CS 5432:
Proactive Obfuscation
and Moving Target Defenses

Fred B. Schneider
Samuel B Eckert Professor of Computer Science

Department of Computer Science
Cornell University
Ithaca, New York 14853
U.S.A.
Fault-tolerance by Replication

Implements:
- Integrity
- Availability

The basic recipe ...

- Servers are deterministic state machines. Clients make requests.
- Server replicas run on distinct hosts.
- Servers fail independently.
- $2t+1$ servers tolerate $t$ Byzantine
Attack-tolerance by Replication?

Assume \( n = 2t + 1 \) server replicas:

- Server **failures** are independent.
  \[
  \text{Prob}[t + 1 \text{ servers fail}] = \binom{2t+1}{t+1} \text{Prob}[\text{one server fails}]^{t+1}
  \]

- Server **vulnerabilities** present at all replicas.
  - A single attack can be used to subvert all replicas.
  - Diversity increases independence wrt attacks.
Replica Independence

Eschewing Shared Design / Code

Solution: Diversity!
- Expensive or impossible to obtain:
  - Development costs
  - Interoperability risks

- Leverage what diversity exists.
- Mechanically create "artificial diversity".
  ... Employ a program obfuscator.

Random key: 0110101100…

Obfuscator

Server Code

server replica
A mobile adversary can erode independence.

**Idea:** Proactively re-obfuscating server code defends against this:

- tolerates t compromises over *lifetime*
- versus -
- tolerates t compromises in *window of vulnerability*

X: server compromise
Replica Independence

Implementing Proactive Obfuscation

Challenges:
- State recovery
- Protect Obfuscator
- Protect Egg-timer
- Tolerate server outage
Replica Independence

Obfuscation: Goals and Options

Semantics-preserving random program rewriting...

**Goals:** Attacker does not know:
- address of specific instruction subsequences.
- address or representation scheme for variables.
- name or service entry point for any system service.

**Options:**
- Obfuscate source (arglist, stack layout, ...).
- Obfuscate object or binary (syscall meanings, basic block and variable positions, relative offsets, ...).
- All of the above.
Replica Independence

Independence By Obfuscation?

Given program $S$, obfuscator computes morphs:

$$T(S, K_1), T(S, K_2), \ldots T(S, K_n)$$

- **Attacker knows:**
  - Obfuscator $T$
  - Input program $S$

- **Attacker does not know:**
  - Random keys $K_1, K_2, \ldots K_n$
    - Knowledge of the $K_i$ would enable attackers to automate attacks!

Will an attack succeed against a **majority** of morphs?

- Seg fault likely if attack doesn’t succeed.
  - Integrity compromise $\rightarrow$ availability compromise.
All morphs implement the same interface.

- **Interface attacks.** Obfuscation cannot blunt attacks that exploit the semantics of that (flawed) interface.
- **Implementation attacks.** Obfuscation can blunt attacks that exploit implementation details.

**Def:** implementation attack: An input for which all morphs (in some given set) don’t all produce the same output.
Ultimate Goal: Determine the probability that a majority of morphs generate the same output for a set of attacks?

Modest goal: Understand how effective obfuscation is as compared with other defenses?
- Obvious candidate: Type checking
Replica Independence

Type Checking as a Defense

**Type checking**: Process to establish that all executions satisfy certain properties.
- **Static**: Checks made prior to exec.
  - Requires a decision procedure
- **Dynamic**: Checks made as exec proceeds.
  - Requires adding checks. Exec aborted if violated.

**Probabilistic dynamic type checking**: Some checks are skipped on a random basis.
Putting it Together: CoPrOF

Cornell Proactive Obfuscation Firewall

Specification:
- Unlikely that attacker can gain control of the service.
- A steady stream of attacks might block service. (But service is restored once that stream is terminated.)

Server:
- Receives messages from “outside”.
- Manages state (encodes history of messages seen).
- Forward subset of messages to “inside”.

Theory → Practice

CoPrOF: Prototype

Inside Traffic

Server Code

Controller

Server 0
Server 1

Server 6

Outside Traffic

N = 7 = 3(t+1) + 1 servers
Theory → Practice

CoPrOF: In the Flesh

Tom Roeder

Controllers:
7 server replicas

Processors:
3 GHz Pentium 4
OpenBSD 4.0
Controller:
- toggle power to reboot each server.
- Rebuilds image with obfuscation: re-numbers kernel calls (5 min!).
- transmits PXE boot and private keys to a rebooting server.
- broadcasts new public key certificate to all servers.

When server replica is powered on:
- Issues PXE boot request via network card.
- Awaits receipt of new image and priv keys.
- New image executes:
  - requests current state
  - votes on states it receives
    - PFsync basis to send and receive state.
  - starts processing packets (UCB Packet Filter PF)
Every sequence number has a master.
- Master selects an unprocessed message.
- Runs Byz PAXOS to ensure all replicas agree.
- All non-faulty replicas process that msg.

**Master for seqno s:**
- **Def:** $M(s) = s \text{ mod } 7$
- **Master for s:** smallest non-faulty successor of $M(s)$.
- **non-faulty p:** $p$ did not fail a timeout test for seqno $s$. 
Theory → Practice

Server Output Protocol

- Each CoPrOF host outputs msg with partial sigs.
- Client assembles t+1 partial sigs to obtain signed output of PF.

Alternative (so no client modification required):

- Replicas broadcast partial sigs to each other.
- Replicas assemble partial signatures and send to client.
  - Client can check if signature is correct.
  - Client does receive duplicate messages.
    - Replica snooping can suppress duplicate transmission.
CoPrOF: 2nd Generation Prototype

Inside Traffic

Server 0
Server 1
Server 6

Controller

Outside Traffic

N = 7 = 3(t+1) + 1 servers
Theory $\rightarrow$ Practice

CoPrOF: Ultimate Prototype

Inside Traffic

Server 0  \hspace{1cm} Server 1  \hspace{1cm} \ldots  \hspace{1cm} Server 6

CD-ROM \hspace{1cm} CD-ROM \hspace{1cm} CD-ROM

Outside Traffic

$N = 7 = 3(t+1) + 1$ servers

Egg timer

power

server coord
From 30,000 feet…

(What we really did)
Diversity as a Defense

Create independence from diversity.
- Independence increases the cost to attackers, since attacks against one component do not compromise another.

Forms of diversity:
- Static diversity ("in space").
- Dynamic diversity ("in time").
  - Also known as: “moving target defense”
  - Adds uncertainty for attackers, due to changes in system.
  - Can refresh [amplify] static diversity (e.g., proactive obfuscation).
Diversity Challenges

● Differences in interface:
  – Requires clients to adapt.

● Differences in internals but not interface:
  – Does not defend against exploits that leverage problematic interface semantics.
    ▪ ... Therefore: only defends against internal logic errors or under-specification.
  – Could require state migration or translation.
Why Attacks Work

- Some attacks are facilitated by information.
  - Brute force analysis (off-line / on-line).
  - Discovered by recon.

... Period of preparation. Then able to attack.

*Moving target defense invalidates preparation.*

- Some attacks exploit idiosyncratic technical details.
  - Specific behaviors when “underspecified operation” attempted are not available to attacker if those details change.
  - Changing the interpretation of “underspecified” blocks attacks that depend on that semantics.
Design of Moving Target Defenses

- **What to move?**
  - Must change some aspect of system that is used by attacker.

- **How to move?**
  - May require distinguishing “self” from “other.”

- **When to move**
  - **Reactive:** Based on system event, possibly attacker-caused.
  - **Proactive:** At fixed or random intervals.
Diversification techniques 1

Processor Storage
- Address space layout randomization (ASLR)
- Heap layout randomization
- Stack layout
  - Variable reordering on run-time stack
    - Can’t re-order fields within a variable.
  - Change direction of stack growth (e.g. support upward growth).
  - Stack frame padding.
- Register name randomization
  - Only some registers can be renamed
- Data representation (e.g., XOR with some key)
  - Values
  - Addresses (e.g., return pointers)
Diversification techniques 2

Processor Instructions

– Interface:
  ▪ Instruction set randomization (ISR)
  ▪ System call number randomization.
  ▪ Library location/name randomization.

– Internals
  ▪ Optimize code (or not)
Diversification techniques 3

System Level

- Network IP address, port, protocol
  - Port hopping (like spread spectrum comm)
- Virtual Machine
- Software stack / components
ISR Details

ISR defends against code-injection attacks, but does not work against attacks in data (e.g., attacks delivered as scripts).

- All binaries are pre-randomized when stored on disk.
  - Creates a randomly-mutating exec env whose language is not known to attackers
  - Attempts to guess code locations are hindered by mutating the env.
- Do randomization when binary is loaded and stored on disk.
- To de-randomize, need to know about context switches and calls, so correct key can be found for target of xfer.
- HW-based ISR: Hardware does xor on instruction fetch.
- SW-based ISR: Use binary translation.
How Attackers Bypass ISR

- Guess the key
- Get key using known plaintext attack...
  - Feasible for 16 bit XOR encrypt but not for AES encrypt
  - Best not to allow an attacker to export a binary in library or file sys
- Attacker finds another interpreter (e.g., uses another shell) that does not employ ISR.
  - Hard for sys owner to have found and fixed all interpreters.
Overcoming Diversity Defense

● Attacker: Use the full system rather than adding to it.
  – Use existing instructions to circumvent ISR.
  – Use existing storage to circumvent ASLR.
  – Recruit a “confused deputy” to perform operations.

● Attacker: Design attacks that work in many system variants (transcending diversity).
  – Use of a NOP sled to overcome uncertainty in memory layout when doing buffer overflow attack.