CHAINSECURITY

SRILAB

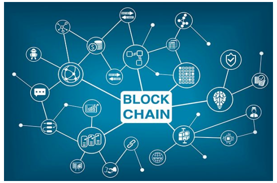
Smart Contract Security: Testing, Verification, and Data Privacy

Dr. Petar Tsankov
@ptsankov

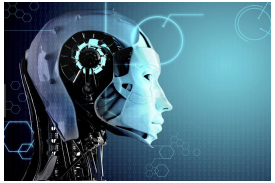
Co-founder and Chief scientist, ChainSecurity
Senior researcher, ETH Zurich

SRILAB ETH Zürich

Security @ SRI Lab:



Blockchain security



Safety and security of AI

Security and privacy

CHAINSECURITY Spinoff ETH zürich

Next-generation blockchain security with automated reasoning

https://chainsecurity.com
@chainsecurity

ethereum foundation WEB3 zilliqa

Smart contract security @ SRI

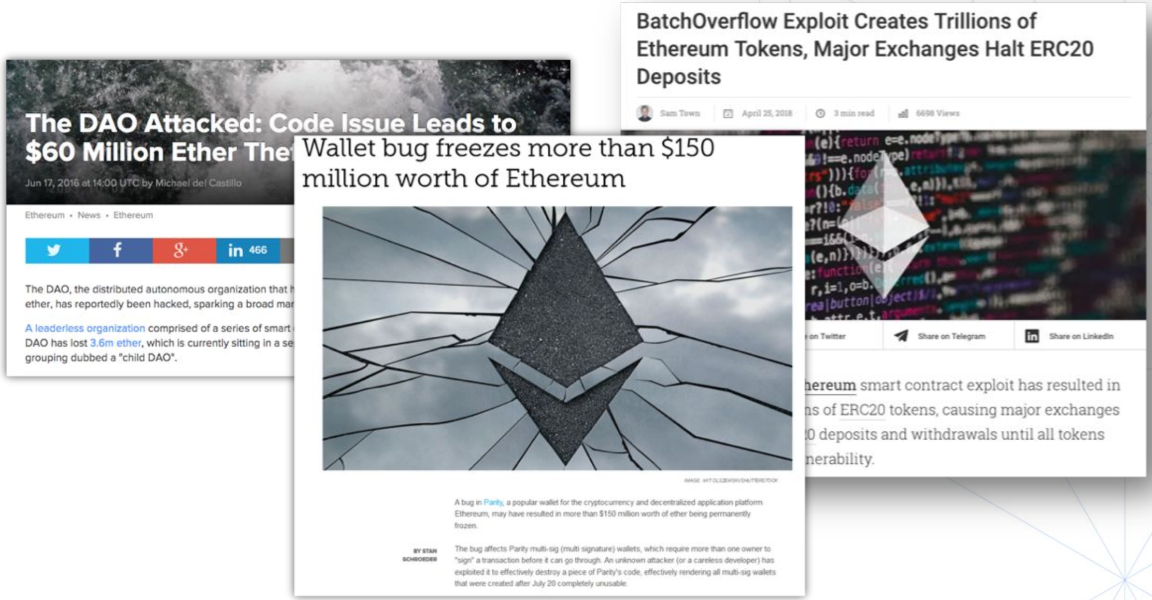
Security goal	Avoid generic vulnerabilities; Enforce data privacy; Ensure functional correctness
Technique	Static analysis; Datalog; Type checking; Predicate abstraction; Temporal logic; Fuzzing; Symbolic execution; Zero-knowledge; Reinforcement learning
System	SECURIFY — ACM CCS'18; ILF — ACM CCS'19; VerX — IEEE S&P'20; zkay — ACM CCS'19

Smart contract security @ SRI

Security goal	Avoid generic vulnerabilities	Ensure functional correctness	Enforce data privacy	
Technique	Static analysis, Datalog, Symbolic execution	Zero-knowledge, Temporal logic	Type checking, Fuzzing	Predicate abstraction, Reinforcement learning
System	SECURIFY ACM CCS'18	ILF ACM CCS'19	VerX IEEE S&P'20	zkay ACM CCS'19

Many well-known vulnerabilities

- Unexpected ether flows
- Unprivileged writes
- Use of unsafe inputs
- Reentrant method calls
- Transaction reordering



Smart Contract Weakness classification registry: https://swcregistry.io

Many well-known vulnerabilities

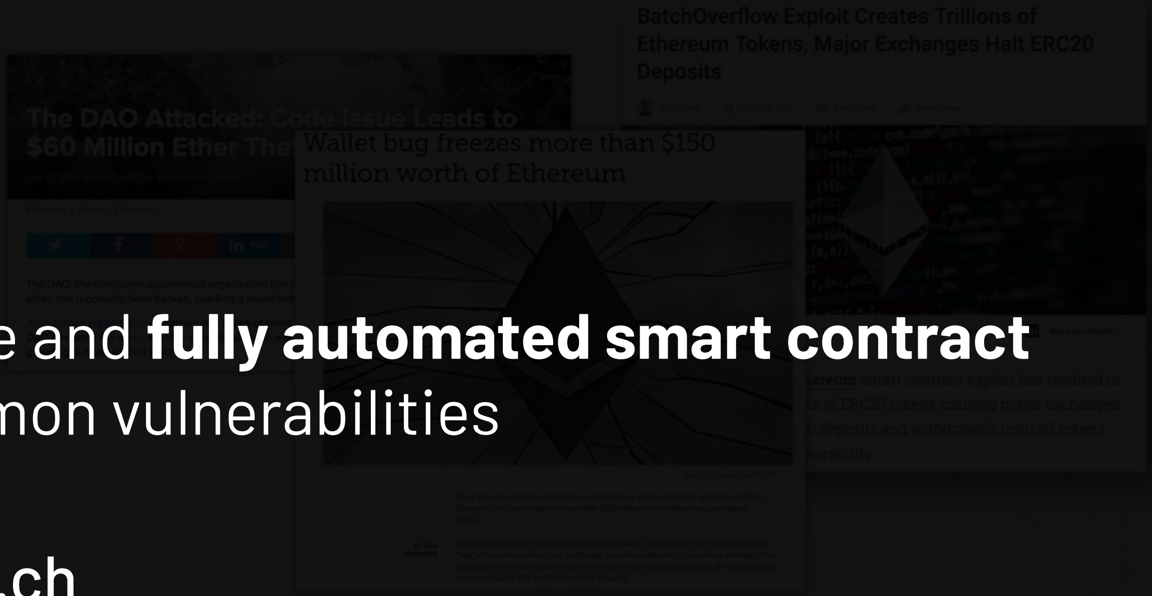
Unexpe…

Unprivileged writes

Use of u…

Reentrant method calls

Transaction reordering



SECURIFY

The first scalable and **fully automated smart contract verifier** for common vulnerabilities

https://securify.ch

Smart Contract Weakness classification registry: https://swcregistry.io

How does it work?

Declarative static analysis in Datalog

Check sufficient conditions for safety and violation

```
push 0x04
dataload
push 0x08
jump
jumpdest
stop
jumpdest
```

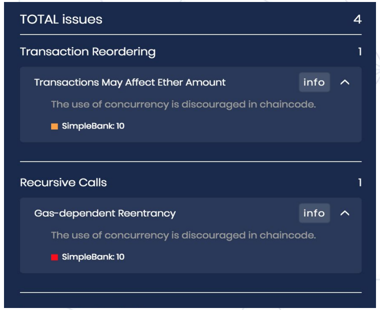
EVM

```
1: a = 0x04
2: b = load(a)
3: abi_00(b)
4: stop
   abi_00(b)
5:   c = 0x00
6:   sstore(c,b)
```

IR

```
assign(1,a,0x04)
follow(2,1)
mayDepOn(b,a)
load(2,b,a)
follow(3,2)
follow(5,3)
...
```

Semantic facts



Security report

Securify: Practical Security Analysis of Smart Contracts

P.Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, M. Vechev

ACM CCS 2018

NEWS: Securify 2.0 coming out soon!

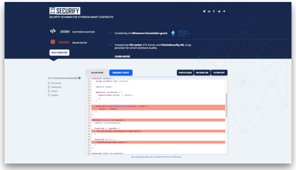
CHAINSECURITY

SRILAB

Impact

Used daily by security auditors

1K+ subscribers



https://securify.ch

https://github.com/eth-sri/securify

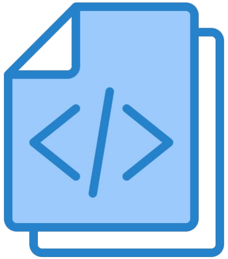
Grants

ethereum foundation

Blockchain security @ SRI

Security goal	Avoid generic vulnerabilities; Enforce data privacy; Ensure functional correctness
Technique	Static analysis; Datalog; Type checking; Predicate abstraction; Fuzzing; Temporal logic; Symbolic execution; Zero-knowledge; Reinforcement learning
System	SECURIFY (ACM CCS'18); ILF (ACM CCS'19); VerX (IEEE S&P'20); zkay (ACM CCS'19)

Functional correctness



Smart contract

$\models$

Functional specification

Correctness of ERC20

```
mapping(addr => uint) balances;
uint totalSupply;
addr owner;

function mint();
function changeOwner(addr o);
```

$\models$

1. The sum of all balances equals the total supply
2. Only the owner can increase the total supply of tokens
3. ...

Smart contract

Functional requirements

Step 1: *Formalize* requirements

"The sum of balances equals the total supply"

Formal property

```
SUM(ERC20.balances) == ERC20.totalSupply
```

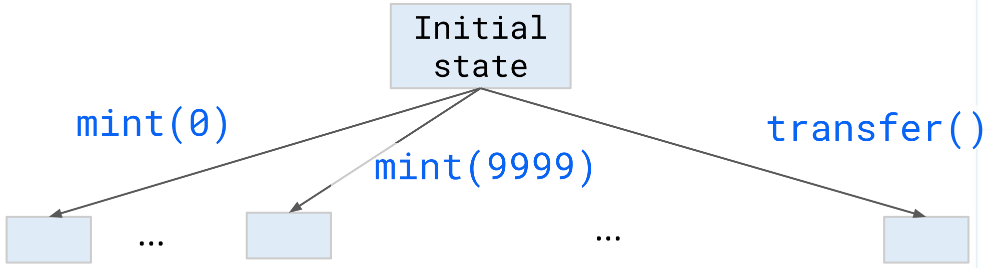
```
balances[0x10] = 50
balances[0x20] = 50
totalSupply = 100
```

Property holds

```
balances[0x10] = 100
balances[0x20] = 50
totalSupply = 100
```

Property does not hold

Step 2: ***Check*** formal property for all states



Step 2: *Check* formal property for all states

State transitions

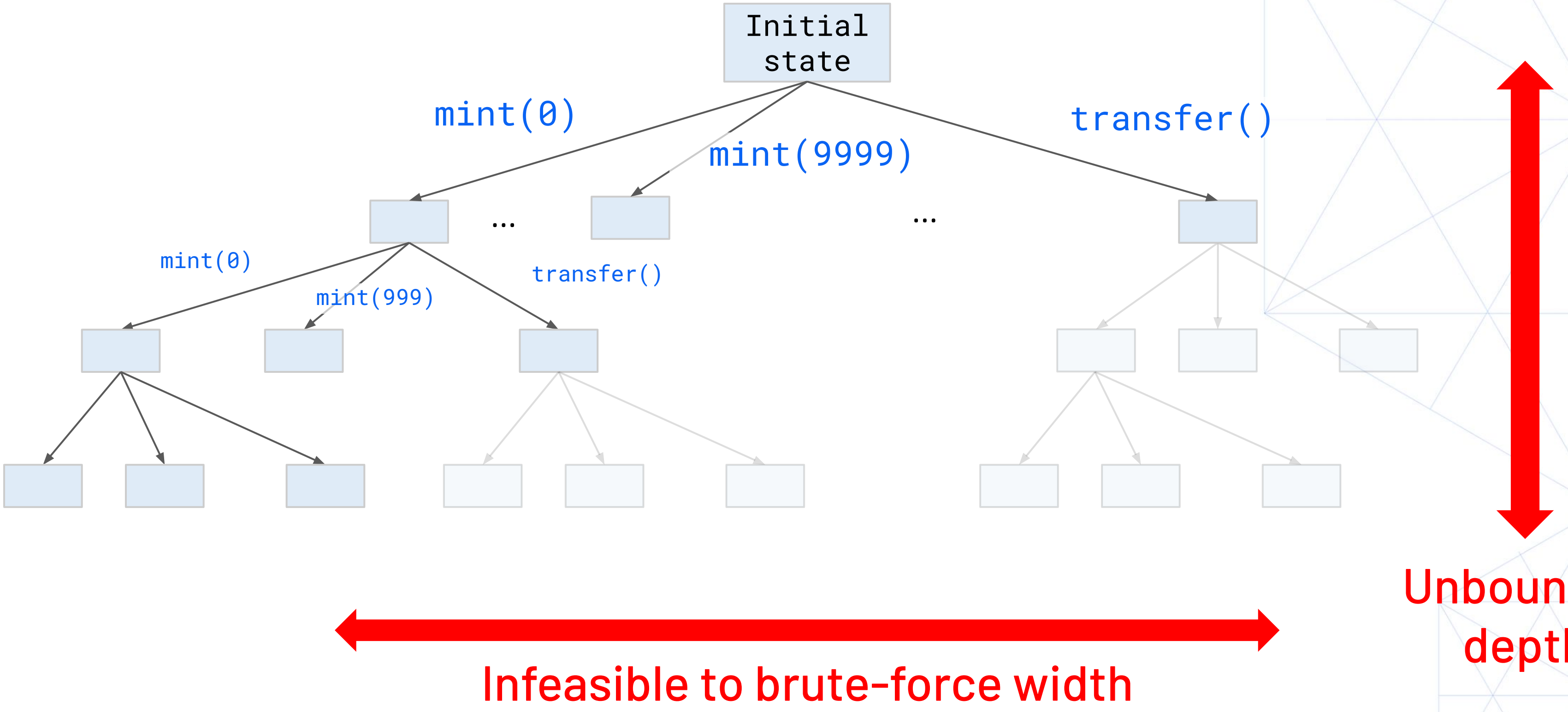
```
totalSupply = 100
owner = 0x10
balances[0x10] = 50
balances[0x20] = 50
```

mint(100)

```
totalSupply = 200
owner = 0x10
balances[0x10] = 150
balances[0x20] = 50
```

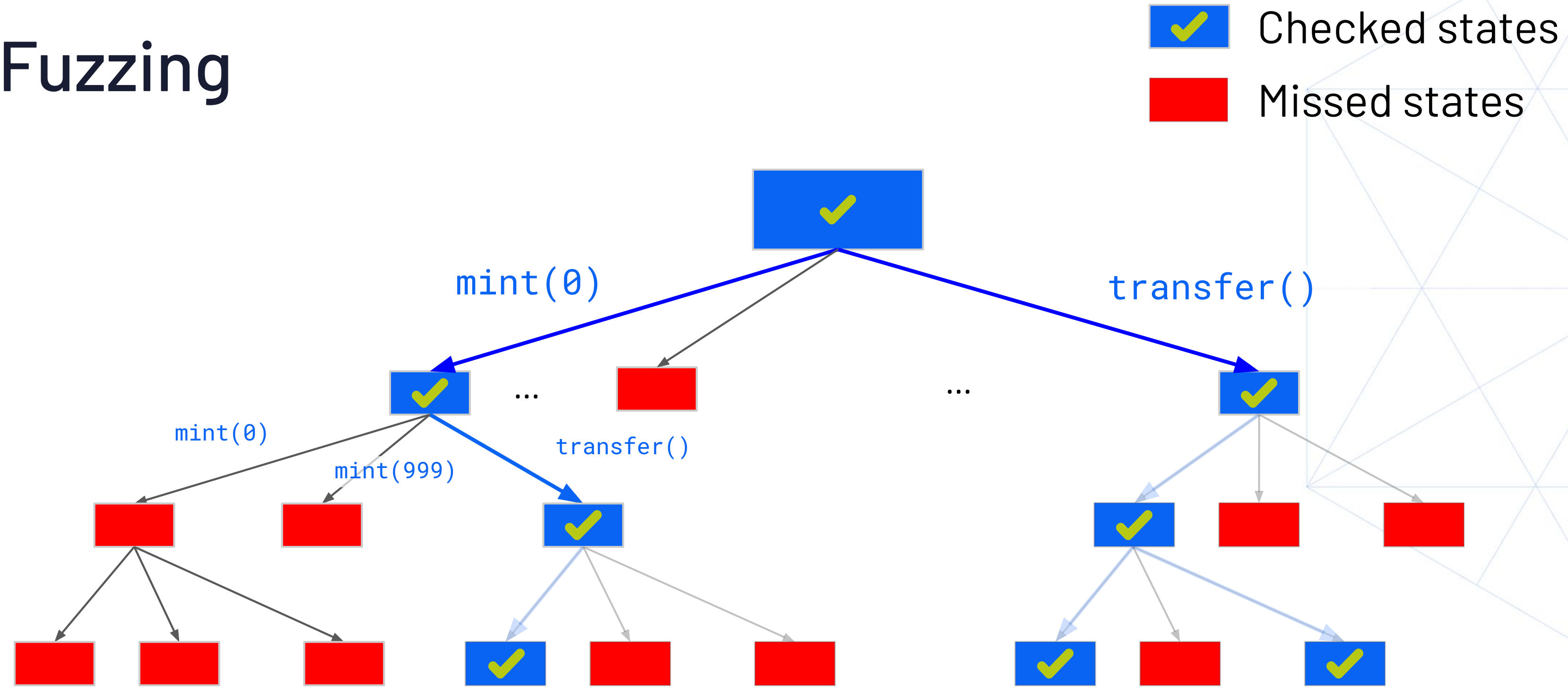
```
function mint(uint numTokens) {
    totalSupply += numTokens;
    balances[owner] += numTokens;
}
```

Step 2: ***Check*** formal property for all states



Fuzzing

Fuzzing



Wanted: Transaction sequences that thoroughly explore the state space

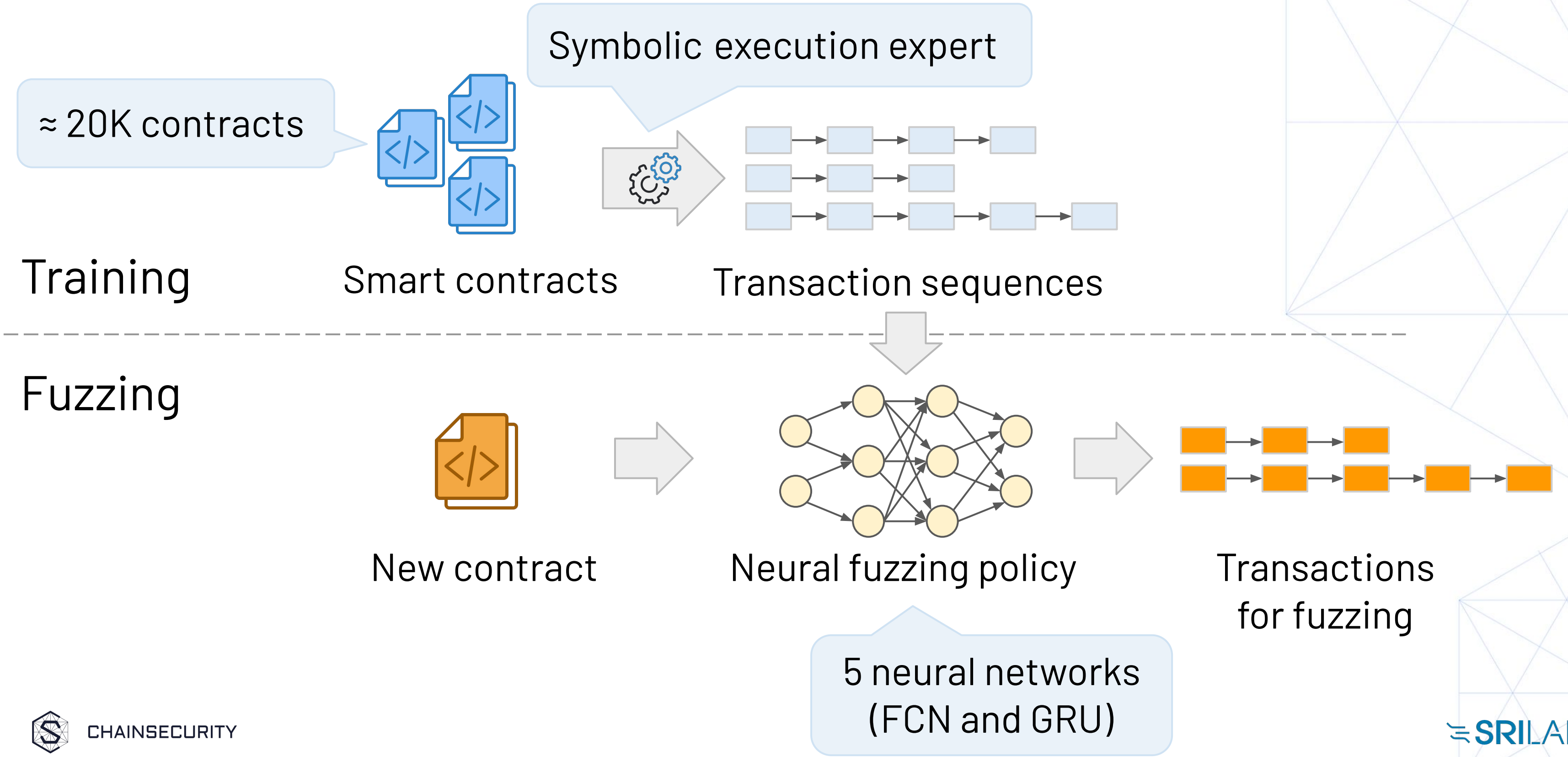
Generating good transaction sequences is hard

	Random fuzzing	Symbolic execution	ILF
Speed	Fast	Slow	Fast
Coverage	Low	High	High

ILF is described in:

Learning to Fuzz from Symbolic Execution with Application to Smart Contracts
J. He, M. Balunovic, N. Ambroladze, P. Tsankov, M. Vechev
ACM CCS 2019

ILF: Learning to fuzz from symbolic execution



Blockchain security @ SRI

Security goal

Avoid generic vulnerabilities

Enforce data privacy

Ensure functional correctness

Technique

Static analysis

Datalog

Type checking

Predicate abstraction

Temporal logic

Fuzzing

Symbolic execution

Zero-knowledge

Reinforcement learning

System

Securify	ILF	VerX	zkay
ACM CCS'18	ACM CCS'19	IEEE S&P'20	ACM CCS'19

Requirements for ensuring functional correctness

	Manual verification	Symbolic execution
Full guarantees	✓	?
User-friendly specifications	✗	?
Automated technique	✗	?
Scales to real-world contracts	✓	?

Smart contracts are usually **loop-free**. Can we leverage **symbolic execution** to automatically verify them?

Execution model of smart contracts

(Implicit) unbound loop

```
while True {
    (user, func, args) := // arbitrary
    run func(args) as user
}
```

(Typically) loop-free

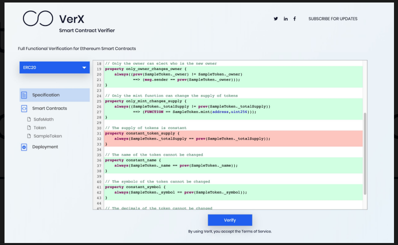
Symbolic execution **cannot** verify programs with unbounded loops

Requirements for ensuring functional correctness

	Manual verification	Sym. Exec. tools	VerX
Full guarantees	✓	✗	✓
User-friendly specifications	✗	✗	✓
Automated technique	✗	✓	✓
Scales to real-world contracts	✓	✓	✓

VerX

The first automated verifier for smart contracts



https://verx.ch

IEEE Symposium on Security & Privacy 2020

Requirements for ensuring functional correctness

	Manual verification	Sym. Exec. tools	VerX
Full guarantees	✓	✗	✓
User-friendly specifications	✗	✗	✓
Automat	✗	✓	✓
Scales to … cts	✓		✓

Automated formal verification with VerX

"The sum of balances always equals the total supply"

Smart contract

```
mapping(addr => uint) balances;
function transfer(address,uint);
```

Specification in the VerX language

```
always SUM(ERC.balances)
    == ERC.totalSupply
```

VerX

Verified

May not hold

CHAINSECURITY

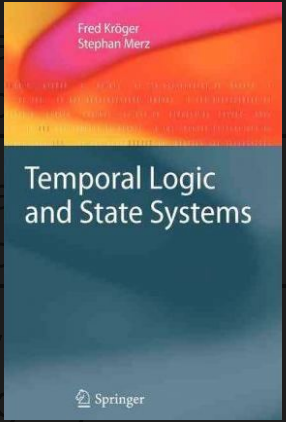
SRILAB

VerX specification language

Access control	`always Escrow.deposit(address) ==> (msg.sender == Escrow.owner)`
State-based properties	`always (now > Vault.refundTime + 1 week) ==> ! Vault.refund(uint256)`
State machine properties	`always !(once(state == REFUND) && once(state == FINALIZED))`
Invariants over aggregates	`always totalSupply == sum(balances)`
Multi-contract invariants	`always Token.totalSupply >= Sale.issuance`

VerX specification language

Access control	`always Escrow.deposit(address) ==> (msg.sender == Escrow.owner)`
Sta… pro…	`always (now > Vault.refundTime + 1 week) ==> ! Vault.refund(uint256)`
Sta… pro…	`always !(once(state == REFUND) && once(state == FINALIZED))`
Inv… agg…	`always totalSupply == sum(balances)`
Multi-contract invariants	`always Token.totalSupply >= Sale.issuance`



Past Linear Temporal Logic

Specification in the presence of callbacks

Contract function

```
function mint(uint n) {
   supply += n;
   msg.sender.call.value()();
}
```

Does this property hold?

```
always mint(uint n)
          ==> supply == prev(supply) + n;
```

1. Enforce that the contract is *effectively external callback free*
2. Interpret temporal properties over traces without callbacks

Automated formal verification with VerX

"The sum of balances always equals the total supply"

Smart contract

```
mapping(addr => uint) balances;
function transfer(address,uint);
```

Specification in the VerX language

```
always SUM(ERC.balances)
   == ERC.totalSupply
```

VerX

Verified

May not hold

Automated verification method

CHAINSECURITY

SRILAB

Where to apply abstraction?

```
func transfer(addr to, uint amt) {
  balances[msg.sender] -= amt;
  balances[to] += amt;
}
```

```
sum(ERC.balances) == ERC.totalSupply
```

```
while True {
    (user, func, args) := // arbitrary
    run func(args) as user
}
```

Abstraction within transactions is:

- **hard** (storage invariants often temporarily violated)
- **unnecessary** (no loops)

Where to apply abstraction?

Abstraction across transaction is:

- **easy** (storage invariants preserved)
- **necessary** (unbounded loop)

```
while True {
    (user, func, args) := // arbitrary
    run func(args) as user
}
```

Abstraction within transactions is:

- **hard** (storage invariants often temporarily violated)
- **unnecessary** (no loops)

Delayed predicate abstraction

2 Predicate abstraction

```
while True {
    (user, func, args) := // arbitrary
    run func(args) as user
}
```

1 Scalable and precise symbolic execution

Step 1: Symbolic execution

```
function mint(uint numTokens) {
    totalSupply += numTokens;
    balances[owner] += numTokens;
}
```

```
totalSupply = 100
owner = 0x10
balances[0x10] = 50
balances[0x20] = 50
```

mint(X)

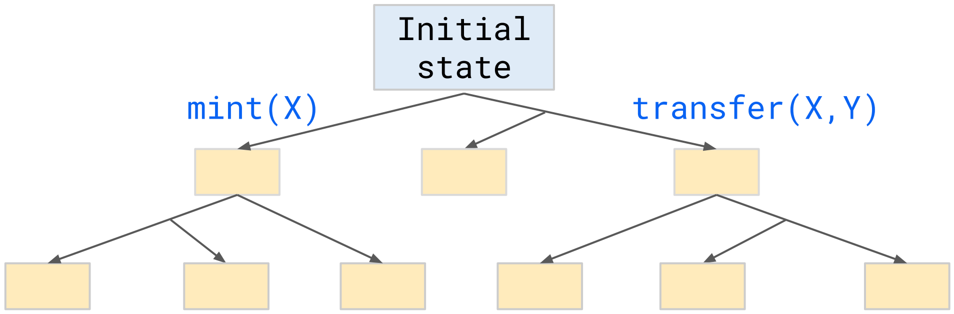
Symbolic arguments

```
totalSupply = 100 + X
owner = 0x10
balances[0x10] = 50 + X
balances[0x20] = 50
```

Symbolic state

Constraints capture a set of concrete states

Analysis with symbolic execution



Depth usually bounded to 2-3 transactions

Feasible width (# of paths)

To reach a fixed-point of all feasible states we need **abstraction**

Step 2: Predicate abstraction

Predicates

```
P:   sum(balances) == totalSupply
Q:   totalSupply < 1000
```

Abstract states

P ∧ Q	¬P ∧ Q	P ∧ ¬Q	¬P ∧ ¬Q

Captures all concrete states that satisfy both **P** and **Q**

Abstraction step

Predicates

```
P:  sum(balances) == totalSupply
Q:  totalSupply < 1000
```

Concrete or symbolic state

```
totalSupply = 100
balances[0x10] = 50
balances[0x20] = 50
```

```
totalSupply = 100 + X
balances[0x10] = 100
balances[0x20] = 50 + X
X <= 500
```

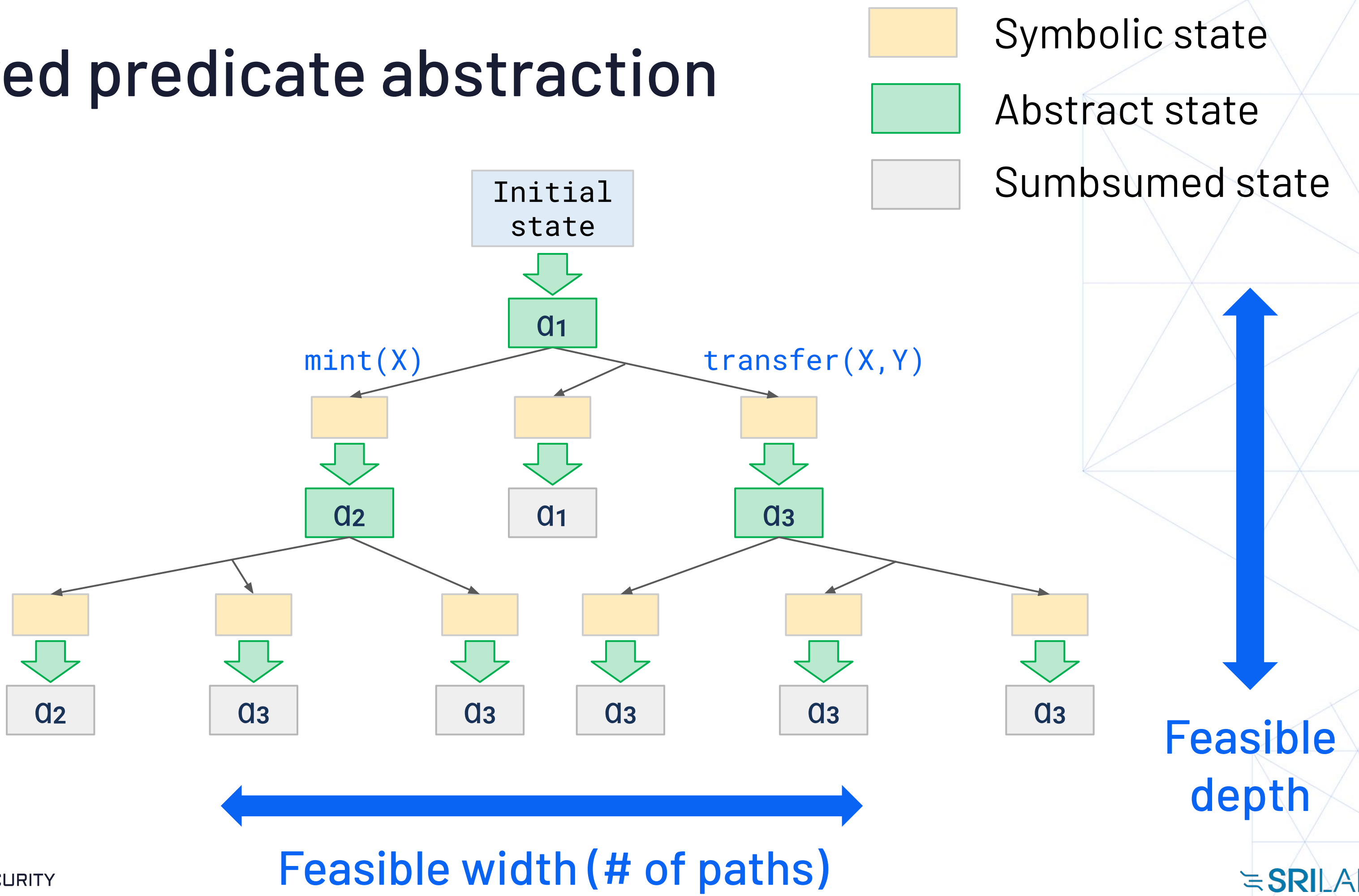
abstraction step

Abstract state

$P \land Q$

$\neg P \land Q$

Delayed predicate abstraction



Impact

Verification time down to **hours** (for contracts with standard specs)

100+ smart contracts verified

Polkadot. MADANA monart

Jul 26, 2019 Sep 23, 2019 Aug 28, 2019

https://chainsecurity.com/audits

Blockchain security @ SRI

Security goal	Avoid generic vulnerabilities	Ensure functional correctness	Enforce data privacy	
Technique	Static analysis, Symbolic execution	Datalog, Temporal logic	Type checking, Zero-knowledge	Predicate abstraction, Fuzzing, Reinforcement learning
System	SECURIFY ACM CCS'18	ILF ACM CCS'19	VerX IEEE S&P'20	zkay ACM CCS'19

Are smart contracts and

data privacy

compatible?

Public blockchains

Smart contract

```
mapping(addr => uint) balances;
function mint(uint);
function transfer(address,uint);
```

Public data storage

Address	Balance
0x10	100
0x20	50

Miners need **code** and **data** to process transactions

Sensitive data, want to keep private

Can we easily achieve data privacy using zero-knowledge proofs?

The promise of zero-knowledge proofs (zkSNARKs)

Smart contract

```
mapping(addr => uint) balances;
function mint(uint);
function transfer(address,uint);
```

Public data storage

0xFA034FFAD245

1. Only keep the hash of the state
2. Update hash and provide a zero-knowledge proof of correctness

Challenges

zkSNARKs are incomplete	Cannot move all computation off-chain
Knowledge restrictions	Users have different secrets
Obfuscated logic	Mix of smart contract and proof circuits
Information leaks	No clear privacy guarantee

Specification and enforcement of data privacy

zkay contract **with privacy annotations**

```
mapping(addr!x => uint@x) balances;
function add(uint@msg.sender val){
   balances[msg.sender] += val; }
```

Private data storage

0x10	100
0x20	50

zkay's type checker ensures that:

- privacy is realizable with zkSNARKs
- no implicit information leaks

Paper:

zkay: Specifying and enforcing data privacy in smart contracts

S. Steffen, B. Bichsel, M. Gersbach, N. Melchoir, P. Tsankov, M. Vechev

ACM CCS 2019

Specification and enforcement of data privacy

zkay contract **with privacy annotations**

```
mapping(addr!x => uint@x) balances;
function add(uint@msg.sender val){
   balances[msg.sender] += val; }
```

Private data storage

0x10	100
0x20	50

Sensitive data is encrypted

Confidentiality protected using encryption

Correctness ensured with zkSNARKs

Ethereum smart contract

Prove that "values" are correct

```
mapping(addr => bin) incomes;
function add(bin x, v, proof) {
   verify(x,v,proof,balances[msg.sender]);
   incomes[msg.sender] = x; }
```

Public data storage

0x10	A71B03CD3..
0x20	84F1A0D32..

Specificat

zkay contract w

```
mapping(addr!x =
function add(uir
    balances[msg.
```

Ethereum smar

```
mapping(addr =>
function add(bi
    verify(x,v,p
    incomes[msg.
```

```
…
vk.gammaBeta2 = Pairing.G2Point([0x13442e2b8cabe4d083a82fe866f57ac6226121f53a3352760bd80e2e52f64693,
0x7409405004b9c09d0472d6850ae871463b696b1789a68130af412ef20269c8b],
[0x16e8f82231808bb14b02be5771a2ebd9b765f53f07a5c2e4b80c0e15c6c44047,
0x2b29fd644252cfa867688eb7bf755627b8984e6312ea36a7ce82c9d3c9d0e665]);
vk.Z = Pairing.G2Point([0x5490f1553d46b72d5065d6f5871cba69cfd21a304ac6d51de223259f9790c50,
0x2c67c3dd4c22cc93e4fee9872297e280801b32f0c8435c1afaace0d6b8b90965],
[0x2c242fef130aae1603754fe03ea501d79385afbfb1f70bdcf0eff7683859a6a,
0x2400f6464e0a2c32be4f0e0da5a701b7d13f78bb7b30e361f90140f664a99fb7]);
vk.IC = new Pairing.G1Point[](5);
vk.IC[0] = Pairing.G1Point(0x2ee4f10762b35f2a67ae89efebac135e4203cdc7b162f42d0a944ab8ba44a086,
0x2a7d46ff63646ba212c15001c58f4f9c4ecd433557c274fd7320eb824953b914);
vk.IC[1] = Pairing.G1Point(0x2cfa9355b8bb1cdec908183814869029724b02e02d337e03916d9cb8a989d9f1,
0x164b565e3808c3d9868f1072ebe6b42c84ef340976fe1dbc6faa39ff3adfd3fc);
vk.IC[2] = Pairing.G1Point(0x10b5e80161c4d707bc4e14f6c30b9c4420ec3cd52483c3226236f3ae6b055128,
0x28c978aeb361c6995ab5cd7d16db711fd06b89220cfdcf9c79a9254962ca1714);
vk.IC[3] = Pairing.G1Point(0x294e0fe8a51402e49f309de676f367d8348dd5f7a6f78d10175ec0fa5f3b4a46,
0x2c24e6fda46bbe79237df7b02516fcfbc109fd0b9e1cc6b6fccd8c1bade31e90);
vk.IC[4] = Pairing.G1Point(0x1bc44dac5f2d4a04b7e8221bb8ed82c371a7f850dc5c7896482eb1bdc1083996,
0x19bc80a310c0ca4fec41dda4be8e7d50bc51b773412f894e4f925b73b9534568);
}
function verify(uint[] input, Proof proof) internal returns (uint) {
VerifyingKey memory vk = verifyingKey();
require(input.length + 1 == vk.IC.length);
Pairing.G1Point memory vk_x = Pairing.G1Point(0, 0);
for (uint i = 0; i < input.length; i++)
        vk_x = Pairing.addition(vk_x, Pairing.scalar_mul(vk.IC[i + 1], input[i]));
vk_x = Pairing.addition(vk_x, vk.IC[0]);
if (!Pairing.pairingProd2(proof.A, vk.A, Pairing.negate(proof.A_p), Pairing.P2())) return 1;
…
```

data is

Blockchain security @ SRI

Security goal

Avoid generic vulnerabilities

Enforce data privacy

Ensure functional correctness

Technique

Static analysis

Datalog

Type checking

Predicate abstraction

Temporal logic

Fuzzing

Symbolic execution

Zero-knowledge

Reinforcement learning

System

SECURIFY	ILF	VerX	zkay
ACM CCS'18	ACM CCS'19	IEEE S&P'20	ACM CCS'19