Program Analysis for System Security and Reliability FS20

jasonf
joels

March 2020

Contents

1 Blockchain Basics and Bitcoin

1.1 Hash Functions

Properties:
- Arbitrary input size
- Fixed-output size
- Deterministic (equal inputs hash to the same value)
- Uniform (inputs are mapped evenly to the space of possible outputs)
- Efficient (should be fast to compute)

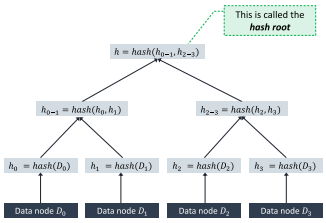
Note: Collisions always exists since input space is unbounded. However, cryptographic hash functions ensure collisions are hard to find

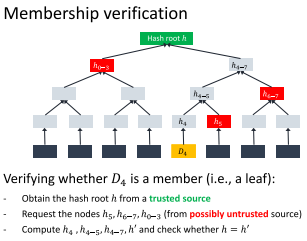
Cryptographic hash functions properties:
- **Pre-image resistant:** Given $y \in Y$, it is infeasible to find $x \in X$ such that $h(x) = y$
- **Second pre-image resistant:** Given $x \in X$, it is infeasible to find $x' \in X$ such that $x \neq x'$ and $h(x) = h(x')$
- **Collision resistance:** It is infeasible to find a pair $(x, x') \in X \times X$ such that $x \neq x'$ and $h(x) = h(x')$. This implies second-pre-image resistance

Applications:
- **Data equality:** If we know that $h(x) = h(y)$ and h is a cryptographic hash function, then it is safe to assume that $x = y$
- **Data integrity:**
  - To verify the integrity a piece of data d, we can remember its hash, `hash` = $h(d)$
  - Later, when we obtain d' from `untrusted source`, we can verify whether `hash` = $h(d')$ to check whether $d = d'$
  - Useful because `hash` is small (e.g. 256 bits)

Crypto puzzles:
- **Puzzle-friendly** For any output y, if r is chosen from a probability distribution with high min-entropy, then it is infeasible to find x such that $h(r|x) = y$
- **Search puzzle:** Given a puzzle ID id, chosen from a probability distribution with high min-entropy (min-entropy = minimum amount of randomness in a distribution or how good it is "shuffled"), and an output target $T \subseteq Y$, find a solution x such that $h(id|x) \in T$. Puzzle-friendliness implies that no solving strategy is much better than trying random values of x

1.2 Merkle Trees





Membership verification requires $\log(n)$ elements. Useful when the set of data elements is large

1.3 Digital Signatures

High-level goal: allow only one user to sign but anyone to verify the signature
API for signatures:
- (sk, pk) = `generateKeys`$(keySize)$
  - sk is the `secret key`, which the owner keeps in private
  - pk is the `public key`, which is distributed to all users
- sig = `sign`(sk, msg)
- `verify`(pk, msg, sig)

Establish Identity:
- User generates (`sk`, `pk`) key pair
- `hash`(pk) is the public name of the user
- sk allows the user to endorse a statements $stmt$ using a digital signature
- sig = `sign`$(sk, stmt)$
- anyone can verify statements endorsed by the user using `verify`$(pk,$ stmt, $sig)$

1.4 Bitcoin Basics

Distributed Consensus:
- **Termination:** Every correct process decides on some value
- **Agreement:** All correct processes decide on the same value
- **Validity:** If all correct processes propose the same value, then any correct process must decide on that value

Traditional motivation: reliability in distributed systems (node replication)

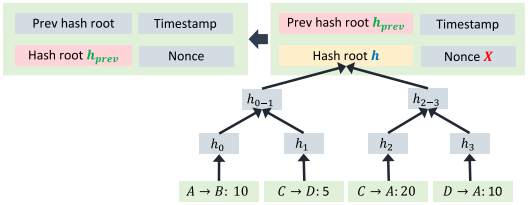
Distributed ledger via consensus:
- To make a transaction, users broadcast transactions to the network nodes
- All nodes have a sequence of all blocks of agreed transactions they have reached consensus on
- Each node has a set of outstanding transactions (to be added to a block in the blockchain)

Problems:
- Nodes may crash
- Nodes may be malicious
- Network is imperfect
  - Not all pair of nodes are connected
  - Messages may have arbitrary delays
  - No global time
  - Nodes may fail

Simplified Bitcoin consensus protocol:
1. New transactions are broadcast to all nodes
2. Each node collects new transactions into a block
3. In each round, a `random node` 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

Note: `random node` selection: Select nodes in proportion to a resource that no one can monopolize. Several schemes:
- In proportion to computing power: proof-of-work used by Bitcoin
- In proportion to ownership: proof-of-stake
- others

Proof of work:



Given:
- Previous block with hash root h_{prev}
- Merkle tree consisting of all new transactions with root h

Find:
- Find a nonce $\mathtt{X}$ such that

$$\text{hash}\left(h_{\text{prev}}|h|\mathtt{X}\right) \leq \text{difficulty}$$

Properties:
- Difficult to compute
  - Current rate in Bitcoin: about 10.2 minutes per block
  - Probability that a minor succeeds: 10.2 min / fraction of hash power Current hash power: 103,559,611,798GH/s
- Parameterizable cost
  - Nodes can adjust the target difficulty
  - Current difficulty: 15,546,745,765,529
- Easy to verify
  - Other nodes verify that hash $\left(h_{\text{prev}}|h|\mathtt{X}\right) \leq$ difficulty
- Key security assumption
  - Attacks infeasible if majority of miners weighted by hash power follow the protocol

Note to double spending: Honest nodes extend the longest valid branch

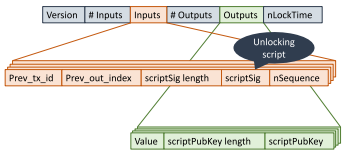
Incentivizing honest nodes to extend the longest chain
- **Block reward**. The creator of a block can:
  - Include special coin creation transaction in the block
  - Choose recipient address of this transaction
  - Value of transaction is fixed, gets halfed every 210,000 blocks
  - Block creator "receives" the reward only if the block ends up on the long-term consensus branch
- **Transaction fees**
  - Creator of transaction may choose to make output value less than input value. `Transaction fee` $= \sum inputs - \sum outputs$

1.5 Bitcoin Scripting:

- **Data**
  - public keys
  - signatures
- **Script opcodes**

  Constants: OP_0, OP_PUSHDATA_1, OP_TRUE
  Flow control: OP_IF, OP_ELSE, ...
  Stack: OP_DUP, OP_SWAP
  Arithmetic: OP_ADD, OP_NEGATE, OP_LESSTHAN, ...
  Crypto: OP_HASH160, OP_CHECKSIG
  Time: OP_CHECKLOCKTIMEVERIFY
- **Important ops and params:**
  - Checksig: takes args (signature, public key)

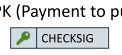
Transaction structure:



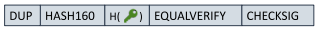
TODO unlock/locking script

Common Bitcoin scripts

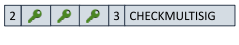
P2PK (Payment to public key)



P2PKH (Payment to public key hash)



P2MS (Pay to multi-signatures)



Bitcoin script Miscellaneous
- BitML: Language for Bitcoin smart contracts
- Oracles
- Escrow and arbitration
- Fair multi-party lotteries
- Gambling games (Poker)
- Crowdfunding

Bitcoin scripts Restrictions
- No loops
- No signature verification on arbitrary messages
- No multiplication and shifting
- No arithmetic on long numbers
- No concatenation on bitstrings

2 Ethereum and smart contracts

2.1 Smart Contracts and Ethereum

Smart contract:
- computerized transaction protocol, that executes the terms of a contract.
- Minimizes the need for trusted intermediaries.

Ethereum:
- A decentralized platform designed to run smart contracts.
- Eth smart contracts are Turing-complete.
- Transactions change the state of one or more contracts.
- Latest block stores the latest local state of all smart contracts.
- Difference to Bitcoin:
  - In Eth we own private keys to an account, not to a set of unspent transactions.
  - In this account the balance is updated, we don't simply store unspent transactions.
  - Account contains executable code.

Ethereum accounts:
- **User accounts:** Owned by some person
  - Can send transactions to transfer ether or trigger contract code.
  - Contains Address and Ether balance.
- **Contract accounts:** Owned by contract (autonomous).
  - Code execution triggered by transactions that call functions.
  - Contains Address, Ether balance, Associated contract code and persistent storage (state)

Writing smart contracts: Multiple high-level languages (Solidity, Vyper) and single low-level language (EVM bytecode).

Ethereum transactions

From	<address>	Address of the target contract
To	<address>	Address of the transaction sender
Data	<method> <arg>, ..., <arg>	ID of the method to invoke, along with arguments
Value	<amount>	Amount of Ether sent to the target contract
Gas/gas price		(see next slide)

Gas: How to prevent infinite loops in contract?
- A transaction requires gas to fuel contract execution.
- Each EVM opcode has its amount of gas, it needs in order to be executed.
- Every transaction specifies a maximum amount of gas the sender is willing to spend.
- If the contract successfully executes, the unspent gas money is refunded to the sender.
- If execution runs out of gas, the execution is reverted, but gas money is not refunded.

→ Exploit: The attacker can craft a transaction that triggers an out-of-gas exception at an arbitrary point in the execution. May create undesirable behaviors if not entire transaction is reverted.

Ether transfers:
- An ether transfer to a contract implicitly calls the receiver contract (called contract takes control over the execution).

→ Exploit: Thus an attacker may own the external contract and execute arbitrary code. In particular, he can call back the contract before returning control to the caller.
- Any user can call arbitrary functions in contracts unless they have a `require(statement)` clause. This reverts the transaction if the statement is false

Types ether transfer

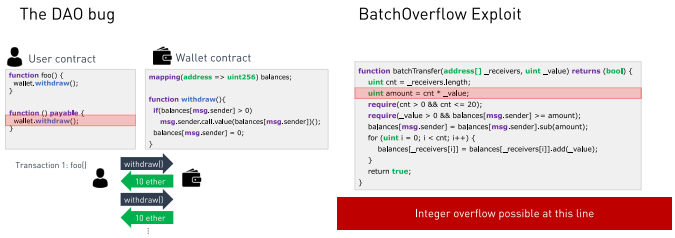
	Gas provided to the receiver	Error propagation
`transfer(amount)`	2300	yes
`send(amount)`	2300	no
`call.value(amount)()`	All remaining gas	no

Storage model In the following variableID and mappingsId are just numbers that identify the variable (e.g. 0 for variable0, 1 for variable1, ..).

- **Variables:** value stored at offset = $variableID$
- **Mappings:** mapping[key] stored at offset = $SHA3(mappingID||key)$
- **Arrays:** array[10] stored at offset = $SHA3(arrayID) + 10$

2.2 Vulnerabilities of Smart Contracts

1. **The DAO bug:** Call `withdraw()` twice, first normally, then again before balance is set to zero.
2. **BatchOverflow Exploit:**



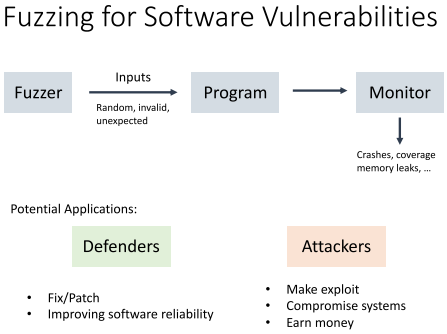
2.3 Semantics and Security Properties

- **Ethereum virtual machine (EVM) state:** $\sigma = (S, M, Q, B)$
  - Storage [S]: Persistent, inital storage is defined by the constructor.
  - Memory [M]: Not-persistent, re-initialized before executing a transaction.
  - Stack [Q]: Each element is 256 bits.
  - Block information [B]: Number, timestamp, etc. Fixed for a given transaction.
- **Transaction:** $T = (\texttt{caller}, \texttt{func}, \texttt{args})$
- **Correctness of smart contracts:** `SmartContract` $\Longrightarrow$ `Func/Sec Properties`
  - Check formal properties for all states.

3 Fuzzing

3.1 Fuzzer Basics

Fuzzer pipeline:



Functional Testing	Fuzzing
Run programs on well-structured and normal inputs (e.g., unit tests)	Test programs on abnormal inputs (e.g., randomly generated
Prevent normal users from encountering functionality errors (e.g., assertion failures)	Prevent attackers from making exploits (e.g., memory leaks)

Fuzzer Categories:

Black-Box **No information** about the program or inputs Easy-to-use and fast, but only explore shallow states

Grammar-based Generate inputs specified a **grammar**. Can reach deeper states, but assume a grammar

White-box Use **heavy-weight program analysis** to generate inputs. Can reach deeper states, but computationally expensive

Grey-box Use **light-weight** instrumentation for getting input coverage. Can reach deeper states , but need careful tuning

Fuzzing Inputs:

- **Mutation** Mutate from "good" seed inputs
  - \+ Easy to setup
  - \+ No input format specification required
  - – Seed inputs required
  - – May fail for complex functions (e.g., checksum)
- **Generation** Generate inputs from scratch, usually by a grammar
  - \+ No need for seed inputs
  - – Labor expensive to write generators
  - – Input formed specification required

American Fuzzy Loop AFL:

- AFL is a mutational, grey-box fuzzer, with (compile-time or binary-level) instrumentation for branch coverage
- Starts a queue from a set of seed inputs
- Generates new test cases by mutation from the queue:
  - – bit or byte flips
  - – addition or subtraction of small integers to bytes
  - – test cases splicing
- Add new test case to the queue if new branches are triggered.

TODO ILF and Compiler fuzzing examples

3.2 Symbolic Execution

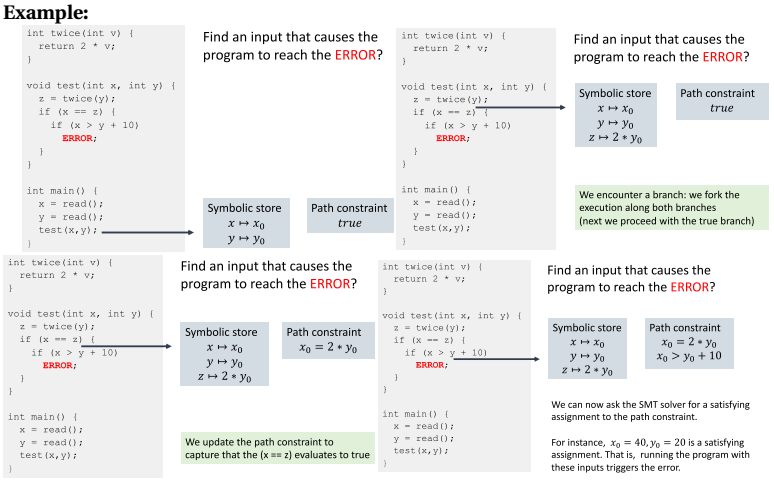
- We associate each variables with a symbolic value instead of a concrete value.
- We then run the program with the symbolic values, and obtain a constraint formula
- At any program point, we can invoke a constraint (SMT) solver to find satisfying assignments to the formula, which can be used to indicate concrete inputs reaching the program point.

Tracking: Symbolic Execution tracks the following two formulas:

- symbolic store: Tracks possible values of variables
- path constraint: Tracks history of all branches taken so far

Symbolic state is then defined as the conjunction of these two formulas

Example:



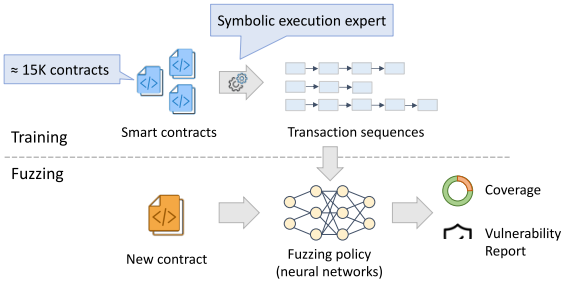
In Practice:

- Many challenges:
  - – Path explosion (exponential in number of branches)
  - – Constraint solving (e.g., non-linear and hash constraints)
- Real-world symbolic execution fuzzers:
  - – Stanford KLEE (http://klee.github.io/)
  - – NASA's Java PathFinder (http://javapathfinder.sourceforge.net/)
  - – Microsoft SAGE
  - – UC Berkeley's CUTE

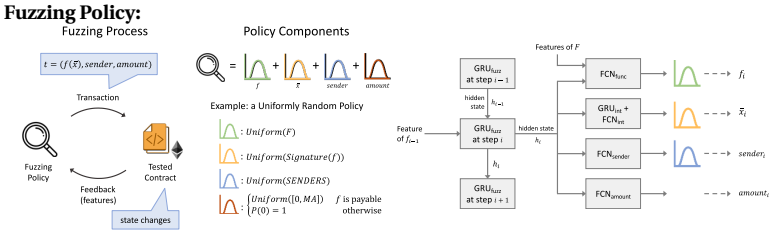
Comparison:

	Random Fuzzing	Symbolic Execution
Speed	✓ Fast	✗ Slow
Inputs	✗ Ineffective	✓ Effective
Coverage	Low	Low

3.3 Imitation Learning based Fuzzer ILF



Fuzzing Policy:



- **GRU**: Gated Recurrent Units
  - Can deal with variable length sequential inputs with hidden states
  - A natural fit for our setting for handling sequence of transactions
- Features of f_{i-1} could be Coverage, opcodes, function name. (can be dynamic)
- Fully Connected Networks (**FCN**): Standard network with linear layers and ReLU activation functions
- Softmax: Normalizes function scores (output of $\textbf{FCN}_{func}$) to probability distrubtion

Neural Network description:

- **Function:** FCN_{func} + Softmax
- **Arguments:** GRU_{int} + FCN_{int}. Initial distribution from expert. Challenges:
  - Functions have a variable length of arguments
  - Search space of arguments is large (e.g., integers of 256 bits)
- **Senders:** FCN_{sender}. Distribution over 5 predefined senders
- **Amount:** FCN_{amount}. Distribution over 50 seed amount values from expert

Symbolic Execution Expert:

- Standard symbolic engines (breadth-first style):
  - Operate on symbolic blockchain state and generate symbolictransactions.
  - Only scale to limited depth (e.g. 3)
- Our expert (depth-first style with revisits):
  - Operates on concrete blockchain state and generate concrete transactions.
  - Scales to large depth (e.g. 30) and large training set.
  - Algorithm:
    1. Start on initial blockchain state b_{init}
    2. Greedy search for transaction t with highest coverage
    3. Execute transaction t to get to next blockchain state b_1
    4. Revisit blockchain states b_i and take different transactions to improve code coverage
    5. Repeat

Miscellaneous

- **Training NN Fuzzing Policy:** Use Cross-Entropy Loss on inferred transactions and transactions by expert. aka standard supervised learning
- **ILF System: Reports the following:**
  - Instruction coverage.
  - Basic block coverage.
  - Locking: The contract cannot send out but can receive ether.
  - Leaking: An attacker can steal ether from the contract.
  - Suicidal: An attacker can deconstruct the contract.
  - Block Dependency: Ether transfer depends on block state variables.
  - Unhandled Exception: Root call does not catch exceptions from child calls.
  - Controlled Delegatecall: Transaction parameters explicitly flow into arguments of a delegatecall instruction.

4 Datalog and static analysis

4.1 Stratified Datalog

Datalog:

- Declarative logic-programming language.
- Useful for expressing recursive queries.
- solid formal foundations.
- Used in:
  - Security Analysis
  - Network config analysis
  - Access control

Datalog atoms: Datalog consists of a variety of atoms.

Datalog atoms

Constants $C = \{alice, bob, carol, dave, eve, frank, gwen\}$
Variables $V = \{X, Y, Z\}$
Predicates $P = \{parent(\_,\_), anc(\_,\_)\}$

Only constants

Ground atoms
$G_{P,\mathcal{C}} = \{parent(alice, alice), anc(alice, bob), \dots\}$
Atoms
$A_{P,\mathcal{C},\mathcal{V}} = \{parent(X, X), anc(alice, X), \dots\}$

May contain variables

Datalog programs:

- A set of rules of the form $a \leftarrow l_1, ..., l_n$
- Where $l_1, ..., l_n$ are literals of the form a or $\neg a$
- A program is well-formed if for any rule, all variables that appear in the head (left of arrow) also appear in the body (right of arrow). Plus the variables in the head are not allowed to be negated.
- Predicates are not allowed to by cyclic and appear in negative literals.
- A program is positive if its rules do not contain negative literals.
- The semantics of a positive Datalog program P is the least-fixed point of it.
- For any positive Datalog program the consequence operator T_P is monotone, but not for programs with negation.

Consequence Operator T_P / J:

- $T_p(I) = \{\sigma(a) | a \leftarrow l_1, ..., l_n \in P, \exists \sigma : \forall i \in [1, ..., n] : I \vdash \sigma(l_i)\}$
  where $I \vdash l_i$ if $l_i = a$ and $a \in I$
  and $I \vdash l_i$ if $l_i = \neg a$ and $a \notin I$
- Note that $(J, \subseteq, \cup, \cap)$ is a complete lattice.
- **Least-fixed point** lfp_{T_P} can be computed by iteratively applying the consequene operator until reaching a fixed-point.

Stratified Datalog programs:

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

→ For each stratum, compute the lfp that contains the lfp of the previous stratum.

Stratified Datalog programs

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

Stratum P_1
$r(X) \leftarrow q(X)$
$q(X) \leftarrow !p(X), r(X)$

Stratum P_0
$p(b) \leftarrow p(a)$
$p(a) \leftarrow$

Semantics (informally): For each stratum, compute the lfp that contains the lfp of the previous stratum

4.2 Static analysis with Datalog

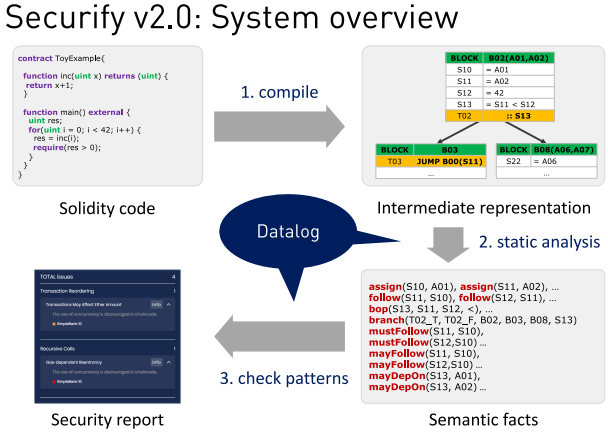
Effective callback freedom EECF: A contract is EECF if for any execution with external callbacks there exists an equivalent execution without external callbacks (which starts in the same state and reaches the same final state).

→ In practice, developers enforce the following security property: Do not perform any stat changes after calls to external contracts.

Checking security properties:

- Usually we cannot automatically find all calls where a certain property does not hold, because smart contracts are Turing-complete.
- However, when contracts violate/satisfy a security property they often violate/satisfy a simpler property. (E.g. instead of EECF check if there are writes after call.value())

Securify: v2.0



- Securify compiles Solidity to IR (Intermediate Representation)
- Bytecode is too low-level and lacks semantic information (No function boundaries, no storage model)
- Source code (Solidity) is too rich/changes often. This leads to a complex analysis.
- **Basic blocks:**
  - Sequence of statements without branching.
  - Take Arguments
  - All variables are in single static assignment (SSA) form.
  - End with a transfer to next blocks
- **Transfers:** Goto, Jumps, Branches. Args and return values passed to next block
- **Statements:**
  - Static Single Assignment (SSA): each variable is treated as a variable that stores the statement's result.
  - Three address form: t0 = t1 op t2

Securify: Inferring semantic facts
- Scalable inference of semantic facts using Datalog.
- IR → Datalog input → Datalog fixpoint.

Relevant semantic facts

Control-flow analysis	
$mayFollow(L_1, L_2)$	Instruction at label L_1 may follow that at label L_2
$mustFollow(L_1, L_2)$	Instruction at label L_1 must follow that at label L_2
Data-flow analysis	
$mayDepOn(X, T)$	The value of X may depend on tag T
$eq(X, T)$	The values of X and T are equal
$detBy(X, T)$	For different values of T the value of X is different

For real-world contracts, Securify infers 1 - 10M such facts

Taint analysis:
- Taint analysis is a popular method which consists to check which variables can be modified by the user input. All user input can be dangerous if they aren't properly checked.
- **Call-site sensitivity:** Function arguments are tainted as they are controlled by the user → `mayDepOn(arguments, tainted)`
- Context tracks a bounded history of the most recent function calls (e.g. context = [func1, func2,...]). → `mayDepOn(context, variable, tainted)`

Security patterns language:
- A security pattern is a logical formula over semantic predicates e.g: $\varphi = mayDepOn(X, Y) | mustFollow(L, L) | \neg\alpha | ...$
- We need to convert a Security property e.g. "No state changes after call instructions" to a suitable violation pattern for securify.
- Securify then creates a Security report, where all unsafe calls are reported as either violations or warnings.

Benefits of static analysis using Datalog:
- Declarative (concise spec of the analysis)
- Modular (can merge the rules of multiple analyses)
- Scalable (can leverage existing Datalog solvers)

5 Functional specification

5.1 Introduction

Smart contract vulnerabilities:
- Unexpected ether flows
- Unprivileged writes
- Use of unsafe inputs
- Reentrant method calls
- Transaction reordering
- Arithmetic overflows

Unknown unknowns: Some "cool" quote from Christoph Jentzsch: *We believe more security audits or more tests would have made no difference. The main problem was that reviewers did not know what to look for.*

The problem:

How do vulnerabilities manifest?	What can we do about this?
Unexpected behavior	1. Specify expectations precisely 2. Verify contract wrt. specification

TODO WTF?

5.2 Property Behaviors

Definitions:
- **Execution** = sequence of function calls
- **Behavior** = corresponding sequence of (message, state) pairs
- The set of possible **execution trees** is defined by the following grammar:
  Exec ::= Start$^\infty$ Start ::= Call Call::= Msg [Cmd*] Cmd ::= Call | ...
- A **complete call** of a bundle of contracts is a **subtree** of a valid execution tree such that:
  1. the root is a **call** to the bundle
  2. **not nested** in other such call
- Every complete call of the bundle gives rise to an **observation of the bundle** which records:
  1. the **message** used to call the bundle
  2. the **bundle state** right after the call
- A **property** w.r.t. **the bundle** is merely a set P of infinite observation sequences α
  Examples:
  P_1 = {α : all investments in α get deposited to Escrow}
  P_2 = {α : all withdrawals in α happen only after the Crowdsale was closed}
  P_3 = {α : the Crowdsale is eventually closed in α}

Safety vs Liveness

Safety

A safety property requires that something **bad never happen**, e.g., a withdrawal never happens before a closure.

Formally:
P is a **safety** property iff for all infinite α

$$\alpha \notin P \Leftrightarrow \exists \text{ bad} \prec \alpha \forall \beta \in P : \text{bad} \not\prec \beta$$

Safety ~ correctness.

Liveness

A liveness property requires that something **good happens often enough**, e.g., the crowdsale eventually closes.

Formally:
P is a **liveness** property iff for all finite α

$$\exists \beta : \alpha\beta \in P$$

Liveness ~ delivery

Examples:
P_1 = {α : all investments in α get deposited to Escrow}
- Safety (depends on English interpretation)

P_2 = {α : all withdrawals in α happen only after the Crowdsale was closed}
- Safety

P_3 = {α : the Crowdsale is eventually closed in α}
- Liveness

5.3 Linear Temporal Logic (LTL)

Introduction Consider the property: {α : all withdrawals in α happen only after the Crowdsale was closed}
- Classical logic definition:
  $\varphi(\alpha) \equiv \forall t \in \mathbb{N} : \text{fun}\,[\alpha|t] = \text{withdraw} \rightarrow \exists s \leqslant t : \text{fun}\,[\alpha|s] = \text{close}.$
- LTL definition
  $\varphi \equiv \Box(\,\text{fun} = \text{withdraw} \rightarrow \text{Gfun} = \text{close}\,)$

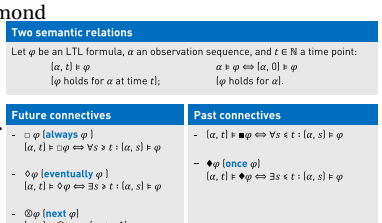
5.4 LTL Syntax

Terms
- **Variables** (a.k.a. flexible variables) fun, raised, goal, ...
- **Constants** (a.k.a. rigid variables) $u, v, w, \ldots, 0, 1, 2, \ldots$
- Application of **function symbols** raised + goal, ...

Formulas
- Application of **relation symbols** fun = withdraw, raised <= goal
- Application of **logical connectives** $\varphi \wedge \psi, \varphi \vee \psi, \varphi \rightarrow \psi, \ldots$
- Application of **temporal connectives** $\Box\varphi, \diamond\varphi, \blacksquare\varphi, \varphi, \ldots$

TODO filled diamond

filled square = **so far**



Constants (a.k.a. rigid variables)

A constant u may vary in time only w.r.t. surrounding temporal connectives:
LTL: $\Box : (cl = u) \wedge \otimes(cl = u + 1)]$
Classical: $\forall t \in \mathbb{N} : \exists u : \text{cl}[t] = u \wedge c\lfloor[t + 1] = u + 1$

Variables (a.k.a. flexible variables)

A variable u may vary in time freely:

LTL: $\Box\exists u : (\text{cl} = u) \wedge \otimes(cl = u + 1))$
Classical: $\forall t \in \mathbb{N} : \exists u : \text{cl}[t] = \text{u}[t] \wedge \text{cl}[t + 1] = \text{u}[t + 1] + 1$

Logical connectives

$\neg(\varphi \wedge \psi) \Leftrightarrow (\neg\varphi) \vee (\neg\psi)$
$\neg\forall u.\ \varphi \Leftrightarrow \exists u \neg\varphi$
$\neg(\varphi \vee \psi) \Leftrightarrow (\neg\varphi) \wedge (\neg\psi)$
$\neg\exists u.\ \varphi \Leftrightarrow \forall u \neg\varphi$

Duality: De Morgan's laws

Temporal connectives

$\neg\Box\varphi \Leftrightarrow \Diamond\neg\varphi$
$\neg\blacksquare\varphi \Leftrightarrow \blacklozenge\neg\varphi$
$\neg\Diamond\varphi \Leftrightarrow \Box\neg\varphi$
$\neg\blacklozenge\varphi \Leftrightarrow \blacksquare\neg\varphi$

Theorems:
If φ is a non-temporal formula (no temporal connectives), $\Box\varphi$ defines a safety property.
If φ be a past formula (no future connectives), $\Box\varphi$ defines a safety property. Also known as **Canonical Safety**

6 Formal Verification

Goal: For a contract with a set of behaviours B and for a safety property P, prove that B ⊆ P. Reduce this to assertion verification.

Verification Recipe: Start with a bundle of contracts and a canonical safety specification $\Box\varphi$.

1. Convert the bundle to a program Π that generates all bundle behaviors B.
2. Turn φ to an assertion by instrumenting Π with variables tracking past values (now assertion = non-temporal formula).
3. Use standard assertion verification techniques to verify φ.

State Transition Systems: The mathematical perspective of the verification.

Definition

A *state transition system* (STS) consists of

- S is a set of *states* (observations);
- $S_0 \subseteq S$ is a set of *initial states*.
- $T \subseteq S \times S$ is a *transition relation*;

Definition

A *behavior* is a sequence $\alpha \in S^\omega$ s.t.:

- $\alpha_0 \in S_0$;
- $(\alpha_i, \alpha_{i+1}) \in T$ for $i = 0..$;

The verification problem(STS)

Given:

1. STS (S, S_0, T) with behaviors $\alpha \in B$;
2. state property $I \subseteq S$;

Prove:

$$\forall a \in B : \forall i \in \omega : a_i \in I.$$

Definition

A property I for which the above holds is called an invariant of the STS.

Invariants:

- **Invariant:** definition in image above.
- proving invariants is too difficult, thats why we need inductive invariants.
- **Inductive Invariants:** An inductive invariant of a STS is a state property I such that $S_0 \subseteq I$ and $T[I] \subseteq I$.
- **Theorem:** Every inductive invariant is an invariant.

How to prove inductiveness automatically?

- Use the strongest postcondition operator **sp** : Program x Assertion → Assertion.

Definition

- Let Π be a program encoding a transition relation $T \subseteq S \times S$.
- Let φ be an assertion defining a state property $P \subseteq S$.

The *strongest postcondition* $\mathbf{sp}(\Pi, \varphi)$ is any assertion defining the set of states:

$$T[P] = \{s \in S : \exists s_0 \in P : (s_0, s) \in T\}.$$

1. a) A state transition system (S, S_0, T) is given by two progtams `init` and `step`:
   `init; while true { step; }`
   b) A candidate invariant I is given by a formula φ.
2. Use and automated theorem prover to prove the validity of the formulas:
   a) **sp** (`init, true`) → φ; (encodes $S_0 \subseteq I$)
   b) **sp** (`step,`) $\varphi \rightarrow \varphi$; (encodes $T[I] \subseteq I$)

→ Example of this process on slide 15.

Computing $sp(\Pi, \varphi)$ by symbolic execution:

Execute all program paths symbolically collecting a path assertion per program path:

se : Program × (Variable → Term) → $\mathcal{P}$(Assertion)

Definition

- Let Π be a program.
- Let $x, y, z, \ldots$ be the program variables of Π.

Symbolic execution of Π with *symbolic store* $\Sigma_0 \equiv x = x_0 \wedge y = y_0 \wedge z = z_0 \wedge \ldots$ satisfies:

$$\bigvee \mathbf{se}(\Pi, \Sigma_0) \equiv \mathbf{sp}(\Pi, \Sigma_0)$$

Question

Express $\mathbf{sp}(\Pi, \varphi) \equiv \exists x_0, y_0, z_0, \ldots : \varphi(x_0/x, y_0/y, z_0/z, \ldots) \wedge \bigvee \mathbf{se}(\Pi, \Sigma_0)$

- If Π contains loops then, #paths = ∞, requiring more sophisticated techniques.

→ Example of this process on slide 18.

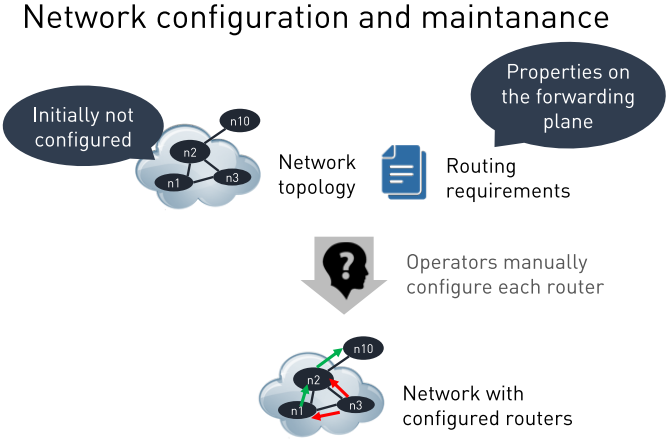
How to prove non-inductive variants?

- Find an inductive strengthening I manually: I is inductive and $I \rightarrow \varphi$.
- and prove analogously the strengthening I automatically.

7 Network verification

7.1 Introduction

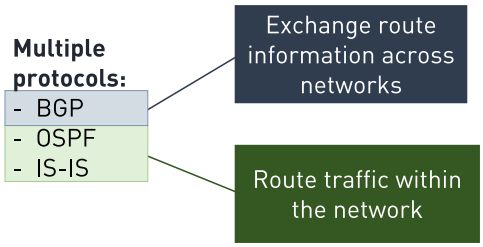
- Each router:
  - Runs multiple routing protocols (e.g. OSPF, BGP, ISIS)
  - Stores a configuration (parameters used by the protocols)
- Forwarding table:
  - Defines the next hop for any packet
  - Collectively, the forwarding tables of all routers define the forwarding plane



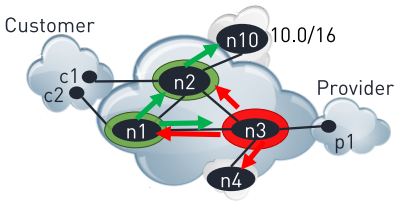
Why network configuration is hard:

We have multiple protocols (BGP, OSPF, IS-IS), each with their own low-level configurations (BGP-Policies, Link costs, Static routes). Also the protocols themselves interact with each other (Preferences like: static over OSPF; Dependencies: BGP relies OSPF)

Example



OSPF: Forward traffic along the least-cost path to the destination




```
// Configuration of n1
1. ospf n2 cost 10
2. ospf n3 cost 5
```

```
// Configuration of n2
1. ospf n1 cost 10
2. ospf n3 cost 5
```

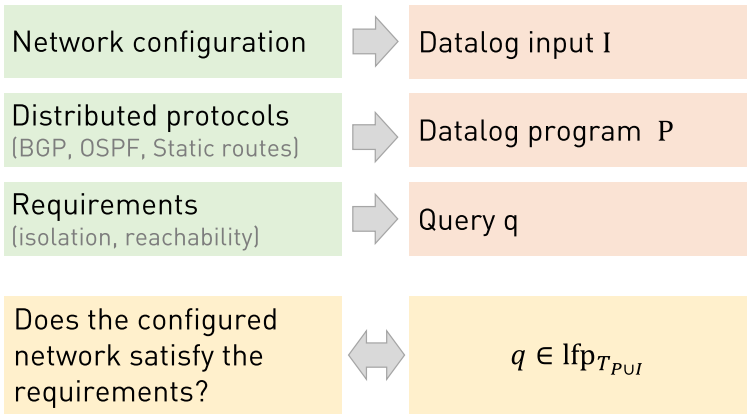
```
// Configuration of n3
1. ospf n1 cost 5
2. ospf n2 cost 5
3. static route 10.0/16 drop
```

Are our requirements enforced?

- 10.0/16 is reachable from Customer
- 10.0/16 is unreachable from Provider, n4

Multi-path inconsistency: Traffic from n1 to 10.0/16 is non-deterministically delivered / not-delivered

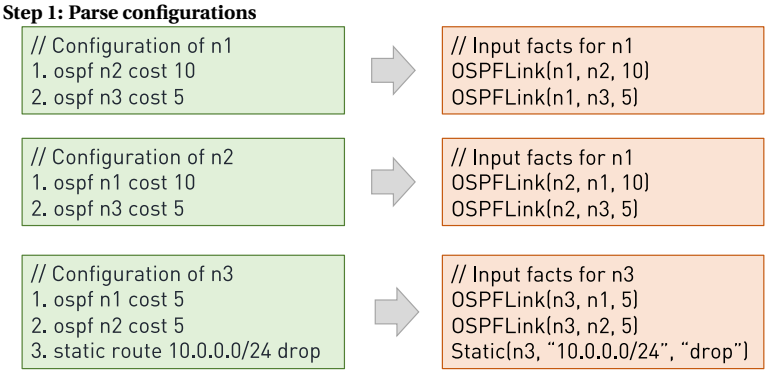
7.2 Analysis of Network Configurations via Datalog



7.3 Batfish

- Network configuration analyzer
- Available at http://www.batfish.org/
- Has found real bugs in real networks
- Three steps:
  1. Parse configurations (to derive input facts)
  2. Compute forwarding plane (by computing a fixedpoint)
  3. Check for violations (by querying the fixed-point)

Step 1: Parse configurations



Step 2: Compute forwarding plane

- Compute OSPF routes
  - Input: OSPFLink(Router1, Router2, Cost)
  - Wanted Output: BestOSPFRoute(Router, Network, NextHop)

```
OSPFroute(Router, Network, Network, Cost) <-
        OSPFLink(Router, Network, Cost).

OSPFroute(Router, Network, NextHop, Cost) <-
        OSPFLink(Router, NextHop, Cost1),
        OSPFroute(NextHop, Network, _, Cost2),
        Cost = Cost1 + Cost2, Cost < MAX_COST.

nonMinNextHop(Router, Network, NextHop, Cost1) <-
        OSPFroute(Router, Network, NextHop, Cost1),
        OSPFroute(Router, Network, _, Cost2),
        Cost1 > Cost2.

bestOSPFroute(Router, Network, NextHop) <-
        OSPFroute(Router, Network, NextHop , Cost),
        !nonMinNextHop(Router, Network, NextHop, Cost).
```

- Compose protocols (by given preference)
  - Inputs:
    - bestOSPFRoute(Router, Network, NextHop, Cost)
    - static(Router, Network, NextHop)
    - localPref(Router, Protocol, Preference)
  - Output: fwd(Router, Network, NextHop)

```
route(router, Network, Protocol, NextHop) <-
        bestOSPFroute(router, Network, NextHop),
        Protocol="OSPF"
route(router, Network, Protocol, NextHop) <-
        static(router, Network, NextHop) ,
        Protocol="static"
nonPrefroute(router, Network, Protocol1) <-
        route(router, Network, Protocol1, _), localPref(router, Protocol1, Pref1),
        route(router, Network, Protocol2, _), localPref(router, Protocol2, Pref2),
        Pref1 < Pref2.
fwd(router, Network, NextHop) <-
        route(router, Network, Protocol, NextHop),
        !nonPrefroute(router, Network, Protocol).
```

Step 3: Analyze forwarding plane

For example: Detect Multi-path Inconsistencies with the following

```
delivered(Router, Network) <-
        fwd(Router, Network, Network).

nondelivered(Router, Network) <-
        fwd(Router, Network, "drop").

delivered(Router, Network) <-
        fwd(Router, Network, NextHop),
        delivered(NextHop, Network).

nondelivered(Router, Network) <-
        fwd(Router, Network, NextHop),
        nondelivered(NextHop, Network).

inconsistent(Router, Network) <-
        delivered(Router, Network),
        nondelivered(Router, Network).
```

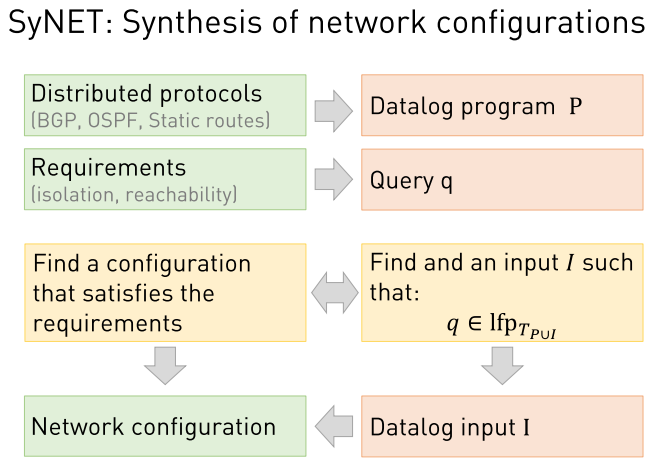
More properties:

- Paths: Packets for traffic class tc must follow the path `r1 -> ... -> r10`
  `holds <- fwd(r1, tc, r2), ..., fwd(r9, tc, r10)`
- Traffic isolation: The paths for two distinct traffic classes `tc1` and `tc2` do not share links in the same direction
  `holds <- ∀ R1, R2. fwd(R1, tc1, R2) => (!fwd(R1, tc2, R2))`
- Reachability: Router can reach a given traffic class
- Loop-freeness: The forwarding plane has no loops

8 Network Synthesis

Program Synthesis: Learning a function from specifications (e.g. examples). In this section we cover the synthesis of correct network configurations through incorporation of powerful SAT/SMT solvers. Two such network conficuration synthesisers are SyNET and NetComplete.

8.1 SyNET



Datalog rules vs SMT constraints: Datalog rules are logical constraints that can be fed to an SMT solver.

- constraints on datalog inputs (i.e. containing no derived datalog rules) can be converted directly to logical constraints.
- Derived datalog rules need a bit more work (unrolling) because the datalog derivation works different than the logical implication.

Input synthesis for Datalog via SMT:

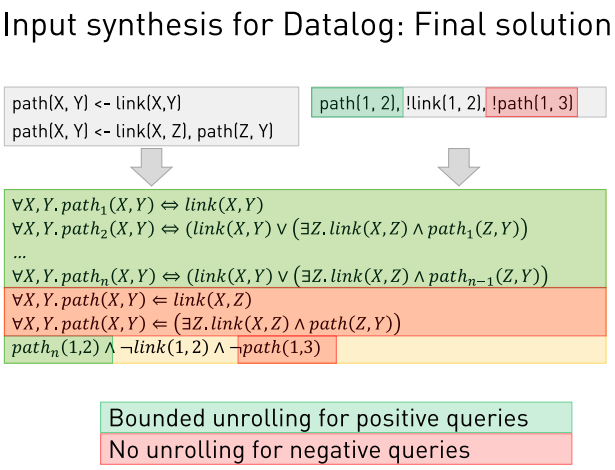
1. Encode Program P into SMT constraints, which capture the fixed-point computed by P for a given input I.
2. Encode query q as assertions that must hold on the fixed-point.
3. Get a model M that satisfies the conjunction of the above constraints.
4. Derive input I from M by checking which atoms are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

→ But be careful: In Datalog, given a rule $p \leftarrow q$, p is derived iff q is true. However in logic, the constraing $p \Longleftarrow q$ is satisfied if p is true an q is false! Switch implication symbol to a if and only if symbol and unroll the datalog rule - make a formula for each time applying the consequence operator)

→ What about negative queries? Unrolling the Datalog rules works for positive queries beacause, by monotonicity of the consequence operator, any atom derived in a given step of the consequence operator is guaranteed to remain in the fixed point. For negative queries e.g. !path(1,3) we don't need to unroll, simply use the implication symbol.

→ What about stratified Datalog? Can be done by iteratively synthesizing an input for

each stratum, starting from the highest stratum, going towards the lowest one. The iterative process may require backtracking if we end up with no possible solution.



Challenges: Key challenge is the scaling of synthesis for OSPF protocol (takes much more computation time)

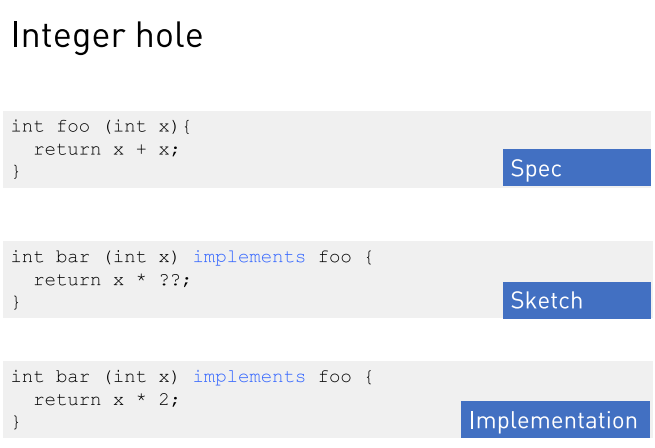
8.2 NetComplete

Sketching: A concept to express intent

- Allow users to express insight by defining the hypothesis space and bias the search.
- Machine helps with low-level reasoning

Expressing Intent Two ways to express intent and control hypothesis space:

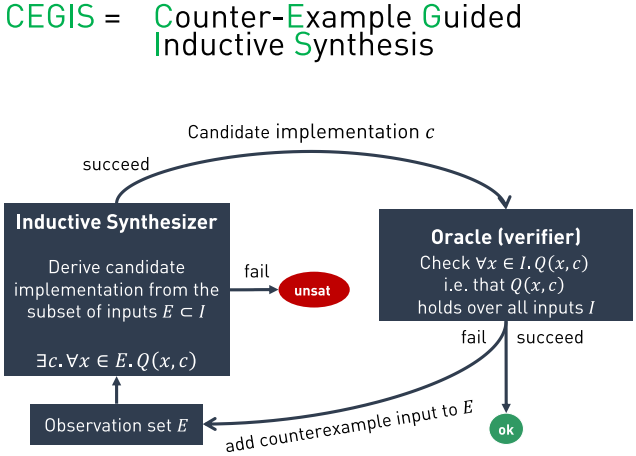
- Specifications: Define what function you actually want:
  - Assertions: `assert x > y;`
  - Concrete Examples: `x = 5 or x = 6`
  - Function Equivalence: `blockedMatMul(Mat a, Mat b) implements matMul`
- Holes: To be instantiated by the synthesizer. Fragment must come from a set defined by the user. Holes define the hypothesis space.



Synthesizing functions: Look at slides 49ff. (chume nöd ganz drus)

1. Turn holes into special inputs of the function (Control inputs C)
2. Constraining the set of controls. Constraints are collected into a predicate Q(x,c)
3. Synthesize function: Learning reduces to constraint satisfaction

Counter-example guided inductive synthesis (CEGIS): CEGIS aims to find the right small set of examples S so that if we learn a function F satisfying S, then F is likely to be correct on all (potentially infinite set of) examples



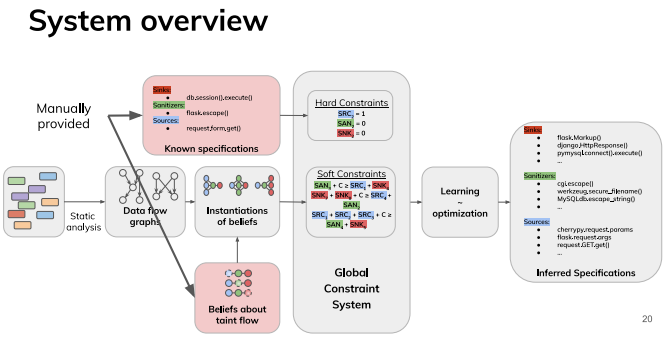
Efficient OSPF synthesis using CEGIS: Define logical constraints that capture the OSPF requirements. Find cost assignment f such that requirements hold. Formula given to the solver.

9 Big Code

Big Code: Create new software tools that leverage massive codebases to solve problems beyond what is possible with traditional techniques. Usually the more specifications (understanding of how program code works) to standard models the better they get. The goal is to learn specifications automatically from large codebases.

9.1 Example: Injection Attacks

Tools to find Injection Attacks Use static taint analysis, but what about missing (not detected) sinks, sanitizers and sources?



9.2 Pointer Analysis

Problems with Pointer Analysis:

- Analysis of code for different libraries.
- Dynamic Runs (e.g. complex runtime behaviours of java.sql)
- Need knowledge how to analyze corner cases and libraries

→ Use ML for a scalable solution to pointer analysis!

10 Exercises

Revert vs. Require vs. Assert:

- Revert: Refund remaining Gas, Revert state and return a value, but does not take condition as argument.
- Require: Same as Revert but takes condition as argument.
- Assert: Burn all remaining Gas, Revert state.

Call vs. Send vs. Transfer

- Call: No error propagation but returns false, not safe against reentrancy.
- Send: No error propagation but returns false if there is an error. Only gas for one single event.
- Transfer: Error propagation, Only gas for single event