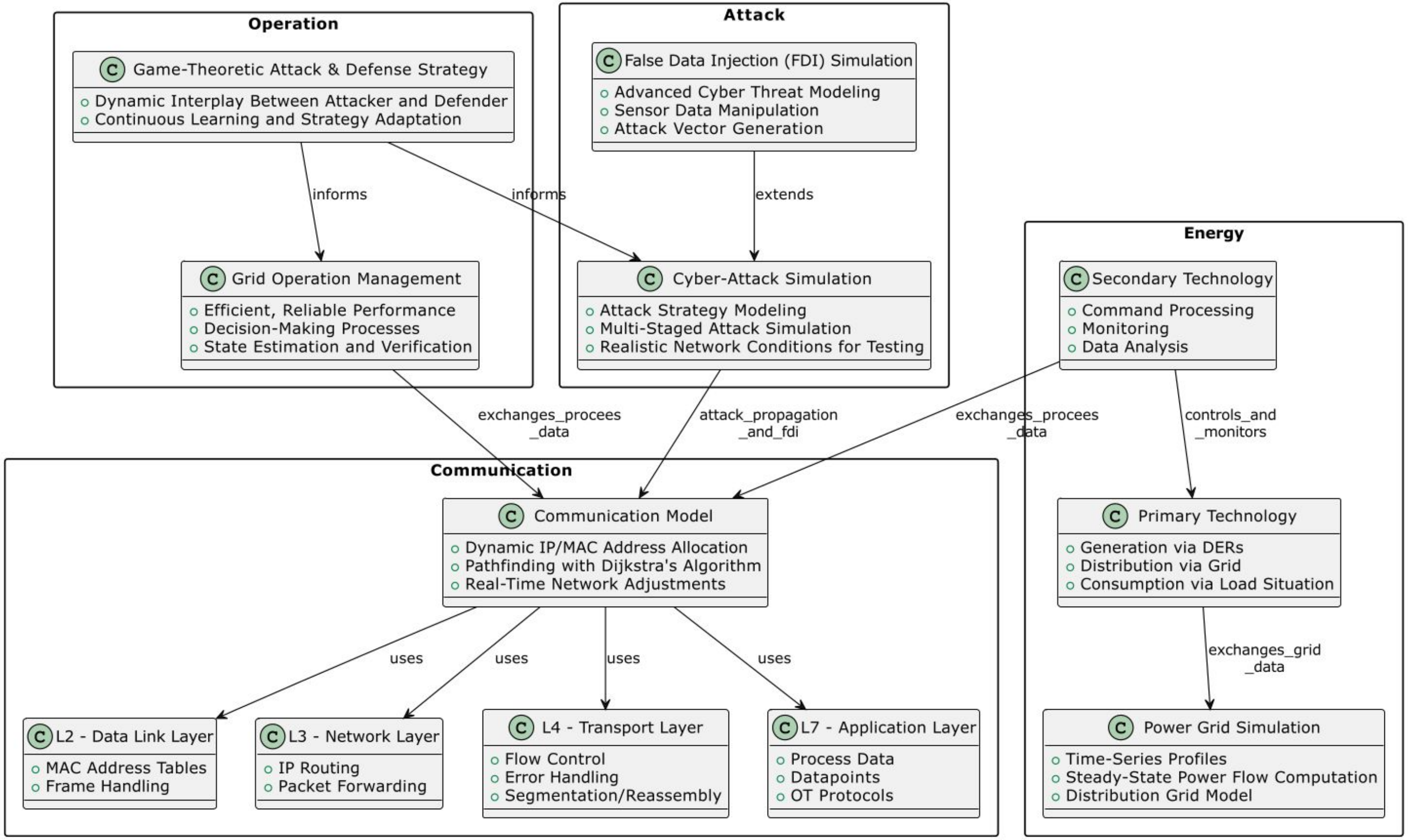
Open games for cybersecurity modelling

Aven Dauz

Cybersecurity and game theory

- Modelling: network agents and strategic interactions for network security problems
- Inform real defensive systems
- Conduct simulations of attack and defense strategies
- Predict rational behaviour of attackers
- Security problems such as:
  - Defensive resource allocation in smart grid
  - Ad-hoc networks and collaboration, IoT
  - Jamming/signalling
  - Cyber-physical systems



Operation
Game-Theoretic Attack & Defense Strategy
Dynamic Interplay Between Attacker and Defender
Continuous Learning and Strategy Adaptation
informs
informs
Grid Operation Management
Efficient, Reliable Performance
Decision-Making Processes
State Estimation and Verification
Attack
False Data Injection (FDI) Simulation
Advanced Cyber Threat Modeling
Sensor Data Manipulation
Attack Vector Generation
extends
Cyber-Attack Simulation
Attack Strategy Modeling
Multi-Staged Attack Simulation
Realistic Network Conditions for Testing
Energy
Secondary Technology
Command Processing
Monitoring
Data Analysis
exchanges_procees _data
attack_propagation _and_fdi
exchanges_procees _data
controls_and _monitors
Communication
Communication Model
Dynamic IP/MAC Address Allocation
Pathfinding with Dijkstra's Algorithm
Real-Time Network Adjustments
Primary Technology
Generation via DERs
Distribution via Grid
Consumption via Load Situation
uses
uses
uses
uses
exchanges_grid _data
L2 - Data Link Layer
MAC Address Tables
Frame Handling
L3 - Network Layer
IP Routing
Packet Forwarding
L4 - Transport Layer
Flow Control
Error Handling
Segmentation/Reassembly
L7 - Application Layer
Process Data
Datapoints
OT Protocols
Power Grid Simulation
Time-Series Profiles
Steady-State Power Flow Computation
Distribution Grid Model

Current state-of-the-art

1. Build a game-theoretic model to solve a specific problem for a system
2. Abstract definitions for attackers and defenders, make assumptions
3. Write algorithms for solving solution concept like BNE, determine probability distributions over attacker behaviours
4. Apply strategies to a testbed
5. Use learning algorithms to determine optimal defense strategies
6. Future work? Consider other variations of model to capture different attacks

How can we use compositional game theory?

1. Develop a design process for building game-theoretic cybersecurity models compositionally
2. Flexibly adapt models and leverage code-reuse to capture other attack scenarios
3. Use analytics provided by open games engine to inform defensive systems

Bayesian games

- In a Bayesian game, player knows some prior distribution, makes an observation, and updates their belief accordingly
- With posterior belief, try to maximize expected utility
  - Consider all other possibilities
- In extensive form, nature draws the type at the root of the tree

Non-deterministic open games

1. Define generalized **Lens** category, or lenses over **Kl**(D)
   a. D : finitary distribution monad
   b. Kl : Kleisli category
2. Can define a category $\mathbf{Game}_{\mathbf{Kl(D)}}$
   a. Objects: pairs of sets
   b. Morphisms: Bayesian open games

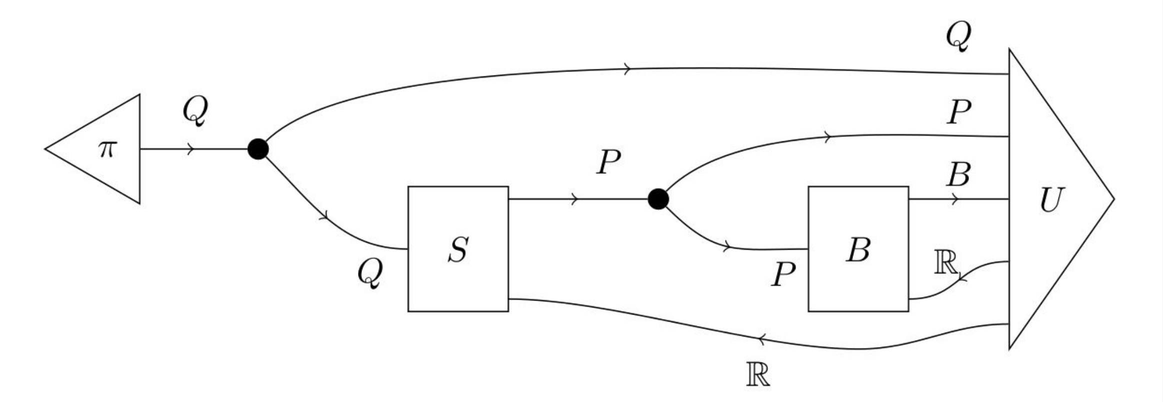


Figure: Market for lemons

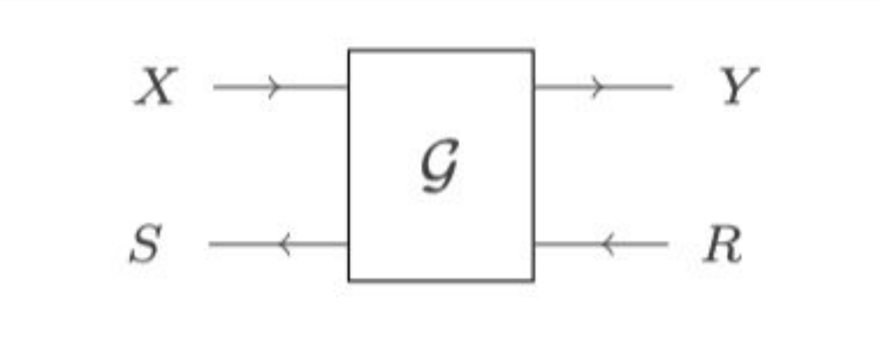
Haskell DSL

- Domain-specific language to build open games compositionally
- Haskell functions to encode payoffs, strategies
- Supply a set of strategies:
    - Calculate expected payoffs
    - Check if strategies are in equilibrium

```
bayesianPD  = [opengame|

   inputs    :      ;
   feedback  :      ;

   :---------------------------:
   inputs    :      ;
   feedback  :      ;
   operation : nature (uniformDist [Rat, NoRat]) ;
   outputs   : prisoner2Type ;
   returns   : ;

   inputs    :  ;
   feedback  :      ;
   operation : dependentDecision "prisoner1" (const [Confess, DontConfess]);
   outputs   : decision1 ;
   returns   : pdPayoff1 decision1 decision2;

   inputs    : prisoner2Type ;
   feedback  :      ;
   operation : dependentDecision "prisoner2" (const [Confess, DontConfess]);
   outputs   : decision2 ;
   returns   : pdPayoff2 prisoner2Type decision1 decision2 ;

   :---------------------------:

   outputs   :      ;
   returns   :      ;
   |]
```

Translation

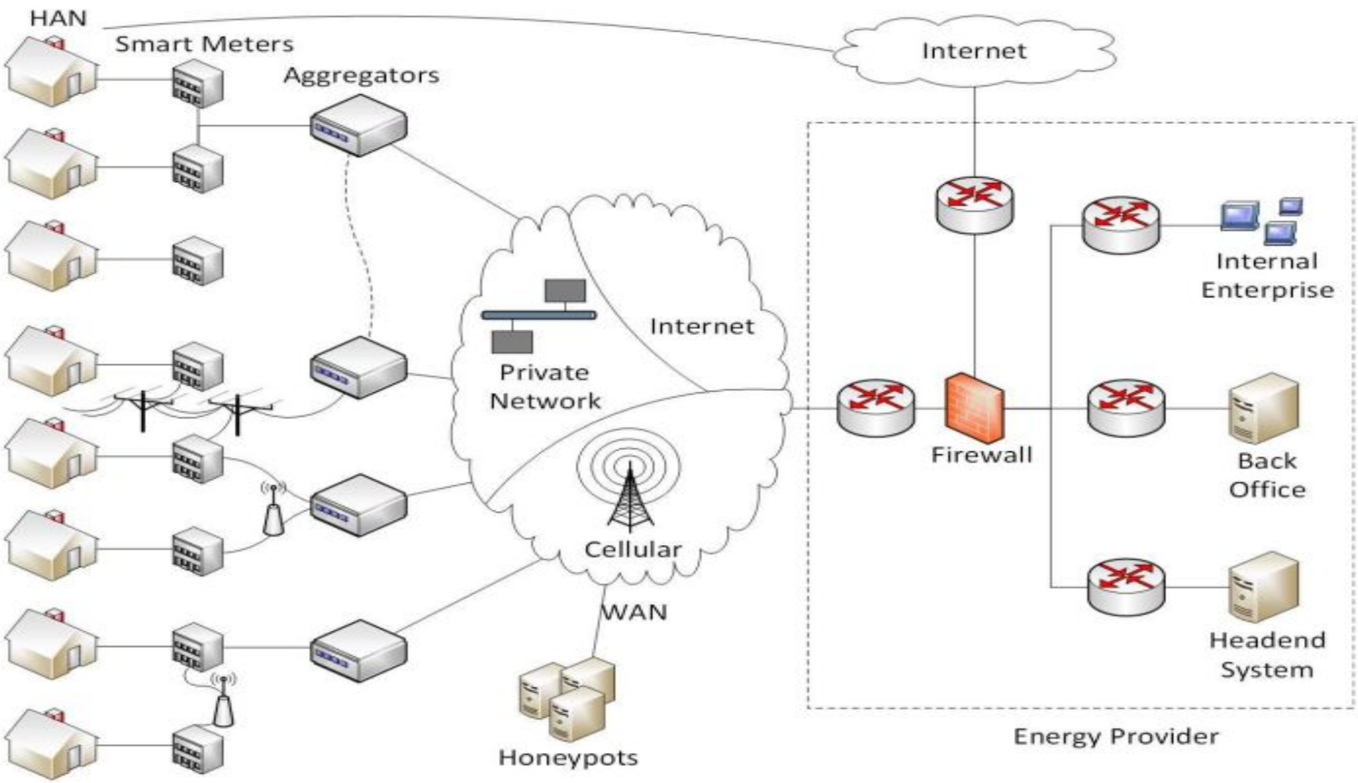
- X : inputs (set of observations)
- Y : outputs (set of possible decisions)
- R : returns (set of possible outcomes)
- S : outputs (set of possible co-outcomes)




```
aggregator aggregatorName = [opengame|

   inputs    :  defenseResourceAllocation, visitorDecision;
   feedback  :  ;

   :--------------------------:
   inputs    : defenseResourceAllocation, visitorDecision;
   feedback  :     ;
   operation : dependentDecision aggregatorName (const [Open, Close]);
   outputs   : aggregatorDecision ;
   returns   : aggregatorPayoff ;
   :--------------------------:

   outputs   : aggregatorDecision;
   returns   : aggregatorPayoff;
|]
```

Intrusion Detection System (IDS)

- System to monitor and detect malicious behaviour
- Track network traffic, seek anomalies, raise flags according to predetermined security policies
- Can be enhanced with honeypot deployment to act as decoy systems
    - Possibly gain knowledge of attackers
- Asymmetric information
    - Types, system configuration, common knowledge
- Goal: balance performance and defense with the optimal honeypot allocation scheme


HAN
Smart Meters
Aggregators
Internet
Private Network
Internet
Cellular
WAN
Honeypots
Firewall
Internal Enterprise
Back Office
Headend System
Energy Provider

Where do we start?

- Two-player sequential game
- Players: Visitor and Aggregator
  - Visitor can either “Access” or “NotAccess”
  - Aggregator observes Visitor move, then can either “Open” or “Close”
- Types:
  - Visitor can be either “Attacker” or “User”
  - Aggregator can be “Honeypot” or “Normal”

Building block: nature deals out players' types

```
natureVisitor probAttacker = [opengame|
    inputs : ;
    feedback : ;
        :----------------------------:

    inputs : ;
    feedback: ;
    operation : nature (distributionUser probAttacker);
    outputs    : visitorType;
    returns : ;
    :----------------------------:
    outputs : visitorType;
    returns : ;
|]
```

Building block: 2-player attack defense game

```
aggregatorHoneypot aggregatorName visitorName payoffConfig = [opengame|
   inputs    : [redacted], visitorType, defenseResourceAllocation;
   feedback  : [redacted] ;

   :----------------------------:

   inputs    : visitorType;
   feedback  :      ;
   operation : dependentDecision visitorName (const [Access, DoesNotAccess]);
   outputs   : visitorDecision ;
   returns   : calculateVisitorPayoff payoffConfig visitorType defenseResourceAllocation visitorDecision aggregatorDecision;

   inputs    : defenseResourceAllocation, visitorDecision;
   feedback  :      ;
   operation : dependentDecision aggregatorName (const [Open, Close]);
   outputs   : aggregatorDecision ;
   returns   : calculateAggregatorPayoff payoffConfig defenseResourceAllocation visitorType visitorDecision aggregatorDecision ;

:----------------------------:

   outputs   : [redacted];
   returns   : ;
|]
```

Building block: resource allocator

- Meta-representation of system administrator
- Allocates different defensive settings: HighInteractionHP, LowInteractionHP, Normal
- Payoff: expected value of defense over probability of detection
- Type: Active or Passive

```
resourceAllocator = [opengame|

    inputs: defenderType;
    feedback: ;

    :----------------------------:

    inputs: defenderType;
    feedback: ;
    operation: dependentDecision "allocator" (const allConfigurations);
    outputs: allocation1, allocation2, allocation3;
    returns: allocatorPayoff;

    :----------------------------:
    outputs: allocation1, allocation2, allocation3;
    returns: allocatorPayoff;
|]
```

Building block: 2-player attack defense game with allocator

```
aggregatorHoneypot aggregatorName visitorName payoffConfig = [opengame|
   inputs    :  defenderType, visitorType, defenseResourceAllocation;
   feedback  :  calculateAllocatorPayoff payoffConfig defenderType defenseResourceAllocation visitorType visitorDecision aggregatorDecision   ;

   :---------------------------:

   inputs    : visitorType;
   feedback  :      ;
   operation : dependentDecision visitorName (const [Access, DoesNotAccess]);
   outputs   : visitorDecision ;
   returns   : calculateVisitorPayoff payoffConfig visitorType defenseResourceAllocation visitorDecision aggregatorDecision;

   inputs    : defenseResourceAllocation, visitorDecision;
   feedback  :      ;
   operation : dependentDecision aggregatorName (const [Open, Close]);
   outputs   : aggregatorDecision ;
   returns   : calculateAggregatorPayoff payoffConfig defenseResourceAllocation visitorType visitorDecision aggregatorDecision ;

:---------------------------:

   outputs   : visitorDecision, aggregatorDecision;
   returns   : ;
|]
```

Parallel games: three subsystems

```
aggregatorDefenseGame (importanceA, importanceB, importanceC) payoffConfig = [opengame|
   inputs    :  defenderType, (visitorType1, visitorType2, visitorType3), (allocation1, allocation2, allocation3)    ;
   feedback  :    importanceA * aPerformance + importanceB * bPerformance + importanceC * cPerformance;

   :--------------------------:

   inputs    : defenderType, visitorType1, allocation1     ;
   feedback  :   aPerformance    ;
   operation : aggregatorHoneypot "A" "Alice" payoffConfig;
   outputs   : aliceDecision, aDecision;
   returns   : ;

   inputs    : defenderType, visitorType2, allocation2     ;
   feedback  :   bPerformance    ;
   operation : aggregatorHoneypot "B" "Bob" payoffConfig;
   outputs   : bobDecision, bDecision;
   returns   : ;

   inputs    : defenderType, visitorType3, allocation3     ;
   feedback  :   cPerformance    ;
   operation : aggregatorHoneypot "C" "Charlie" payoffConfig ;
   outputs   : charlieDecision, cDecision;
   returns   : ;

:--------------------------:

   outputs   : ;
   returns   : ;
|]
```

Model parameters

- Aggregators: proportion of energy resources
- Visitors:
  - Users: access to service
  - Attackers: expected value of attack
- Resource allocator: expected value of defense
- System parameters
  - Priors
  - Defense costs
  - Attack costs

```haskell
data Parameters = Parameters {
    probDetected :: DefenseAllocationMove -> Double,
    costOfAttack :: Double,
    costOfDefense :: DefenseAllocationMove -> Double,
    computationReductionUnderAttack :: DefenseAllocationMove -> Double,
    attackImpact :: DefenseAllocationMove -> Double,
    priorDistributionDefender :: Double,
    priorDistributionAttackerA :: Double,
    priorDistributionAttackerB :: Double,
    priorDistributionAttackerC :: Double,
    importanceLevel :: (Double, Double, Double),
    activeDefenseFactor :: Double
}
```

```haskell
data PayoffConfig =  PayoffConfig {
    calculateVisitorPayoff :: VisitorType ->
        (DefenseAllocationMove -> VisitorMove -> AggregatorMove -> Double),
    calculateAggregatorPayoff :: AggregatorPayoff,
    calculateAllocatorPayoff :: AllocatorPayoff
}
```

Some aggregator strategies

```
mixedAggregatorStrategy :: Numeric.Probability.Distribution.T Double AggregatorMove
mixedAggregatorStrategy = distFromList [(Close, 0.2), (Open, 0.8)]

aggregatorStrategy :: Kleisli Stochastic (DefenseAllocationMove, VisitorMove) AggregatorMove
aggregatorStrategy = Kleisli (\case {
    (HighInteractionHP, Access) -> playDeterministically Open;
    (HighInteractionHP, DoesNotAccess
) -> playDeterministically Close;
    (LowInteractionHP, Access) -> mixedAggregatorStrategy;
    (LowInteractionHP, DoesNotAccess
) -> playDeterministically Close;
    (Normal, Access) -> playDeterministically Open;
    (Normal, DoesNotAccess) -> playDeterministically Open;
})

aggregatorPowerSaverStrategy :: Kleisli Stochastic (DefenseAllocationMove, VisitorMove) AggregatorMove
aggregatorPowerSaverStrategy = Kleisli (\case {
    (HighInteractionHP, Access) -> mixedAggregatorStrategy;
    (HighInteractionHP, DoesNotAccess
) -> playDeterministically Close;
    (LowInteractionHP, Access) -> mixedAggregatorStrategy;
    (LowInteractionHP, DoesNotAccess
) -> mixedAggregatorStrategy;
    (Normal, Access) -> playDeterministically Open;
    (Normal, DoesNotAccess) -> playDeterministically Open;
})
```

Building block: deception game

Given a defense configuration “actualConfig”, resource allocator decides whether to relay an accurate or inaccurate representation of the configuration “portrayedAllocation”

Purpose: deceive possible attackers to access honeypots, and deter access to normal machines

```
signalDeceptionGame = [opengame|
   inputs    : defenderType, actualConfig;
   feedback  :      ;

   :-------------------------:

   inputs    : defenderType, actualConfig     ;
   feedback  :      ;
   operation : dependentDecision "allocator" (const allConfigurations);
   outputs   : portrayedAllocation;
   returns   : allocatorPayoff     ;
   :-------------------------:

   outputs   :  portrayedAllocation ;
   returns   :  allocatorPayoff   ;
  |]
```

Adaptation: visitor observes deceptive signal

```
deceptiveAggregatorHoneypot aggregatorName visitorName payoffConfig = [opengame|
   inputs    :  defenderType, visitorType, deceptiveSignal, actualAllocation;
   feedback  :  calculateAllocatorPayoff payoffConfig defenderType actualAllocation visitorType visitorDecision aggregatorDecision

   :----------------------------:

   inputs    : visitorType, deceptiveSignal;
   feedback  :      ;
   operation : dependentDecision visitorName (const [Access, DoesNotAccess]);
   outputs   : visitorDecision ;
   returns   : calculateVisitorPayoff payoffConfig visitorType actualAllocation visitorDecision aggregatorDecision ;

   inputs    : actualAllocation, visitorDecision;
   feedback  :      ;
   operation : aggregator aggregatorName;
   outputs   : aggregatorDecision ;
   returns   : calculateAggregatorPayoff payoffConfig actualAllocation visitorType visitorDecision aggregatorDecision ;

:----------------------------:

   outputs   : visitorDecision, aggregatorDecision;
   returns   : ;
|]
```

Building block: Markov game

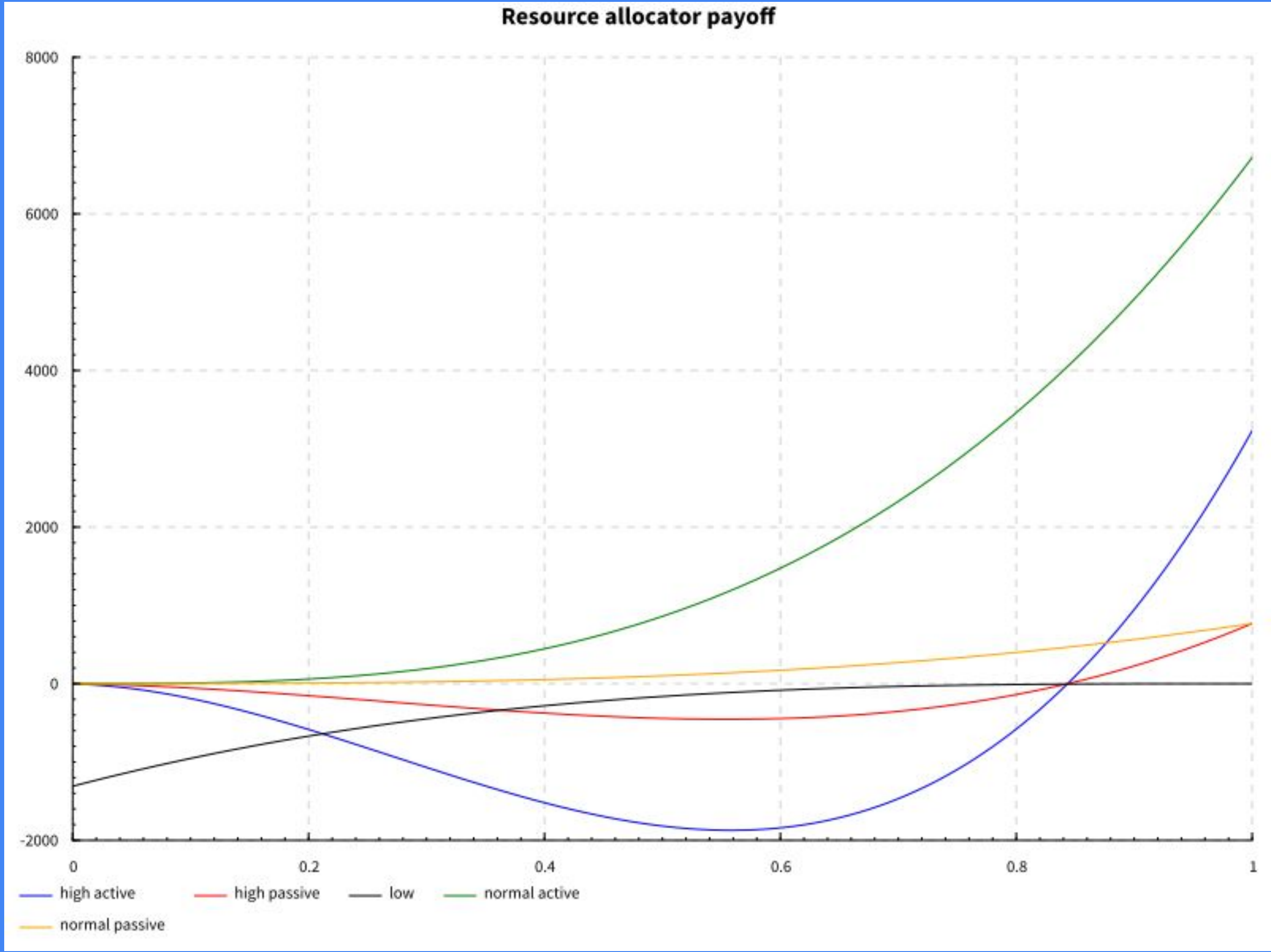
- Purpose: model repeated visitor access attempts
- Model as a repeated game with transition probabilities between states of the game
- Need to define a transition function
  - Transition probabilities: probability of detection

```
honeynetGameRepeated params = [opengame|

   inputs    :  visitorType, defenseResourceAllocation, didAccess, didOpen  ;
   feedback  :
      calculateVisitorPayoff (newPayoffConfig params) visitorType defenseResourceAllocation visitorDecision aggregatorDecision
         + visitorPayoff,
      calculateAggregatorPayoff (newPayoffConfig params) defenseResourceAllocation visitorType visitorDecision aggregatorDecision
         + aggregatorPayoff  ;

   :----------------------------:

   inputs    : visitorType, didAccess, didOpen;
   feedback  :      ;
   operation : dependentDecision "Alice" (const [Access, DoesNotAccess]);
   outputs   : visitorDecision ;
   returns   : visitorPayoff;

   inputs    : defenseResourceAllocation, didAccess, didOpen;
   feedback  :      ;
   operation : dependentDecision "A" (const [Open, Close]);
   outputs   : aggregatorDecision ;
   returns   : aggregatorPayoff;

   :----------------------------:
   outputs : visitorType, defenseResourceAllocation, visitorDecision, aggregatorDecision;
   returns : visitorPayoff, aggregatorPayoff;

   |]
```

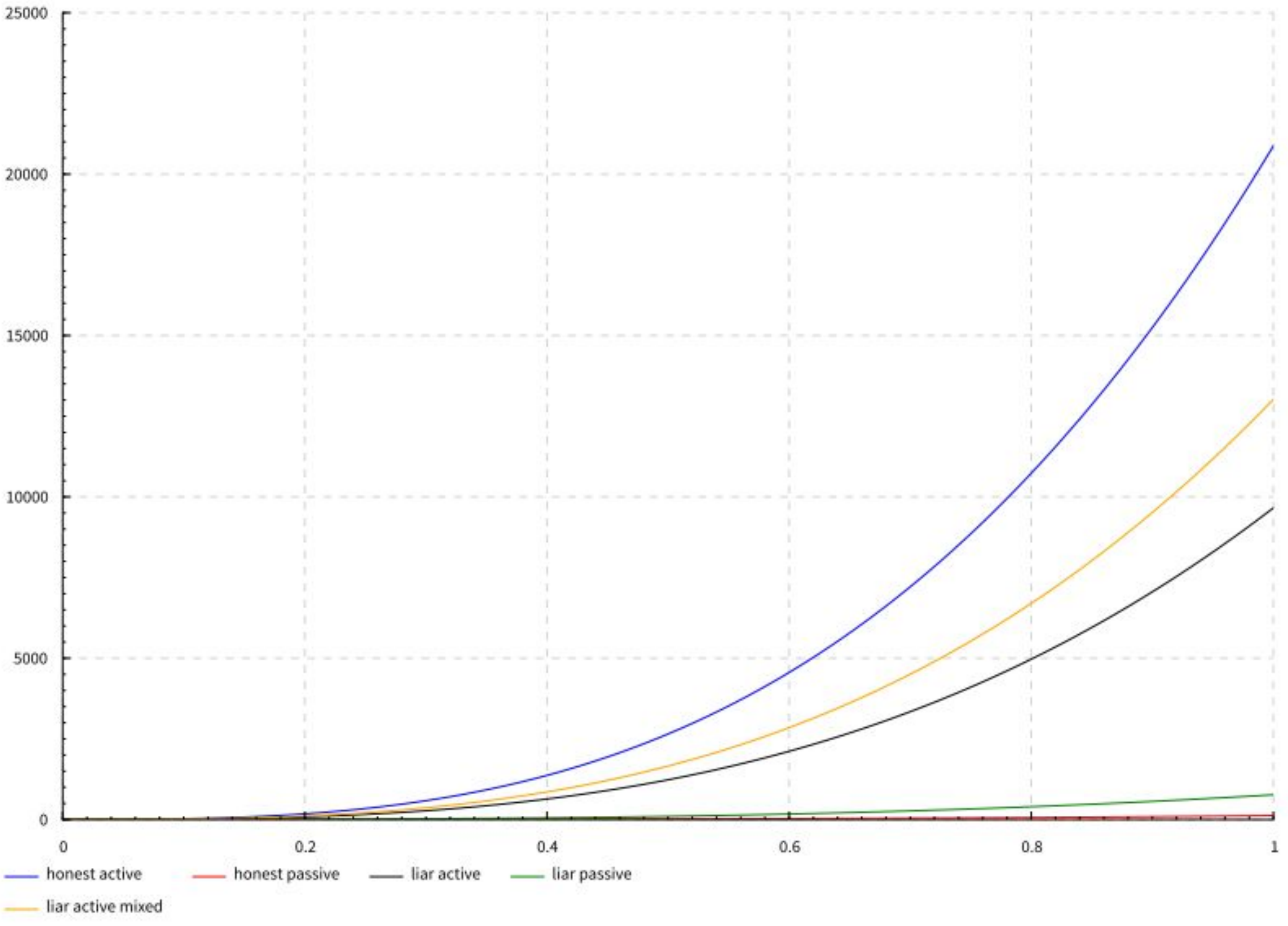
Data analysis

- Iterate over different parameters:
    - Prior distributions
    - Attack costs/impact
- Brittleness: extracting payoffs and parsing through unobservable state
- Graphing
    - Payoff curves for different strategies
- Conduct equilibrium checking

```
--------------------------------------------------------
-- Diagnosticinformation and processesing of information
-- for standard game-theoretic analysis

-- Defining the necessary types for outputting information of a BayesianGame
data DiagnosticInfoBayesian x y = DiagnosticInfoBayesian
  { equilibrium     :: Bool
  , player          :: String
  , optimalMove     :: y
  , strategy        :: Stochastic y
  , optimalPayoff   :: Double
  , context         :: (y -> Double)
  , payoff          :: Double
  , state           :: x
  , unobservedState :: String}
```


Resource allocator payoff
8000
6000
4000
2000
0
-2000
0
0.2
0.4
0.6
0.8
1
high active
high passive
low
normal active
normal passive

assigned (Normal,Normal,Normal) and all attackers, somePriorKnowledge mostly normal



Limitations

- Solution concepts
  - Evolutionary games
- Extracting data from engine
- Real-system scalability, n-players
- Performance against current analytics

Future work

- Learning algorithms for optimal defense strategies
- Multilayer defense
- Benchmarking and integration with defense systems
- Collaborative Intrusion Detection Network
- Organizational interdependent security
  - Networked games

Blockchain security applications

- Bitcoin Lightning Network protocol
  - Routing protocols
  - Wormhole attack
  - Griefing attack
- DDoS attacks between mining pools
  - Incentive mechanisms for pool managers
  - Discourage adverse behaviour