Smart Contract Security: Testing, Verification, and Data Privacy

Dr. Petar Tsankov
@ptsankov

Co-founder and Chief scientist, ChainSecurity
Senior researcher, ETH Zurich
Security @ SRI Lab:

- Blockchain security
- Safety and security of AI
- Security and privacy

Next-generation blockchain security with automated reasoning

https://chainsecurity.com
@chainsecurity
## Smart contract security @ SRI

<table>
<thead>
<tr>
<th>Security goal</th>
<th>Technique</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avoid generic vulnerabilities</td>
<td>Enforce data privacy</td>
<td><strong>SECU</strong> – ACM CCS’18</td>
</tr>
<tr>
<td>Ensure functional correctness</td>
<td></td>
<td><strong>ILF</strong> – ACM CCS’19</td>
</tr>
<tr>
<td></td>
<td>Fuzzing</td>
<td><strong>VerX</strong> – IEEE S&amp;P’20</td>
</tr>
<tr>
<td></td>
<td>Predicate abstraction</td>
<td><strong>zkay</strong> – ACM CCS’19</td>
</tr>
<tr>
<td></td>
<td>Static analysis</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Symbolic execution</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Datalog</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Type checking</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Temporal logic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Zero-knowledge</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reinforcement learning</td>
<td></td>
</tr>
</tbody>
</table>
# Smart contract security @ SRI

<table>
<thead>
<tr>
<th>Security goal</th>
<th>Technique</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avoid generic vulnerabilities</td>
<td>Static analysis</td>
<td>SECURIFY</td>
</tr>
<tr>
<td>Ensure functional correctness</td>
<td>Datalog</td>
<td>ACM CCS'18</td>
</tr>
<tr>
<td>Enforce data privacy</td>
<td>Type checking</td>
<td>ILF</td>
</tr>
<tr>
<td></td>
<td>Temporal logic</td>
<td>ACM CCS'19</td>
</tr>
<tr>
<td></td>
<td>Zero-knowledge</td>
<td>VerX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IEEE S&amp;P'20</td>
</tr>
<tr>
<td></td>
<td>Fuzzing</td>
<td>zkay</td>
</tr>
<tr>
<td></td>
<td>Reinforcement learning</td>
<td>ACM CCS'19</td>
</tr>
</tbody>
</table>

ACM CCS'18

IEEE S&P'20
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

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

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

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!
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
- Ensure functional correctness
- Enforce data privacy

Technique:
- Static analysis
- Symbolic execution
- Datalog
- Type checking
- Temporal logic
- Zero-knowledge
- Fuzzing
- Reinforcement learning

System:
- SECURIFY: ACM CCS’18
- ILF: ACM CCS’19
- VerX: IEEE S&P’20
- zkay: ACM CCS’19
Functional correctness

Smart contract ⊨ Functional specification
Correctness of ERC20

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

\[
\text{mapping}(\text{addr} \Rightarrow \text{uint}) \text{ balances;}
\text{uint totalSupply;}
\text{addr owner;}
\]

\text{function mint();}
\text{function changeOwner(addr o);}

<table>
<thead>
<tr>
<th>Smart contract</th>
<th>Functional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. The sum of all balances equals the total supply</td>
</tr>
<tr>
<td></td>
<td>2. Only the owner can increase the total supply of tokens</td>
</tr>
<tr>
<td></td>
<td>3. ...</td>
</tr>
</tbody>
</table>

CHAINSECURITY
Step 1: **Formalize** requirements

"The sum of balances equals the total supply"

Formal property

\[ \text{SUM}(\text{ERC20.balances}) \equiv \text{ERC20.totalSupply} \]

<table>
<thead>
<tr>
<th>Case</th>
<th>Balances</th>
<th>Total Supply</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>balances[0x10] = 50, balances[0x20] = 50</td>
<td>totalSupply = 100</td>
</tr>
<tr>
<td></td>
<td>balances[0x10] = 100, balances[0x20] = 50</td>
<td>totalSupply = 100</td>
</tr>
</tbody>
</table>

- ✔️ Property holds
- ❌ Property does not hold
Step 2: **Check** formal property for all states

- Initial state
  - mint(0)
  - mint(9999)
  - transfer()
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

Infeasible to brute-force width
Fuzzing
Fuzzing

Wanted: Transaction sequences that thoroughly explore the state space
Generating good transaction sequences is hard

<table>
<thead>
<tr>
<th></th>
<th>Random fuzzing</th>
<th>Symbolic execution</th>
<th>ILF</th>
</tr>
</thead>
</table>
| Speed              | ✔️ Fast        | ✗ Slow             | ✔️  
| 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

Training
- Smart contracts
- Transaction sequences
  - Symbolic execution expert
  - ≈ 20K contracts

Fuzzing
- New contract
- Neural fuzzing policy
  - 5 neural networks (FCN and GRU)
- Transactions for fuzzing
Blockchain security @ SRI

Security goal
- Avoid generic vulnerabilities
- Ensure functional correctness
- Enforce data privacy

Technique
- Static analysis
- Symbolic execution
- Datalog
- Type checking
- Temporal logic
- Zero-knowledge
- Fuzzing
- Reinforcement learning
- Predicate abstraction

System
- SECURIFY: ACM CCS’18
- ILF: ACM CCS’19
- VerX: IEEE S&P’20
- zkay: ACM CCS’19
## Requirements for ensuring functional correctness

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Manual verification</th>
<th>Symbolic execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full guarantees</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>User-friendly specifications</td>
<td>❌</td>
<td>?</td>
</tr>
<tr>
<td>Automated technique</td>
<td>❌</td>
<td></td>
</tr>
<tr>
<td>Scales to real-world contracts</td>
<td>✔️</td>
<td></td>
</tr>
</tbody>
</table>

Smart contracts are usually **loop-free**. Can we leverage **symbolic execution** to automatically verify them?
while True {
    (user, func, args) := // arbitrary
    run func(args) as user
}
## Requirements for ensuring functional correctness

<table>
<thead>
<tr>
<th>Feature</th>
<th>Manual verification</th>
<th>Sym. Exec. tools</th>
<th>VerX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full guarantees</td>
<td>✔️</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td>User-friendly specifications</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td>Automated technique</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Scales to real-world contracts</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
Requirements for ensuring functional correctness

VerX

The first automated verifier for smart contracts

Full guarantees
User-friendly specifications
Automated verification
Scales to real-world contracts

Manual verification Sym. Exec. tools

https://verx.ch

IEEE Symposium on Security & Privacy 2020
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
<table>
<thead>
<tr>
<th>Access control</th>
<th><code>always Escrow.deposit(address) ==&gt; (msg.sender == Escrow.owner)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>State-based properties</td>
<td><code>always (now &gt; Vault.refundTime + 1 week) ==&gt; ! Vault.refund(uint256)</code></td>
</tr>
<tr>
<td>State machine properties</td>
<td><code>always !(once(state == REFUND) &amp;&amp; once(state == FINALIZED))</code></td>
</tr>
<tr>
<td>Invariants over aggregates</td>
<td><code>always totalSupply == sum(balances)</code></td>
</tr>
<tr>
<td>Multi-contract invariants</td>
<td><code>always Token.totalSupply &gt;= Sale.issuance</code></td>
</tr>
</tbody>
</table>
## VerX specification language

### Access control

<table>
<thead>
<tr>
<th>Property</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>always</td>
<td><code>Escrow.deposit(address) ==&gt; (msg.sender == Escrow.owner)</code></td>
</tr>
</tbody>
</table>

### State-based properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>always</td>
<td><code>(now &gt; Vault.refundTime + 1 week) ==&gt; ! Vault.refund(uint256)</code></td>
</tr>
</tbody>
</table>

### State machine properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>always</td>
<td><code>!(once(state == REFUND) &amp;&amp; once(state == FINALIZED))</code></td>
</tr>
</tbody>
</table>

### Invariants over aggregates

<table>
<thead>
<tr>
<th>Property</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>always</td>
<td><code>totalSupply == sum(balances)</code></td>
</tr>
</tbody>
</table>

### Multi-contract invariants

<table>
<thead>
<tr>
<th>Property</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>always</td>
<td><code>Token.totalSupply &gt;= Sale.issuance</code></td>
</tr>
</tbody>
</table>
Specification in the presence of callbacks

Contract function

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

Does this property hold?

```solidity
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

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

Specification in the VerX language

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

Automated verification method

VerX

Verified

May not hold
Where to apply abstraction?

```solidity
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)

```python
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

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

1. Scalable and precise symbolic execution
Step 1: Symbolic execution

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

Constraints capture a set of concrete states:
- `totalSupply` = 100
- `owner` = 0x10
- `balances[0x10]` = 50
- `balances[0x20]` = 50

Symbolic arguments:
- `mint(X)`

Symbolic state:
- `totalSupply` = \(100 + X\)
- `owner` = 0x10
- `balances[0x10]` = 50 + X
- `balances[0x20]` = 50
Analysis with symbolic execution

To reach a fixed-point of all feasible states we need **abstraction**
Step 2: Predicate abstraction

Predicates

P: \( \text{sum}(\text{balances}) == \text{totalSupply} \)
Q: \( \text{totalSupply} < 1000 \)

Abstract states

\( P \land Q \quad \neg P \land Q \quad P \land \neg Q \quad \neg P \land \neg Q \)

Captures all concrete states that satisfy both \( P \) and \( Q \)
Abstraction step

Predicates

\[ P: \text{sum(balances)} == \text{totalSupply} \]
\[ Q: \text{totalSupply} < 1000 \]

Concrete or symbolic state

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

Abstract state

- \( P \land Q \)
- \( \neg P \land Q \)
Delayed predicate abstraction

Initial state

mint(X)
transfer(X,Y)

Symbolic state
Abstract state
Sumbsumed state

Feasible width (# of paths)

Feasible depth
Impact

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

100+ smart contracts verified

https://chainsecurity.com/audits
## Blockchain security @ SRI

<table>
<thead>
<tr>
<th>Security goal</th>
<th>Technique</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avoid generic vulnerabilities</td>
<td>Ensure functional correctness</td>
<td>Enforce data privacy</td>
</tr>
<tr>
<td>Ensure functional correctness</td>
<td>Static analysis</td>
<td>Datalog</td>
</tr>
<tr>
<td></td>
<td>Symbolic execution</td>
<td>Type checking</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Temporal logic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Zero-knowledge</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Predicate abstraction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fuzzing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reinforcement learning</td>
</tr>
</tbody>
</table>

**System**:
- **SECURIFY**: ACM CCS’18
- **ILF**: ACM CCS’19
- **VerX**: IEEE S&P’20
- **zkay**: ACM CCS’19

**Technique**:
- Static analysis
- Symbolic execution
- Datalog
- Type checking
- Temporal logic
- Zero-knowledge
- Predicate abstraction
- Fuzzing
- Reinforcement learning

**Security goal**:
- Avoid generic vulnerabilities
- Ensure functional correctness
- Enforce data privacy
Are smart contracts and data privacy compatible?
Public blockchains

Smart contract

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

Public data storage

<table>
<thead>
<tr>
<th>Address</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td>100</td>
</tr>
<tr>
<td>0x20</td>
<td>50</td>
</tr>
</tbody>
</table>

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

```solidity
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

Knowledge restrictions

Obfuscated logic

Information leaks

Cannot move all computation off-chain

Users have different secrets

Mix of smart contract and proof circuits

No clear privacy guarantee
Specification and enforcement of data privacy

zkay contract with privacy annotations

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

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**

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

**Private data storage**

- 0x10: 100
- 0x20: 50

Confidentiality protected using encryption
Correctness ensured with zkSNARKs

**Public data storage**

- 0x10: A71B03CD3...
- 0x20: 84F1A0D32...

Prove that “values” are correct
zkay contract with privacy annotations

Specification and enforcement of data privacy

mapping(addr|x) => uint balances;

function add(uint @msg.sender val) {
balances[msg.sender] += val;
}

Private data storage

0x10 100
0x20 50

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 0x20
A71B03CD3..84F1A0D32..Ethereum smart contract
Confidentiality protected using encryption
Correctness ensured with zkSNARKs

Sensitive data is encrypted
Prove that "values" are correct

\[
\begin{align*}
\text{vk.gammaBeta2} &= \text{Pairing.G2Point}([0x13442e2b8cabe4d083a82fe866f57ac6226212f53a3352760bd8e0e252f64693, \\
0x740945004b9c09d472d6850a8871463b6961789a6813a4df412e5f0269c8b], \\
[0x1e6e8f822318008b14b92be5771a2ebd9b765f53f087a5c2e4b80c8e156c44047, \\
0x2b29f644252cfa867688eb7f575627b989846312e6a367ce82c9d3cd90e665]); \\
\text{vk.Z} &= \text{Pairing.G2Point}([0x5490f1553d46b72d5865d6f5871c9a69cd21a304accd615de223259f9790c50, \\
0x2c67c3dd4c22cc93e4f9872297e220081b326c8435c1afaace0d6b8b90965], \\
[0x2c242f8f3a10e693754e003a1e59d79385a4fbbf1f70bdcf0eff768359a6a, \\
0x2400fd6464ee0a2c323e4bfe400da5a57017d13f7bb7b39e3b1f90140f64a999f7]); \\
\text{vk.IC} &= \text{new Pairing.G1Point}([5]); \\
\text{vk.IC}[0] &= \text{Pairing.G1Point}([0x2eefe4f107623b35f26a7e989e4f89e1354e23d8b7162f42d0a944ab8ba44a086, \\
0x2a7d466ff6346e6b212c15001c584f9c4ecd433557c274fd7320eb824953b914]); \\
\text{vk.IC}[1] &= \text{Pairing.G1Point}([0x2ca9355b8b1c0dec9081381486929724b02e02d337e0916d9cb8a99899f1, \\
0x16b565e3888c3d9868f6107e26b42c84ef340976fe1dcb6faa9ff3affd3efc]); \\
\text{vk.IC}[2] &= \text{Pairing.G1Point}([0x10b5e80161c4d7b7c0e4f6c30b9c4420ec3c5d2483c3226236f3ae6b055120, \\
0x28c978aeb361c6959ab5c5d7c1d6b86b92280c7929a254962ca1714]); \\
\text{vk.IC}[3] &= \text{Pairing.G1Point}([0x2894e8f8e5a3f429f390d6763d8348dd5f67a67e78d180175ec0fa5f3b4a46, \\
0x2c24e6fda46bbe79237df7b02516fcfcbe109fd0b9e1cc6b6ffcdcc81bade31e90]); \\
\text{vk.IC}[4] &= \text{Pairing.G1Point}([0x1bc44dacf524da40b7e8221bb8ed82c371af7f850dc5c7896482eb1bdc1083996, \\
0x197c80a310c8ca44fe41d4aa4eb8e7d50bc51b773412f894e4f925b73b9535468]); 
\end{align*}
\]

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++) 
\[
\begin{align*}
\text{vk.x} &= \text{Pairing.addition}(\text{vk.x}, \text{Pairing.scalar_mul(vk.IC[i + 1], input[i]))}; \\
\text{vk.x} &= \text{Pairing.addition}(\text{vk.x}, \text{vk.IC}[0]);
\end{align*}
\]
}
### Blockchain security @ SRI

#### Security goal
- Avoid generic vulnerabilities
- Ensure functional correctness
- Enforce data privacy

#### Technique
- Static analysis
- Symbolic execution
- Datalog
- Type checking
- Temporal logic
- Zero-knowledge
- Fuzzing
- Reinforcement learning

#### System
- **SECURIFY**
  - ACM CCS’18
- **ILF**
  - ACM CCS’19
- **VerX**
  - IEEE S&P’20
- **zkay**
  - ACM CCS’19