PASS Summary

Frédéric Vogel `vogelfr@ethz.ch` , ETH Zürich, FS18

- Blockchain Security
  - Hash functions
    - Cryptographic hash functions
    - Merkle trees
    - Digital signatures
    - Digital identity
    - Simple coin creation and transfer
  - Bitcoin
    - Distributed ledger/consensus
    - Blockchain
    - Concensus algorithm
  - Smart contracts
    - Ethereum
  - Security Properties
    - Semantics of smart contracts
  - Securify
- Attacks and Defenses of Deep Learning
  - Background
    - Classification through Machine Learning
    - Perceptron
    - Loss functions
    - Training with gradient descent
    - Deep learning models
  - Adversarial examples
    - Fast Gradient Sign Method (FGSM)
    - Minimal adversarial examples
    - Black-box attacks
    - Checking Robustness of Neural Networks
    - Decidable verification
  - AI2: AI for AI
    - Neural Network Analysis Problem
    - Abstract Interpretation
    - Zonotope Abstract domain
    - ReLU Layer Abstract Transformer
- Probabilistic Security
  - SPIRE
    - Privacy Policies and Verification
  - SLANG
  - Deobfuscation
    - Approach

Programmable Networks

- Network configuration is hard and misconfigurations are common.
- Local configuration changes have global effects on how traffic is routed.

Mathematical Background

Partially ordered sets (posets)

A *partial order* is a binary relation $\sqsubseteq \subseteq L \times L$ with the following properties:

Reflexivity
$\forall p \in L. p \sqsubseteq p$

Transitivity
$\forall p, q, r \in L. (p \sqsubseteq q \land q \sqsubseteq r) \Rightarrow p \sqsubseteq r$

Antisymmetry
$\forall p, q \in L. (p \sqsubseteq q \land q \sqsubseteq p) \Rightarrow p = q$

A *poset* $(L, \sqsubseteq)$ is a set L equipped with a partial ordering $\sqsubseteq$.

Bounds

Given a poset $(L, \sqsubseteq)$ and a set $Y \subseteq L$:

- $u \in L$ is an *uppper bound* of Y if $\forall p \in Y. p \sqsubseteq u$
- $\sqcup_Y \in L$ is a *least upper bound* of Y if $\sqcup_Y$ is an upper bound of Y and $sqcup_Y \sqsubseteq u$ whenever u is another upper bound of Y
- $l \in L$ is a *lower bound* of Y if $\forall p \in Y. l \sqsubseteq p$
- $\sqcap_Y \in L$ is a *greatest lower bound* of Y if $\sqcap_Y$ is a lower bound of Y and $l \sqsubseteq \sqcap_Y$ whenever l is another lower bound of Y

Complete lattices

A *complete lattice* $(L, \sqsubseteq, \sqcup, \sqcap)$ is a poset $(L, \sqsubseteq)$ where $\sqcup_Y$ and $\sqcap_Y$ exist for any $Y \subseteq L$.

Monotone functions

A function $f : A \to B$ between two posets $(A, \sqsubseteq)$ and $(B, \preceq)$ is *monotone* if

$$\forall a, b \in A : a \sqsubseteq b \Rightarrow f(a) \preceq f(b)$$

For a monotone function $f : A \to A$ we have

$$\forall a, b \in A : a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$$

Least fixed point

For a poset $(L, \sqsubseteq)$ and function $f : L \to L$ element $x \in L$ is a *fixed point* if $f(x) = x$

For a poset $(L, \sqsubseteq)$ and function $f : L \to L$ we say that $\mathrm{lfp}_f \in L$ is a *least fixed point* of f iff:

- lfp_f is a fixed point
- It is the least fixed point: $\forall a \in L : a = f(a) \Rightarrow \mathrm{lfp}_f \sqsubseteq a$

Tarski's fixed-point theorem

If $(L, \sqsubseteq, \sqcup, \sqcap)$ is a complete lattice and $f : L \to L$ is a monotone function, then lfp_f exsits.

Stratified Datalog

Datalog
Declarative logic programming language

Datalog Atoms
- Constants $C = \{alice, bob, carol\}$
- Variables $V = \{X, Y, Z\}$
- Predicates $P = \{parent(\_,\_), anc(\_,\_)\}$
- Ground atoms $G_{P,C} = \{parent(alice, alice), anc(alice, bob), \ldots\}$: only constants
- Atoms $A_{P,C,V} = \{parent(X, X), anc(alice, X), \ldots\}$: may contain variables

Datalog program
Set of rules of the form $a \leftarrow l_1, \ldots, l_n$
- $n \geq 0$, a is an atom
- $l_1, \ldots, l_n$ literals of the form a (positive literal) or $!a$ (negative literal)

Well-formed
Program is well-formed if for any rule all variables appearing in the head also appear in the body

Consequence operator

Interpretations
$\mathcal{I} = \mathcal{P}(G_{P,C})$

Substitutions
$\sigma : V \to C$

Consequence operator $T_P : \mathcal{I} \to \mathcal{I}$
$T_P(I) = \{\sigma(a) \mid a \leftarrow l_1, \ldots, l_n \in P, \exists \sigma : \forall i \in [1, \ldots, n] : I \vdash \sigma(l_i)\}$
with $I \vdash l_i$ if $l_1 = a$ and $a \in I$
$I \vdash l_i$ if $l_1 = !a$ and $a \notin I$

Positive Datalog program

A Datalog program is positive if its rules do not contain negative literals.

For any positive Datalog program P the consequence operator T_P is monotone.

The semantics of a positive Datalog program P is lfp_{T_P}
lfp_{T_P} can be computed by iteratively applying the consequence operator until reaching a fixed-point.

Stratified Datalog

T_P not monotone for Datalog programs with negation

- Same predicate rules in one stratum
- Negated predicates defined in lower stratum
- Positive predicates defined in current or lower stratum

$\rightarrow$ For each stratum compute the lfp that contains the lfp of the previous stratum

Automated Analysis of network configurations

Router configurations
Datalog input

Distributed Protocols (BGP, OSPF, Statis routes)
Datalog program

Forwarding plane
Datalog fixed-point

Batfish

1. Parse configurations (to derive input facts)
2. Compute forwarding plane (by computing fixed-point)
3. Check for violations (by querying the fixed-point)

$\rightarrow$ Is existing configuration correct?

Automatic Network Configuration Synthesis

Symbolic Execution

- Runs program with symbolic values $\rightarrow$ big constraint formula
- SMT solver used to find satisfying assignments to constraint formula
- Symbolic execution keeps *symbolic store* and *path constraint* $\rightarrow$ conjunction gives *symbolic state*

- Challenges:
  - Loops
  - Non-linear constraints
  - Hard-to-solve constraints (e.g. `x=hash(y)`)

SyNET

- Variables in head of of rule are quantified universally, those in body are quantified existencially

1. Encode Datalog program P into SMT constraints
2. Encode Datalog query q as assertions that must hold on the fixed-point
3. Get a model M that satisfies the conjunction of the above constraints
4. Derive input I from M by checking which atoms are `true` in M

Bounded unrolling of Datalog

Problem: in Datalog `p <- q` is only derived iff `q` is `true`
In logic $p \Leftarrow q$ is also satisfied if q is $false$ and p is $true$.

Bounded unrolling:

```
path(X,Y) <- link(X,Y)
path(X,Y) <- link(X,Z), path(Z,Y)
```

leads to the following constraints:
$\forall X, Y. path_1(X, Y) \Leftrightarrow link(X, Y)$
$\forall X, Y. path_2(X, Y) \Leftrightarrow (link(X, Y) \vee (\exists Z. link(X, Z) \wedge path_1(Z, Y)))$

Unrolling works only for *positive* queries.

Handling negative queries

No unrolling for negative queries

Stratified Datalog

Back step
Backtrack to step Synth P_i if step Synth P_{i-1} returns `unsat`

Synth P_n
Compute input I_n for stratum P_n such that $[P_n]_{I_n}$ satisfies q

Synth $P_{n-1}, \ldots, P_1$
Compute input I_i for stratum P_i such that $[P_i]_{I_i}$ produces the input I_{i+1} produced by previous step

Key Challenge: Scaling synthesis for OSPF

Efficient OSPF Synthesis

P
Set of all simple paths from src to dst

C
Set of all cost variables

$\phi(P, C)$
Encoded requirements

A
Cost assignment

Direct OSPF Synthesis

Find cost assignment A such that $\phi(P, A)$ holds

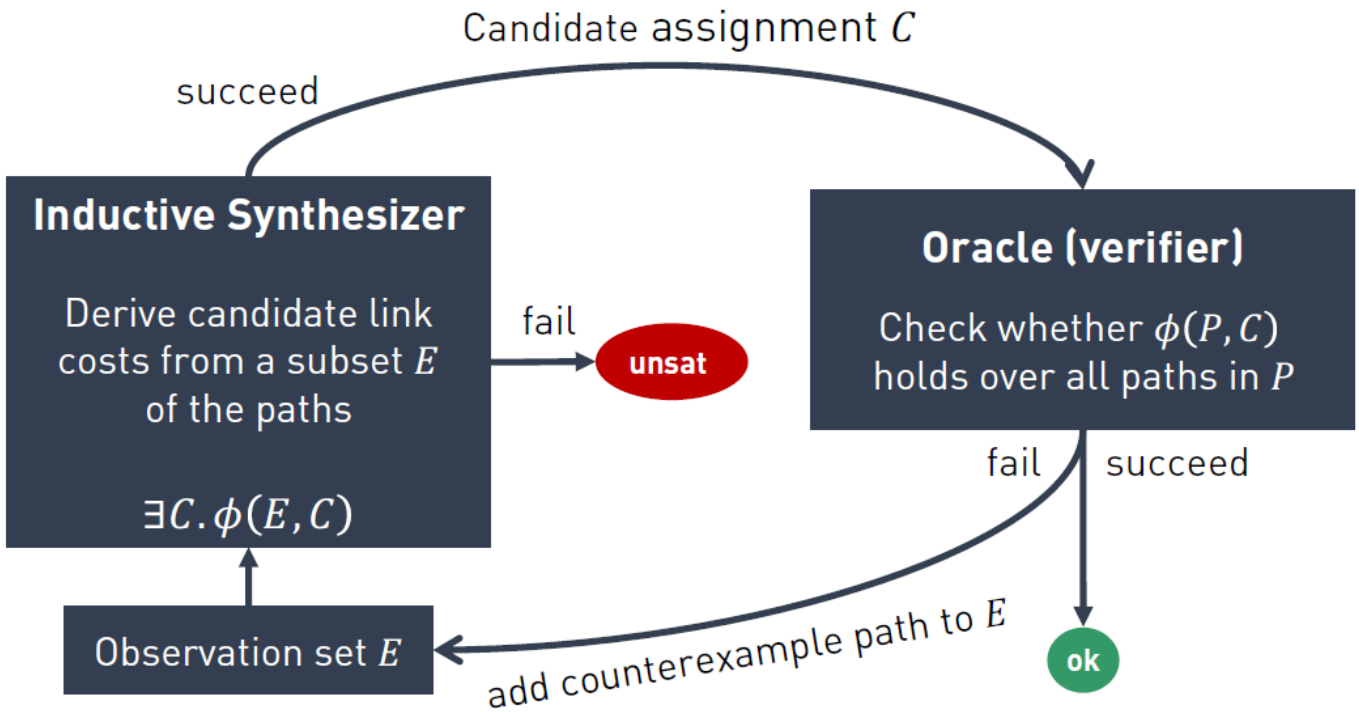
Formula for solver: $\exists C.\, \phi(P, C)$

$\rightarrow$ hard to solve:
- Constraint quantifies over all simple paths in P $\rightarrow$ exponentially many

Counter Example Guided Inductive Synthesis (CEGIS)

Insight: A small set E of path can be sufficient to constrain the solution:

$$\exists C.\, \phi(E, C), \qquad \text{where} \quad E = \{P_1, P_2, \ldots, P_k\} \subseteq P$$



Probabilistic Network analysis

- State
- Distribution
- Analysis
  - Input:
    - Input distribution φ_i over states of program
    - Statement (or program) s
  - Output:
    - Ouput distribution φ_o over states of program
- Normalisation:
  - $P_{new}(X) = \frac{P(X)}{\sum P(X)}$

Bayonet

- Networks exhibit probabilistic behaviours
- Can model in a *probabilistic programming language*
- Use existing solvers for inference. Benefits:
  - Do not reinvent the wheel
  - Can use to test a specialised solution
  - Provide benchmarks to general tools

Blockchain Security

Hash functions

- Function from arbitrary length data to fixed-sized output
- Deterministic
- Uniform
- Efficiently computable
- Collisions exist

Cryptographic hash functions

Properties

Given hash function $h : X \to Y$

Pre-image resistance
Given $y \in Y$, it is infeasible to find $x \in X$ such that $h(x) = y$

Second pre-image resistance
Given $x \in X$, it is infeasible to find $x' \in X$ such that $x \neq x'$ and $h(x) = h(x')$

Collision resistance
It is infeasible to find a pair (x_1, x_2) such that $x_1 \neq x_2$ and $h(x_1) = h(x_2)$

Applications

Data equality
If we know that $h(x) = h(y)$ it is safe to assume that $x = y$

Data integrity
To verify integrity of data d we remember $h_d = h(d)$. When obtaining d' from *untrusted* source we can verify it by checking $h_d \stackrel{?}{=} h(d')$

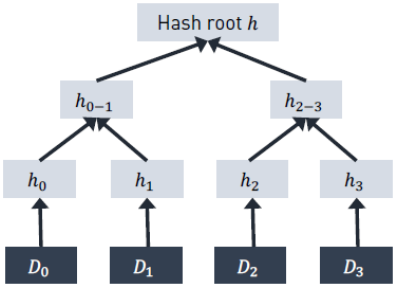
Cryptographic puzzles

Puzzle friendly
For any output y, if r is chosen from probability distribution with high min-entropy, it is infeasible to find x such that $h(r \| x) = y$

Search puzzle
Given puzzle ID id, chosen from a probability distribution with high min-entropy, and an output range $T \subseteq Y$ find a solution x such that $h(id \| x) \in T$. Because of puzzle-friendliness no strategy is better than trying random values of x

Merkle trees

- Hash root h obtained from *trusted* source
- Integrity of data elements verified by reconstructing hash root and comparing



Digital signatures

Allow only one user to sign but anyone to verify the signature

API

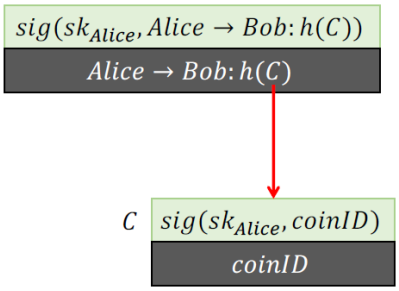
- $(sk, pk) = generateKeys(keySize)$
  - sk is the secret key (kept private)
  - pk is the public key (distributed)
- $sig = sign(sk, msg)$
- $verify(pk, msg, sig)$

Digital identity

- User generates key pair (sk, pk)
- $h(pk)$ is public name of user
- sk allows user to endorse statements $stmt$ using digital signature: $sig = sign(sk, stmt)$
- anyone can verify statements endores by user using $verify(pk, stmt, sig)$

Simple coin creation and transfer

- Alice creates coin (identified by $coinID$) and endorses it (by signing $coinID$)
- Alice can transfer the coin to Bob (identified by $h(pk_{Bob})$) by signing a transaction



Bitcoin

Distributed ledger/consensus

- All nodes must see the same state of the ledger
- The protocol terminates and all correct nodes decide on the same value
- Value must have been proposed by some correct node

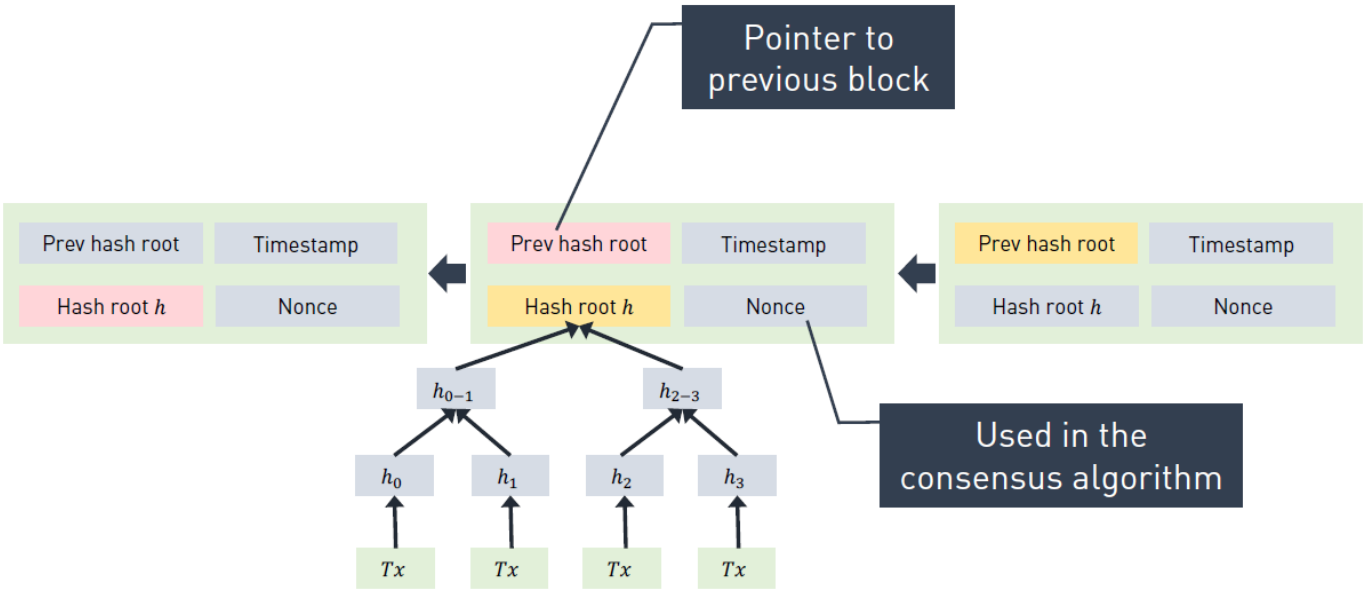
To make a transaction:

- Users *broadcast transactions* to the network nodes
- All nodes have a sequence of blocks of *agreed transactions* they have reached consensus on

- Each node has set of *outstanding transactions*

Blockchain

Chain of blocks of transactions



Concensus algorithm

1. New transactions are broadcast to all nodes
2. Each node collects new transactions in a block
3. In each round a *random block* gets to broadcast its block
4. Other nodes accept the block only if all transactions in it are valid (unspent, valid signatures)
5. Nodes express their acceptance of the block by including its hash in the next block they create

Proof of work

To select a random node that broadcast its block: select nodes in proportion to a resource that no one can monopolize

Given

- Previous block with hash h_{prev}
- Merkle tree consisting of all new transactions with hash h_{tx}

Find

- Nonce $nonce$ such that

$$hash(h_{prev}|h_{tx}|nonce) \leq \text{difficulty}$$

Properties

- Difficult to compute

- Parametrisable cost
- Easy to verify
- Key security assumption: Attacks infeasible if majority of miners weighted by hash power follow the protocol

Honest nodes extend the longest valid branch

Incentives to extend longest chain

Block reward

The creator of a block can:

- Include special coin-creation transactions in the block (fixed value)
- Choose recipient address of this transaction
- **Only** paid if block ends up on the long-term concensus branch

Transaction fees

- Creator of transaction may choose to make output value less than input value
- Difference is transaction fee that goes to block creator

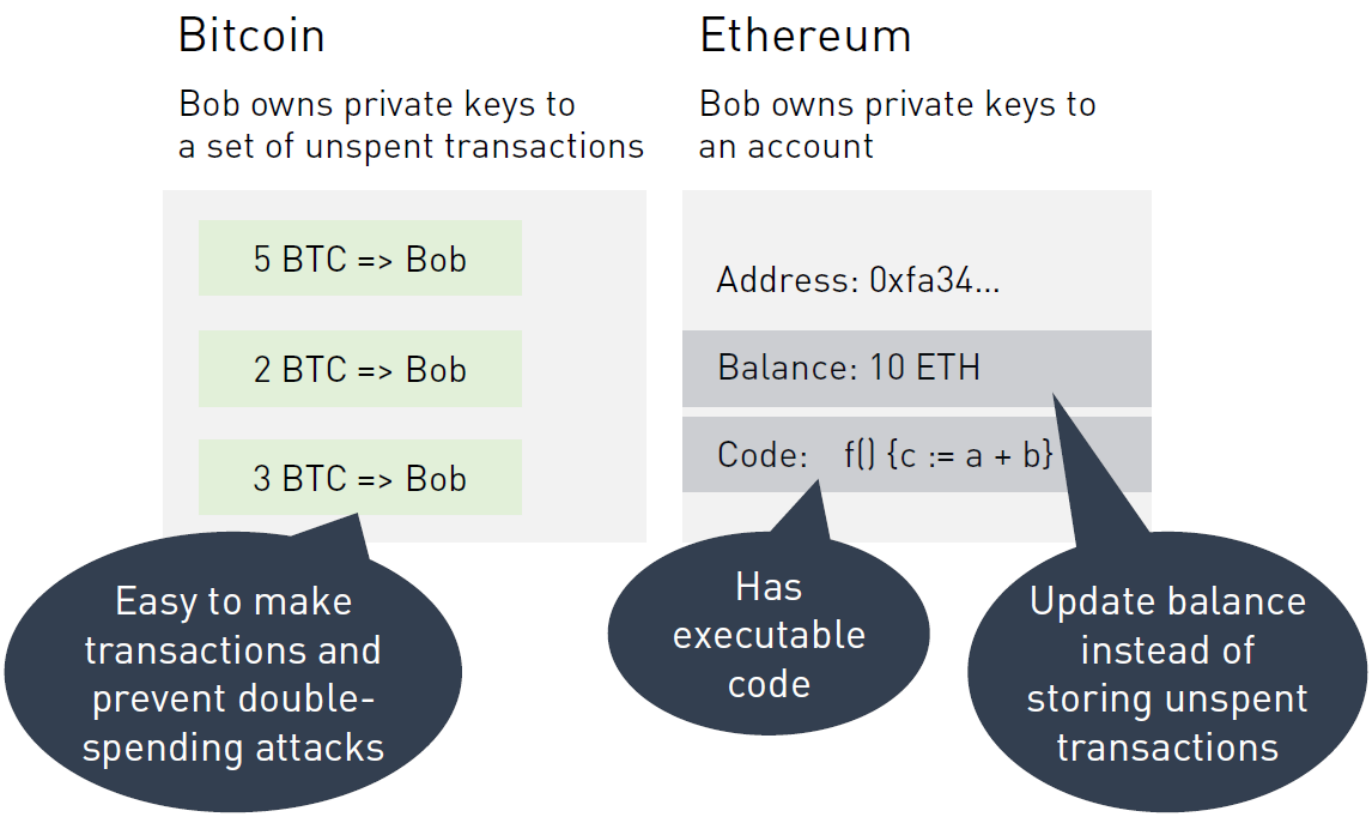
Smart contracts

- Computerised transaction protocol that executes terms of contract
- Used to satisfy common contractual conditions
- Used to minimise exceptions (malicious or accidental)
- Used to minimise need for trusted intermediaries

Ethereum

Decentralised platform designed to run smart contracts

- Similar to a world computer that executes code and maintains state of all smart contracts
- Latest block stores latest local state of all smartest contracts
- Transactions result in executing code (calling a function) in target smart contracts
- Transactions change state in one or more smart contracts
- Turing complete



Ethereum accounts

Externally owned accounts

- Owned by some external entity
- Contains:
  - Address
  - Ether balance

Contract accounts

- "Owned" by contract
- Contains:
  - Address
  - Ether balance
  - Associated contract code
  - Persistent storage

Gas

- Each transaction requires *gas* to fuel contract execution
- Each EVM opcode requires a fixed amount of gas to execute
- Every transactions specifies the maximum ether the sender is willing to spend on the transaction
- Contract successfully executed?
  - Yes: Unspent ether is refunded to sender
  - No: Execution reverts without refunding

Authorisation

Any user can call arbitrary functions in contracts. The contract must explicitly restrict access to sensitive information.

DAO Bug

```
uint balance = 10;

function withdraw() {
    if (balance > 0):
        msg.sender.call.value(balance)();
    balance = 0;
}
```

Calling `withdraw()` multiple times before balance is set to zero $\rightarrow$ profit.
Attacker stole $ 150M of ether from The DAO

Partiy bug #1

Fallback function in wallet contract delegating any non-defined function to wallet library contract.

Attacker re-initialized wallet owner

White-hat hackers saved 377k ETH by hacking vulnerable wallets themselves and giving back funds to owners

Parity bug #2

User accidentally(?) deleted wallet library contract $\rightarrow$ no more withdraws are possible on wallets using the library $\rightarrow$ funds freezed in place, no way of recovering.

$ 170M frozen this way

Security Properties

Solidity, Vyper
: High-level languages

EVM
: Low-level code, stack-based, no types, no functions

Semantics of smart contracts

System state

Storage S
: Persistent initial storage is defined by constructor

Memory M
Non-persistent (re-initialised before executing transaction)

Stack Q
Size limited to 1024 elements, each element 256 bit

Block information B
Number, timestamp, etc.
Fixed for a given transaction

State σ
$\sigma = (S, M, Q, B)$
Storage, memory and stack may change as contract executes for given transaction

Transaction $T = (caller, data)$
Transaction sender ($caller$)
Transaction data ($data$)

Trace
$(\sigma_0, op_0) \rightarrow_T (\sigma_1, op_1) \rightarrow_T \cdots \rightarrow_T (\sigma_{n-1}, op_{n-1}) \rightarrow_T (\sigma_n)$
op_{n-1}: STOP
σ_n: final state
Each op_i is next EVM op-code to be executed
Set of all traces for given contract defines the contract's semantics

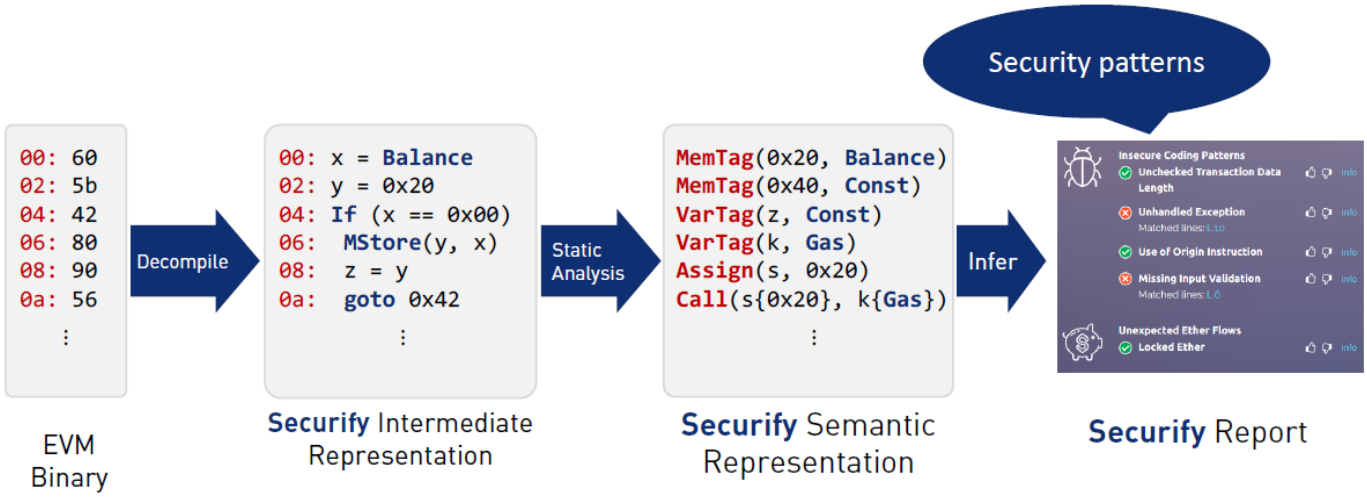
Unchanged storage after call
A contract does not change storage after calls iff for any two traces that are identical up to $call$ instruction the final storage S_n and S'_n are identical.

Unrestricted write
A write to offset o is unrestricted iff for any user address a there is a transaction $T = (a, \_)$ and a trace $(\sigma_0, op_0) \rightarrow_{(a,\_)} \cdots \rightarrow_{(a,\_)} (\sigma_i, op_i) \rightarrow_{(a,\_)} \cdots$ such that $op_i = SStore(o, \_)$

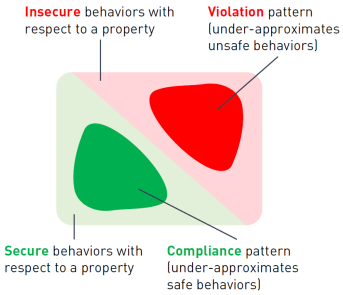
Locked ether
A contract locks ether iff it *can receive* ether and *can not transfer* ether

Further security properties
- Unexpexted ether flows
- Insecure coding (e.g. unpriviledged write)
- Use of unsafe input
- Reentrant method calls (e.g. DAO bug)
- Manipulating ether flow via transaction reordering

Securify



- Security properties get encoded into Compliance and Violation patterns
- Static analysis using fixed-point computation
  - Pointer analysis
  - Data-flow analysis
  - Taint analysis
  - others
- Analysis expressed declaratively in Datalog
  - Declarative (concise specs of analyis)
  - Modular
  - Can leverage existing scalable Datalog solvers



Attacks and Defenses of Deep Learning

Deep model
Mathematical model trained from input-output examples
Mainly for tasks that are easy to perform but hard to formally define

Background

Classification through Machine Learning

Take a *data-driven* approach and learn a *model* (function) f from data
$f : I \to C$ *approximates* the optimal function $f^* : I \to C$
A model is an *architecture* with real-valued *weights* and *biases*
The architecture defines the space of expressible models

Perceptron

A classifier parametrized by *weights* $w_0, \ldots, w_{n-1}$ and *bias* b
Input $\overline{x} = (x_0, \ldots, x_{n-1})$

$$f(x) = \begin{cases} 1 & \sum w_i x_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Linearly seperates the space

Loss functions

Goal: model and optimal classifier are equal: $\forall i.\ f^*(i) = f(i)$

Induces a loss function which measures how good a classifier is: $\sum_{i \in I} [f^*(i) \neq f(i)]$

> $[\cdot]$ Iverson brackets $[true] = 1, [false] = 0$

Goal: find weights and biases of the model f which minimises loss

Empirical loss

Problem: not all labels of input data I are given
Approach: estimate loss function on some of labeled inputs D
Given labeled data D compute *empirical loss* $\sum_{i \in D} [f^*(i) \neq f(i)]$

To avoid overfitting to D:

- Split D into training set D_{Tr} and test set D_{Te}
- Learn model by minimising loss on D_{Tr}, estimate loss on D_{Te}

Optimal solution of $\sum_{i \in D}[f^*(i) \neq f(i)]$ is a *global minimum* $\rightarrow$ define *differentiable* loss function and find point that nullifies its derivative.

Training with gradient descent

Mean Squared Error

$MSE = \sum_{i \in D_{Tr}}(f^*(i) - f(i))^2$

If model consists of single weight $f(i) = w \cdot i$, then $MSE = \sum_{i \in D_{Tr}}(f^*(i) - w \cdot i)^2$

The minimum nullfies the derivative $\sum_{i \in D_{Tr}} 2(f^*(i) - w \cdot i) \cdot (-i) = 0$

Can compute best model (i.e. w) *analytically

Gradients

For $f(w_0, \ldots, w_{n-1}, b)$ with >1 parameter, derivative of $MSE = \sum_{i \in D_{Tr}}(f^*(i) - f(i))^2$ is generalised to *gradient*

A gradient is a vector defined by partial derivative of the variables

$$\nabla MSE = \left(\frac{\partial MSE}{\partial w_0}, \ldots, \frac{\partial MSE}{\partial w_{n-1}}, \frac{\partial MSE}{\partial b} \right)$$

Minimum nullifies ∇MSE in all dimensions $\rightarrow$ hard to compute analytically

Gradient descent

1. Initialise randomly with certain values for weights/bias a_0
2. Compute ∇MSE at a_i
3. Next point is the one maximising decrease in MSE
   $a_{i+1} = a_i - \gamma \nabla MSE(a_i)$, γ is learning rate
4. If loss is small enough, complete, otherwise, repeat from 2.

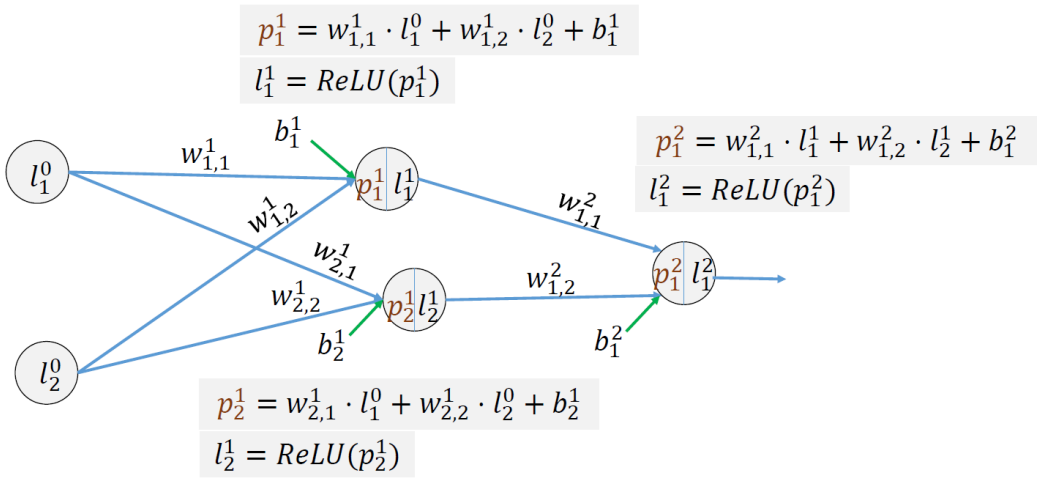
Deep learning models

- Perceptrons are too simplisti models
- Deep models combine multiple simple models

Deep model
Directed graph of neurons organised in layers

Neuron
Simple model followed by activation function

Activation Function
Determines whether to propagate output of function
$ReLU(a) = \max(0, a)$



Feed forward Neural network (FF NN)

Neurons are connected to all neurons in the next layer

Deep models can be tuned by gradient descent, $\rightarrow$ need to compute the partial derivatives of all weights and biases $\rightarrow$ *Hard to compute

Back propagation

Backprpagate the common parts of the derivatives

Multiclass classification

- Output layer has a neuron for each class
- Outputs normalised to a probability distribution with softmax

$$P(c_i) = \frac{e^{f_i(x)}}{\sum_j e^{f_j(x)}}$$

Convolutional Neurol Networks (CNN)

- Neurons are not connected to every part of the input
- Weights are shared between neurons
- Input size reduced by propagating part of it

$\rightarrow$ reduces number of weights significantly

Convolutional layer
Neurons are connected to local regions in the inputs and share weights

Pooling layer
Neurons compute downsample of the input to reduce dimensionality

Fully connected layer
Identical to FF NN, used to perform classification

Adversarial examples

- High accuracy of a network does not imply that the network has learned the underlying concept
- For inputs from new distribution network may behave unexpectedly

Fast Gradient Sign Method (FGSM)

Goal: from correctly classified x find $x' = x + \eta$ so that $\|\eta\|_\infty \leq \epsilon$

Use loss function of network to optimise η for a target t, set every index of η to be in the direction of the loss' gradient

1. Compute perturbation
   $\eta = \epsilon \cdot \text{sign}(-\nabla_x \text{loss}_t(x))$, where

$$\nabla_x \text{loss}_t = \left(\frac{\partial \text{loss}_t}{\partial x_1}, \ldots, \frac{\partial \text{loss}_t}{\partial x_n} \right) \quad \text{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

2. Perturbe the input
   $x' = x + \eta$
3. Check whether $f(x') = t$

Minimal adversarial examples

Given

- Neural network $f : X \to C$
- Input $x \in X$
- Target label $t \in C$ such that $f(x) \neq t$

Compute a minimal η such that $f(x + \eta) = t$

- By computing the salieny map of f, indicating where f changes the most

For some inputs small perturbations can significantly change the output

Goal:

Given the Jacobian

$$J_f = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$$

increase $\frac{\partial f_t}{\partial x_i}$, decrease $\frac{\partial f_j}{\partial x_i}$, $\forall j \neq t$

Which x_i should we change?
For a target t increase those for which $\frac{\partial f_t}{\partial x_i}$ increases, while the combined classification of the other labels f_j decreases

Saliency Map

Saliency map
matrix S defining intensity of inputs whose increase helps the most to accomplish the goal

$$S(x_1, \ldots, x_{n-1}, t)[i] = \begin{cases} 0 & \text{if } \frac{\partial f_t}{\partial x_i} < 0 \text{ or } \sum_{j \neq t} \frac{\partial f_j}{\partial x_i} > 0 \\ \left(\frac{\partial f_t}{\partial x_i} \cdot \left| \sum_{j \neq t} \frac{\partial f_j}{\partial x_i} \right| \right) & \text{otherwise} \end{cases}$$

- Given FF NN f, an input x and a target t:
  1. define $x' = x$
  2. while $f(x') \neq t$:
     1. compute saliency map $S(x'_1, \ldots, x'_n, t)$
     2. let i be index maximising $S(x'_1, \ldots, x'_n, t)$
     3. $x'_i += \theta$

Black-box attacks

- Attacker leverages prior knowledge:
  - Dataset type
  - Common architecture
- Attacker trains *other* model to find advesarial examples

Generate input

How to generate input-output examples?
Assume attacker can collect initial set of inputs similar to dataset, then synthesise more inputs and ask for classification to get new input-output examples

1. Define architecture for $\hat{f}$, the function approximating f

2. Let $D = \{(i_1, o_1), \ldots, (i_k, o_k)\}$ be our initial training set
3. Repeat N times:
    1. Train $\hat{f}$ on D
    2. Generate new inputs and query f for their output
    3. Extend D with these input-output examples

Generating inputs based on gradient

We generate inputs that improve our confidence in $\hat{f}$ by picking inputs whose neighbourhood show a large variance in $\hat{f}$
The gradient of (a single class) $\hat{f}$ points to where it changes most

1. Define architecture for $\hat{f}$, the function approximating f
2. Let $D = \{(i_1, o_1), \ldots, (i_k, o_k)\}$ be our initial training set
3. Repeat N times:
    1. Train $\hat{f}$ on D
    2. For every $(i, o) \in D$
        1. Compute $i' = i + \lambda \cdot \text{sign}\left(\frac{\partial f_o}{\partial i_1}, \ldots, \frac{\partial f_o}{\partial i_n}\right)$
        2. Add $(i', f(i'))$ to D

Checking Robustness of Neural Networks

Robustness

Given a model f and an input x, f is robust for x in a neighbourhood N_x if

$$\forall x' \in N_x : f(x) = f(x')$$

Common N_x:

$$N_x^\epsilon = \{x' \mid \|x' - x\|_p < \epsilon\} \quad \text{with } p = 0, 1, 2, \infty$$

Testing

Check a strict subset of points in N_x

If we find $x' \in Nx$ such that $f(x') = f(x)$, f is not robust in x. Otherwise can not garantee robustness.

Verification

Analyse all points in N_x
f is robust in x iff $\forall x' \in N_x : f(x') = f(x)$

In general this is *undecidable*, but under certain assumptions may be decidable.

Decidable verification

Focus on robustness of x for $p = \infty$

$$\begin{aligned} N_x^\epsilon &= \{x' \mid \|x' - x\|_\infty < \epsilon\} \\ &= \left\{x' \mid |x_0 - x_0'| < \epsilon \wedge \cdots \wedge |x_n - x_n'| < \epsilon\right\} \end{aligned}$$

Goal

Check if $\forall x' \in N_x^\epsilon . f(x) = f(x')$, that is check
$|x_0 - x_0'| < \epsilon \wedge \cdots \wedge |x_n - x_n'| < \epsilon \wedge f(x) = f(x')$$

To determine whether this is satisfiable we define a solver

Theory and Interpretation

Theory
Signature Σ
Interpretation I consisting of domain and interpretation for symbols in Σ

Satisfiability Modulo Theory (SMT) problem
Decision problem of determining whether logical formula in a certain theory is satisfiable

Simplex

Solver for the SMT problem for the linear arithmetic theory

$$\Sigma = \left\langle +, -, \cdot, \leq, \geq, =, 0, \frac{1}{1}, \frac{1}{2}, \ldots, \frac{2}{1}, \frac{2}{2}, \ldots \right\rangle$$

Determines the satisfiability of a conjunction of formulas of the form

$$\sum_{x_i \in X} c_i x_i \bowtie d_i \qquad \bowtie \in \{\leq, \geq, =\}$$

X is a set of variables, c_i, d_i are constants

Iterative algorithm

TODO Understand...

AI^2: AI for AI

Neural Network Analysis Problem

Given

- a neural network N
- a property over inputs φ
- a property over outputs ψ

check whether $\forall i \in I. i \models \varphi \Rightarrow N(i) \models \psi$ holds

Challenges:

- Property φ over inputs usually captures *unbounded* set of inputs
- Existing symbolic solutions *do not scale* to large networks

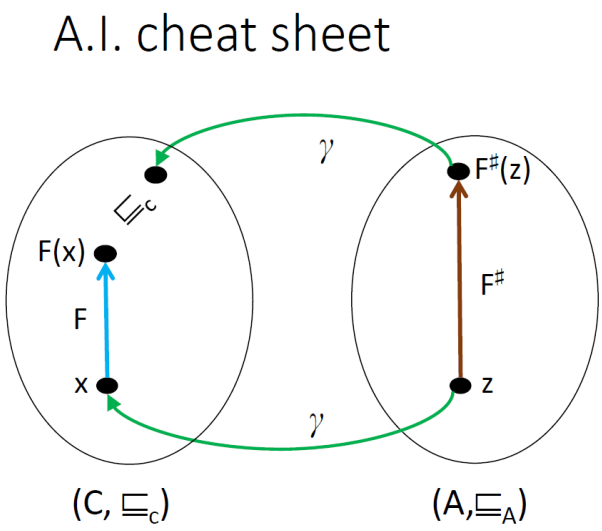
Key technical insight

Deep Neural Nets: Affine transforms & restricted non-linearity
+
Abstract interpretation: scalable and precise numerical domains

Abstract Interpretation

1. Select *abstract domain* based on type of *properties* you want to prove
2. Define abstract semantics for *programming language* w.r.t. to abstract domain
   - Define abstract transformers (effect of statements/expressions on abstract domain)
   - Prove abstract semantics are sound w.r.t. concrete semantics of programming language
3. Iterate abstract transformers over abstract domain until *fixed point*

Fixed point is the *over-approximation* of the program.



- (C, $\sqsubseteq_c$) is the concrete lattice. An element x in C is a set of concrete program states.
- (A,$\sqsubseteq_A$) is the abstract lattice. An element z in A is an abstract element that represents a set of concrete states.
- F is your program. F(x) applies it on set of states x. F is monotone. Least fixed point of F (LFP F) is an element in C that captures all reachable states of F – the set may be infinite or unbounded so we typically cannot compute it.
- $F^\sharp$ is the abstract transformer. $F^\sharp(z)$ applies $F^\sharp$ to abstract element z. $F^\sharp$ should be monotone (see Tarski's theorem)
- γ is the concretization: it defines to which concrete states an abstract element maps to. γ is monotone. It is key to defining what it means for $F^\sharp$ to approximate F.
- We iterate $F^\sharp$ to a fixed point. If $F^\sharp$ approximates F, then its least fixed point (LFP $F^\sharp$) approximates LFP F! We can compute LFP $F^\sharp$!

Function approximation

$F : C \to C$ and $F^\sharp : A \to A$
The approximation of F is defined as $\forall z \in A : F(\gamma(z)) \sqsubseteq_C \gamma(F^\sharp(z))$

Least Fixed Point Approximation

Given

1. *monotone* functions $F : C \to C$ and $F^\sharp : A \to A$
2. $\gamma : A \to C$ is *monotone*
3. $\forall z \in A : F(\gamma(z)) \sqsubseteq_C \gamma(F^\sharp(z))$ ($F^\sharp$ approximates F)

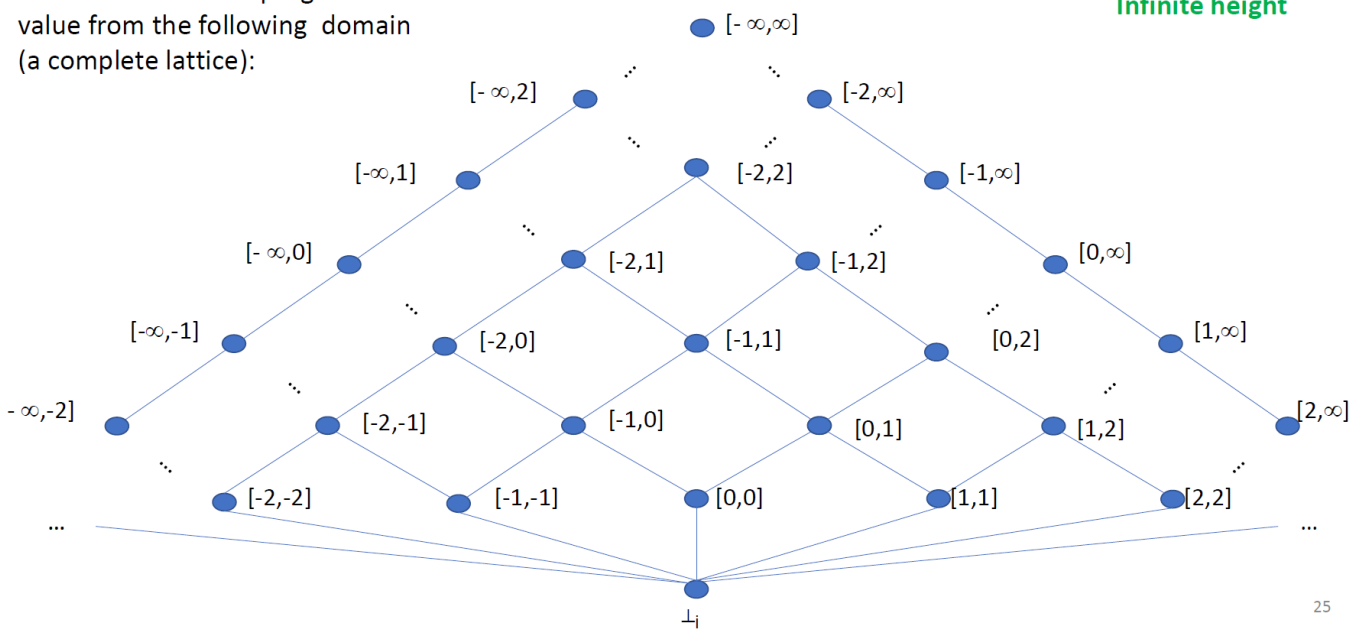
$\Rightarrow \quad \mathrm{lfp}(F) \sqsubseteq_C \gamma(\mathrm{lfp}(F^\sharp))$

Example

Interval domain

Each variable in the program takes a value from the following domain (a complete lattice):

Infinite height



25

Semantics

If we add $\perp_i$ to any other element we get $\perp_i$
If both operands are not $\perp_i$ we get

$$[x, y] + [a, b] = [x + a, y + b]$$
$$[x, y] * [a, b] = [x * a, y * b]$$

Iteration

1. Start from $\perp$
2. Variables initialised to $\top$
3. Iterate and replace interval according to semantics until fix-point

Zonotope Abstract domain

Zonotope

Polytope formed by Minkowski sum of line segments in any dimension, convex, point-symmetric, all faces polytopes of $n-1^{th}$ degree with point symmetry

Minkowski sum

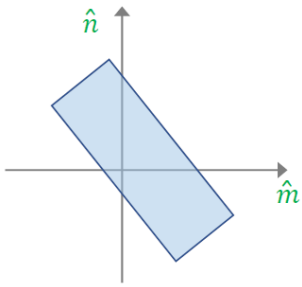
Dilatation, sum of set of position vectors A and B

$$A+B=\{a+b \mid a \in A, b \in B\}$$

Zonotope Abstract domain

- Numerical domain, *exact* for linear operations
- Each variable (*abstract neuron*) is captured in affine form
- More extensive version of interval domain: still about single variables, but can be related through parameters

For two concrete neurons n and m the abstract neurons will be



$$\hat{n}=a_0^n+\sum_{i=1}^{k} a_i^n \epsilon_i$$

$$\hat{m}=a_0^m+\sum_{i=1}^{k} a_i^m \epsilon_i$$

The *meaning* γ is a polytope centered around a_0^n and a_0^m
ϵ_i: noise terms ranging $[-1,1]$ shared between abstract neurons
a_i^n: real number that controls magnitude of noise
Closed under affine transforms, not closed under joints and meets

Operations

Multiplication by a constant C

$$\left(a_0^n+\sum_{i=1}^{k} a_i^n \epsilon_i\right) \cdot C=C \cdot a_0^n+\sum_{i=1}^{k} C \cdot a_i^n \epsilon_i$$

Adding two variables (abstract transformer is exact)

$$\left(a_0^n+\sum_{i=1}^{k} a_i^n \epsilon_i\right)+\left(a_0^m+\sum_{i=1}^{k} a_i^m \epsilon_i\right)=\left(a_0^n+a_0^m\right)+\sum_{i=1}^{k}\left(a_i^n+a_i^m\right) \epsilon_i$$

Multiplication of two variables (non-linear, approximation is computed)

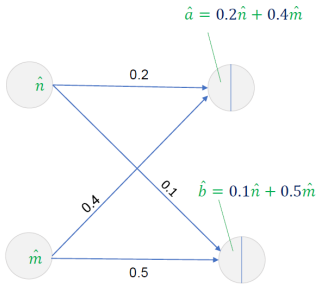
$$\left(a_0^n + \sum_{i=1}^{k} a_i^n \epsilon_i\right) \cdot \left(a_0^m + \sum_{i=1}^{k} a_i^m \epsilon_i\right) = \left(a_0^n a_0^m\right) + \sum_{i=1}^{k} \left(a_i^n a_0^m + a_i^m a_0^n\right) \epsilon_i + \sum_{i=1}^{k} \sum_{j=1}^{k} a_i^m a_j^n \cdot \underbrace{\epsilon_i \epsilon_j}_{\epsilon_{i,j}}$$

$$\epsilon_{i,j} \in \begin{cases} [-1,1] & \text{if } i \neq j \\ [0,1] & \text{if } i = j \end{cases}$$

ReLU Layer Abstract Transformer

Affine

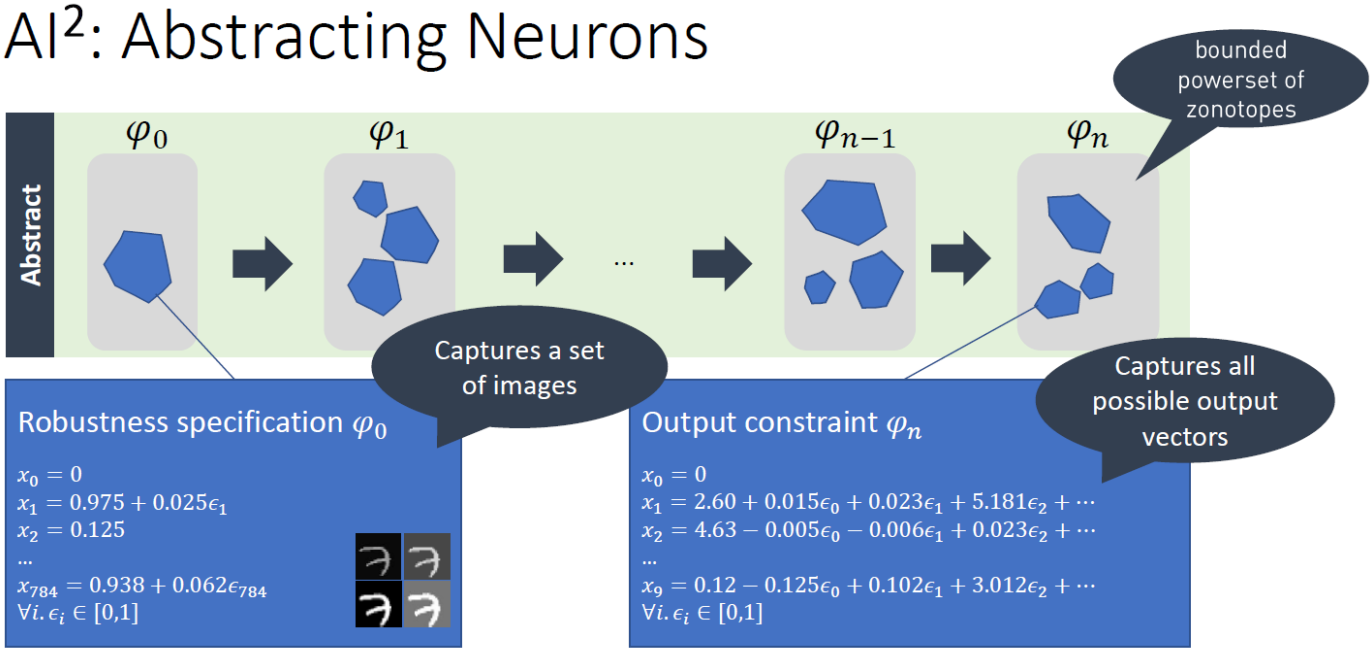
Compute effect of affine transforms on input zonotope → result for output abstract neuron.
$\hat{a}$ and $\hat{b}$ in example, represent zonotope Aff_z



ReLU

Take Aff_z and propagate it through ReLU transformers in layer, optaining one large zonotope as output of layer

$$f^{\sharp}_{ReLU} = f^{\sharp}_k \circ \cdots \circ f^{\sharp}_1(Aff_z)$$
$$f^{\sharp}_i(\psi) = (\psi \sqcap \{x_i \geq 0\}) \sqcup \psi_0$$
$$\psi_0 = \begin{cases} [[x_i = 0]](\psi) & \text{if } (\psi \sqcap \{x_i < 0\}) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$



Label i is possible iff: $\varphi_n \sqcap \{\forall j. x_i \geq x_j\} \neq \bot$

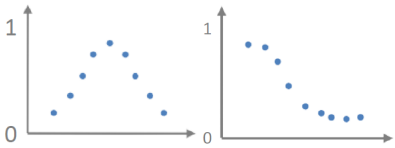
Probabilistic Security

Motivation

Public output reveals information about *confidential* input. We want to *restrict* the amount of information revealed.

SPIRE

Bayesian Inference



Prior attacker belief → Posterior attacker belief

Prior $P(I = i)$
$\rightarrow$ Query $P(O = o \mid I = i)$
$\rightarrow$ Joint Prior $P(I = i, O = o)$
$\rightarrow$ Posterior $P(I = i \mid O = o)$

Privacy Policies and Verification

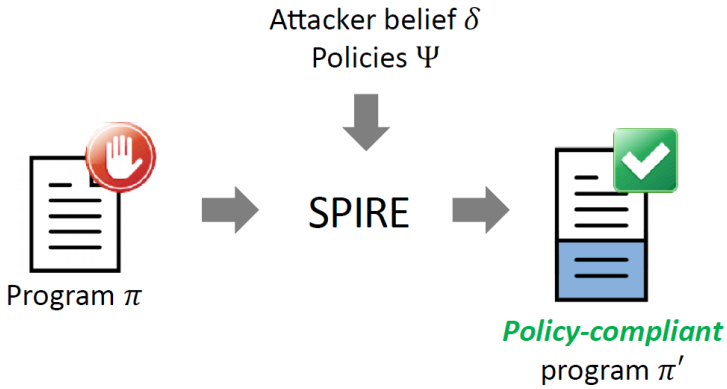
Given

Attacker belief δ, program π and privacy policy Φ

Check

Could running the program π violate the policy Φ?

$$\Phi \equiv \forall o.\, P(I \in S \mid O = o) \in [a, b]$$
$$\text{Secret } S \subseteq I: \quad \text{An event}$$
$$\text{Belief bound:} \quad [a, b] \subseteq [0, 1]$$



In general multiple policies $\Phi_1, \ldots, \Phi_k$

Privacy Enforcement

Enforcement ξ is an equivalence relation over O such that $\forall o.\, P(I \in S \mid O \in [o]_\xi) \in [a, b]$
Intuition: Only report $[o]_\xi$ instead of o, conflate outputs.

Permissiveness

Number of equivalence classes, $\left|\frac{O}{\xi}\right|$

Precision

Number of equivalence classes of size , $|\{o \in O \mid |[o]_\xi| = 1\}|$

Given probabilities $P(O = o), P(I \in S \mid O = o)$ for all o
Want enforcement $\xi(\forall o.\, P(I \in S \mid O \in [o]_\xi) \in [a, b])$

Synthesis of optimally permissive enforcement ξ is NP-equivalent (NP-hard and NP-easy)
Synthesis of optimally precise enforcement ξ of a single policy is possible in $O(n \log n)$ time ($n = |O|$)

Greedy Heuristics for Permissive Enforcement

1. Pick most violating class
2. Select candidate to merge
3. Merge, repeat

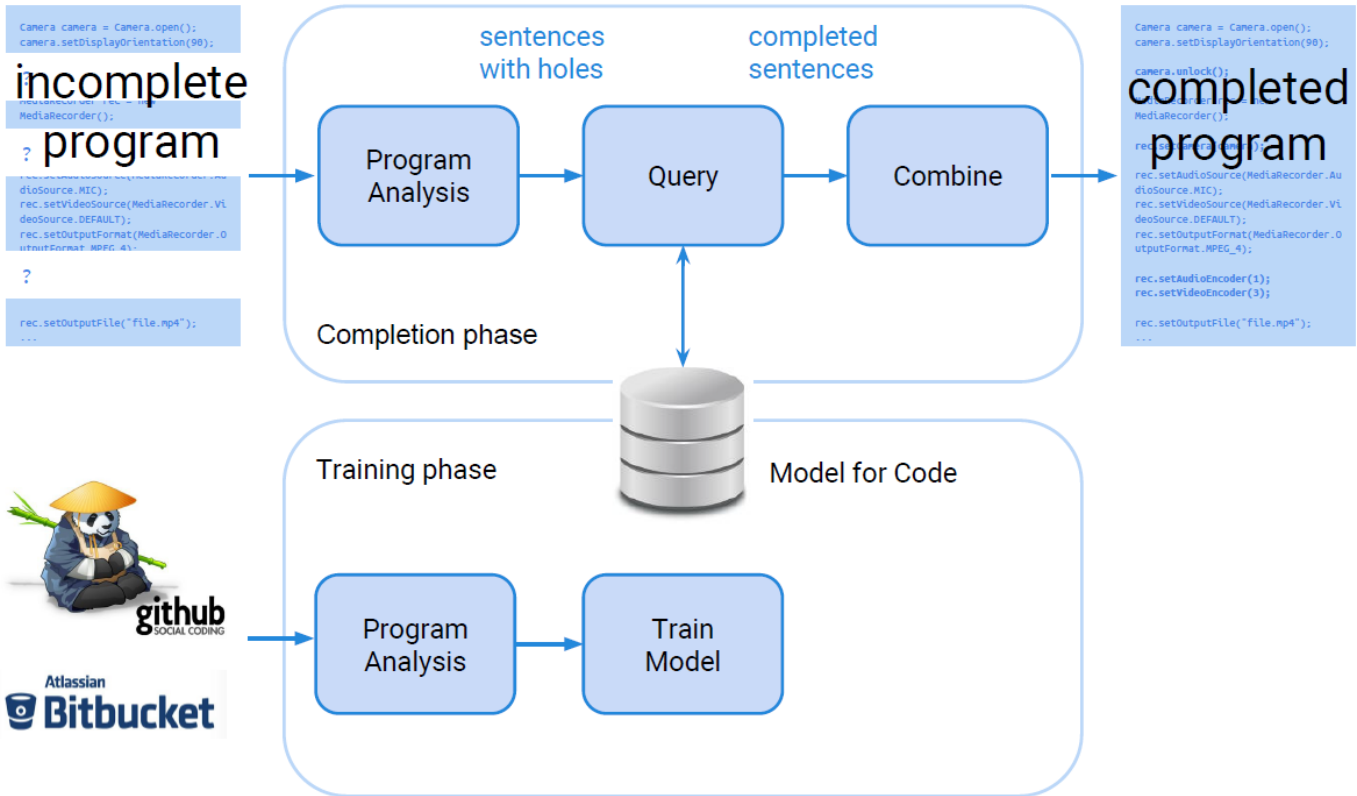
Optimal Algorithm for Precise Enforcement

1. Join all violating classes in to class C
2. Non-violating: done
   Otherwise, wlog[1] $P(S \mid o \in C) > b$
3. Need to merge more outputs into C such that

$$P(I \in S \mid o \in C) = \frac{\sum_{o \in C} P(I \in S \mid O = o) \cdot P(O = o)}{\sum_{o \in C} P(O = o)} \leq b$$

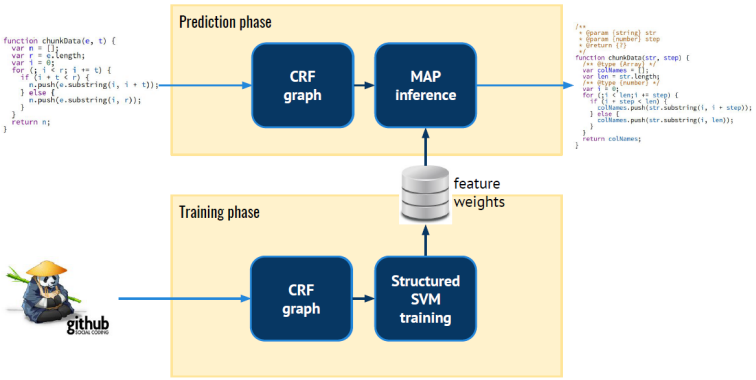
4. Sort by contribution, pick smallest first, merge into C

SLANG



Deobfuscation

Goal: predict *unknown* facts given *known* facts



Approach

1. Model: Conditional Random Fields (CRF)
2. Query: MAP inference
3. Learning: Structured SVM

Conditional Random Fields

$P(y \mid x) = \frac{1}{Z} \prod \varphi_i(x, y)$, x: known facts, y: unknown facts
Z (partition function) makes P a valid probability distribution, very expensive to compute

MAP inference

Goal: find most likely assignment of y that satisfies constraints

$$y = \text{argmax}_{y'} P(y \mid x) = \text{argmax}_{y'} \frac{1}{Z} \prod \varphi_i(x, y)$$

Good news: for this query Z is unecessary
Bad news: still NP-hard
Solution: approximate algorithm

Structured SVM

Other representation: $P(y \mid x) = \frac{1}{Z} \exp \sum \lambda_i f_i(x, y)$
Learning finds weights λ_i from training data $D = \{x^{(j)}, y^{(j)}\}_{j=1..n}$
D: programs with facts of interest already manually annotated, big codebase to learn from

Learn weights such that:

$$\forall j \forall y \sum \lambda_i f_i(x^{(j)}, y^{(i)}) \geq \sum \lambda_i f_i(x^{(j)}, y) + \delta(y, y^{(j)})$$

For all training data samples the given prediction is better than any other prediction by at least a margin

- stochastic (sub-)gradient descent
- MAP inference as subrouting
- no partition function Z

1. without loss of generality ⮌