Kb ROM, 2 Kb RAM
- Transmission
  > Theoretically, a maximum of $S_\sigma + n \times S_h$
  > In practice, a multiple the page size
  > Overhead can be masked by using existing padding bytes

Temporal overhead
- Significant overhead relative to Deluge
  > Drops as update size increases
  > **Absolute overhead remains generally constant**
- Unoptimized cryptographic functions take a long time!
  > 30 - 35 seconds for signature verification
  > 200 ms for hash computation
- Reduce overhead by optimizing crypto routines
The INI is a cooperative endeavor of:

- Electrical and Computer Engineering
- School of Computer Science
- Graduate School of Industrial Administration
- Heinz School of Public Policy
Sluice: Simulation

TinyOS simulator (TOSSIM)
- Modified to simulate long-running cryptographic operations
- N x N grid topology, 15m spacing
- 33-page updates (~35Kb)

Scalability with network size...
- Sluice tracks Deluge
- Sluice has steeper slope
- Sluice tends to lag behind Deluge

Conclusion
- Overhead mostly due to long-running cryptography
The INI is a cooperative endeavor of:

- Electrical and Computer Engineering
- School of Computer Science
- Graduate School of Industrial Administration
- Heinz School of Public Policy
Related Work

Dutta et. al., “Securing the Deluge Network Programming System” (to appear IPSN ‘06)
• Recently and independently proposed a similar approach
• Primarily differ in granularity of chaining
  > Constructed over packets, as opposed to pages
  > Allows malicious packets to be identified
  > Pages do not have to be buffered in RAM
• Truncated SHA1 digest (64-bits) to fit in a single packet
• Imposes much higher overhead
  > 384 bytes per page versus 20 bytes per page for Sluice
  > 48 hash operations per page instead of 1
Sluice: Looking Ahead…

More efficient cryptographic routines
• Not an exercise in code optimization!
• Sluice could benefit from optimized cryptographic libraries
  > signature verification reported in a few seconds or less

Reduce cryptographic operations
• No distinction between benign and malicious errors
• Page-level CRC checks could serve as a hint
  > CRC passes, continue to check signature / hash
  > CRC fails, don’t bother

Expanded threat model
• Address denial-of-service, battery drain, replay attacks

Empirical comparison versus alternatives
Publications

Sluice: Secure Dissemination of Code Updates in Sensor Networks

Poster Abstract: Secure Dissemination of Code Updates in Sensor Networks

Disseminating Code Updates in Sensor Networks: Survey of Protocols and Security Issues
Summary

Network reprogramming is a valuable, but sensitive service
Existing protocols have not addressed security issues
Sluice is a feasible way to address such concerns
  • Prevents propagation & installation of untrusted code
  • Requires little modification to existing protocols
  • Compatible with standard efficiency mechanisms (pipelining, etc)
  • Uses standard cryptographic primitives and modes of operation
  • Is not tied to any particular digital signature scheme
  • A simple solution to a serious problem

Questions?

http://www.ece.cmu.edu/~planigan/research