Symbolic Execution

Constraints
- Generation
- Accumulation
- Pruning

Scalability
- Server
- Client

Experiment
- XPilot
- Cap-Man
- Results

Outline

1. Symbolic Execution
2. Constraints
   - Generation
   - Accumulation
   - Pruning
3. Scalability
   - Server
   - Client
4. Experiment
   - XPilot
   - Cap-Man
   - Results

Authoritative State

Authoritative State is the amount of control or weight carried by a server or client. Used for the ultimate goal of protecting against client misbehavior.

Client-side Behavior Verification

- Constructs model of proper client execution using source code
- Exploits event loop process
- Symbolic execution
- Reasons if client input is possible

Another Approach

Audit Server
- Probabilistic
- Maintains committed state
- Client periodically sends hash
- Server challenge results in full state to be transmitted

Why ours is Better

Symbolic Execution
- Generate vulnerability signatures
- Generate inputs that will induce error conditions
- Automating mimicry attacks
- Optimizing privacy-preserving computations
- many more...
What is Symbolic Execution?

Symbolic Execution is the exploration of all path during the execution of a program. i.e. to find bugs within programs.

Initialization

Initial inputs are allowed to be “anything” and their memory locations are marked as symbolic. Conditional branches including a symbolic variable will create constraints and fork the execution.

Symbolic Values and Constraints

When a, b and c are marked as symbolic a logical constraint is added on a that it should be equal to the sum of b and c.

Constraint Sets

Every possible path has a set of constraints that must hold true along that path of execution. A constraint solver(such as KLEE) then provides concrete values for bad input.
### Outline

1. Symbolic Execution
   - Constraints
     - Generation
     - Accumulation
     - Pruning
2. Scalability
   - Server
   - Client
3. Experiment
   - XPilot
   - Cap-Man

### Simple Example

```plaintext
Round Constraint

\[(\text{loc} = \text{prev}_{-\text{loc}} + 1) \lor \\
(\text{loc} = \text{prev}_{-\text{loc}} - 1) \lor \\
(\text{loc} = \text{prev}_{-\text{loc}})\]

Accumulated Constraint

An Accumulated Constraint is a conjunction of round constraints that represents a sequence of possible paths taken over multiple client loops.

\[g(j) = \{ \text{loc}_j = \text{loc}_{j-1} + 1, \\
\text{loc}_j = \text{loc}_{j-1} - 1, \\
\text{loc}_j = \text{loc}_{j-1} \}\]
```
Symbolic Execution
Constraints
Scalability
Experiment

Accumulation Example

\[
\begin{align*}
loc_2 &= 9 \land loc_2 = loc_1 + 1 \\
loc_2 &= 9 \land loc_2 = loc_1 - 1 \\
loc_2 &= 9 \land loc_2 = loc_1
\end{align*}
\]

Outline

1. Symbolic Execution
   - Constraints
     - Generation
     - Accumulation
     - Pruning
   - Scalability
     - Server
     - Client
   - Experiment
     - XPilot
     - Cap-Man
     - Results

Pruning

Removes duplicate and insignificant constraints from round constraints before accumulation.
More on Round Constraints

Round constraints also result from server messages sent to the client.

- Eager
- Lazy

Eager Round Constraints

Every server message results in a constraint regardless of whether the client processes it.
- decoupled from verification and pre-computed

Lazy Round Constraints

Only client processed server messages result in constraints for the round.
- tightly coupled and model must be built by accessing client logs for viable server messages

Critical Path is Critical

Neither eager or lazy approach is fast enough to be implemented on the critical path for thousands of users.
Log Everything

Game operators already perform detailed logging. A log structure provides readability for GMs and allows for offline verification.

Outline

1 Symbolic Execution
2 Constraints
   - Generation
   - Accumulation
   - Pruning
3 Scalability
   - Server
   - Client
4 Experiment
   - XPilot
   - Cap-Man
   - Results

The Game

- 150,000 lines of C code
- Similar to Asteroids
  - multi-player modes (death match, CTF, racing)
  - better physics (fuel weight, acceleration)
- latest port in 2009 to iPhone

Client Cheating

Independent keystrokes should never result in KEYFIREREDSHOT and KEYSHIELD being sent to the server simultaneously.

Modifications

- add UDP acknowledgment
- add floating point constraint library
- remove trivial client information
  - "reliable" UDP information
  - graphical user modification
  - external graphics references
Lazy Verification

2000 rounds or about 1 minute of gameplay

![Graph of Verification Cost over Round Number]

90% avg=26ms 5% avg=3.4s 5% avg=14.9s

Eager Manual Tuning

Eager Verification

2000 rounds or about 1 minute of gameplay

![Graph of Verification Cost over Round Number]

90% avg=1.6s 10% avg=11.4s

The Game

- gives client more authoritative state
- about 1000 lines of C code
- simple Pac-Man clone

Cap-Man Cheating

With the client having most of the authoritative state it can simply tell the server an updated coord of movement regardless of making a valid move.

Outline

1 Symbolic Execution
2 Constraints
   - Generation
   - Accumulation
   - Pruning
3 Scalability
   - Server
   - Client
4 Experiment
   - XPilot
   - Cap-Man
   - Results

Multiple breaks in frame transmission and user keybind configuration.
Symbolic Execution Prevails

This method of server side verification allows increased security with less bandwidth, regardless of how much authoritative state the server maintains.