Program Analysis for System Security and Reliability

Petar Tsankov
Spring 2018

http://www.srl.inf.ethz.ch
http://ice.ethz.ch
Last time

Introduction to decentralized ledgers

- Merkle trees, digital signatures
- Bitcoin, transactions, proof-of-work
- Ethereum, smart-contracts
Plan for Today

I. Security issues in smart contracts
   - The DAO hack
   - Parity bug #1
   - Parity bug #2

II. Security properties
   - Informal semantics of smart contracts
   - Writes after calls, unrestricted writes, locked ether

III. Automated security analysis
   - Securify [https://securify.ch](https://securify.ch)
Smart contract bugs in the news

The DAO Attacked: Code Issue Leads to $60 Million Ether Theft
Jun 17, 2016 at 14:00 UTC by Michael del Castillo

The DAO, the distributed autonomous organization on Ethereum, has reportedly been hacked, sparking a big crypto currency crisis. It has lost 3.6m ether, which is currently sitting at $60 million.

The DAO Falls Victim to Attack Leading Ether Crash Over 20%
The event is still ongoing as hackers are still exploiting the bug that led to the attack.

Wallet bug freezes more than $150 million worth of Ethereum

CNBC

$32 million worth of digital currency ether stolen by malware

$32.6 million were taken by hackers on Monday where 7 million worth of ether

$30 million worth of ethereum stolen in another hacker attack

A bug in Parity, a popular wallet for the cryptocurrency and decentralized application platform Ethereum, may have resulted in more than $150 million worth of ether being permanently frozen.

A bug in Parity multi-sig (multi-signature) wallets, which require more than one owner to "sign" a transaction before it can go through, has exploited it to effectively destroy a piece of Parity's code, effectively rendering all multi-sig wallets that were created after July 20 completely unusable.
we will look at the three most critical bugs in 2016-2017...
June 2016: The DAO bug

The DAO, the distributed autonomous organization that had collected over $150m worth of the cryptocurrency ether, has reportedly been hacked, sparking a broad market sell-off.

A leaderless organization comprised of a series of smart contracts written on the ethereum codebase, The DAO has lost 3.6m ether, which is currently sitting in a separate wallet after being split off into a separate protocol.

The DAO falls victim to cyber attack leading Ethereum to crash over 20%

The event is still ongoing as hackers have already stolen over 3.5 million ETH from the DAO’s coffers.

CNBC reports $30 million worth of ether was stolen in another hacker attack

Ethereum has been rocked by another hacking attack targeting a popular wallet software.

A bug in Parity, a popular wallet for the cryptocurrency and decentralized application platform Ethereum, may have resulted in more than $150 million worth of ether being permanently frozen.

The bug affects Parity multi-sig (multi signature) wallets, which require more than one owner to “sign” a transaction before it can go through. An unknown attacker (or a careless developer) has exploited it to effectively destroy a piece of Parity’s code, effectively rendering all multi-sig wallets that were created after July 20 completely unusable.

CNBC also reports $32.6 million were taken by hackers on Monday where $7 million worth of ether

CNBC

10:53 AM ET Thu, 20 July 2017

Security

$30 million worth of Ethereum stolen in another hacker attack

Ethereum has been rocked by another hacking attack targeting a popular wallet software.

A bug in Parity, a popular wallet for the cryptocurrency and decentralized application platform, may have resulted in more than $150 million worth of ether being permanently frozen.

The bug affects Parity multi-sig (multi signature) wallets, which require more than one owner to “sign” a transaction before it can go through. An unknown attacker (or a careless developer) has exploited it to effectively destroy a piece of Parity’s code, effectively rendering all multi-sig wallets that were created after July 20 completely unusable.
The DAO bug

Wallet contract

```solidity
uint balance = 10;

function withdraw()
{
    if (balance > 0)
    
        msg.sender.call.value(balance)();

    balance = 0;
}
```

calls the default fallback function of msg.sender

balance is set to zero

Can a user contract withdraw more than 10 ether?
The DAO bug

User contract

```solidity
function foo() {
    wallet.withdraw();
}

function () payable {
    // log payment
}
```

Wallet contract

```solidity
uint balance = 10;

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

Transaction 1: foo()
- User makes a withdrawal of 10 ether.

Transaction 2: foo()
- Later... User makes another withdrawal.
- Wallet contract has no ether left.
Ether transfers in Ethereum

**<receiver>.call.value(amountEther)(amountGas)**
- Specify the amount of ether / gas send to the receiver
- By default, i.e. using `call.value(amountEther)()`, sends all remaining gas to the receiver

**<receiver>.transfer(amountEther)**
- Sends 2300 gas units to the receiver, which allows the receiver to only log events

**<receiver>.send(amountEther)**
- Sends 2300 gas units to the receiver, which allows the receiver to only log events
- Error codes must be explicitly checked, e.g. using:
  
  ```
  require(<receiver>.send(amountEther))
  ```
The DAO bug

User contract

```solidity
function foo() {
    wallet.withdraw();
}

function () payable {
    wallet.withdraw();
}
```

calls withdraw() before balance is set to zero

Wallet contract

```solidity
uint balance = 10;

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

Transfer all remaining gas
The DAO bug

User contract

```solidity
function foo() {
    wallet.withdraw();
}

function () payable {
    wallet.withdraw();
}
```

Transaction 1: foo()

Wallet contract

```solidity
uint balance = 10;

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

An attacker exploited this bug to steal $150M worth of ether.
July 2017: Parity Wallet Bug #1

The DAO Attacked: Code Issue Leads to $60 Million Ether Theft

Ethereum • News • Ethereum

The DAO, the distributed autonomous organization that had collected over $150m worth of the cryptocurrency ether, has reportedly been hacked, sparking a broad market sell-off.

A leaderless organization comprised of a series of smart contracts written on the ethereum codebase, The DAO has lost 3.6m ether, which is currently sitting in a separate wallet after being split off into a separate project.

The DAO Falls Victim to Cyber Attack Leading Ethereum to Crash Over 20%

The event is still ongoing as hackers have already stolen over 3.5 million ETH from the DAO’s coffers.

Over $30 million worth of ethereum stolen in another hacker attack

Theft due to security issue with Parity’s wallet software

CNBC

$32 million worth of digital currency ether stolen by hackers

- Around 153,000 ether tokens worth $32.6 million were taken by hackers on Wednesday.
- A vulnerability in Parity’s multisignature wallet was exploited by hackers.
- This latest theft follows an incident on Monday where $7 million worth of ether tokens were stolen.

Luke Graham | @LukeWGraham
Published 7:41 AM ET Thu, 20 July 2017 | Updated 10:51 AM ET Thu, 20 July 2017

Etherscan.io
Unprivileged write to storage

Wallet

```solidity
address walletLibrary;
address owner;

function Wallet(address _owner) {
    walletLibrary = // address of wallet library;
    walletLibrary.delegatecall(
        bytes4(sha3("initWallet(address)")), _owner);
}

function withdraw(uint amount) returns {
    walletLibrary.delegatecall(
        bytes4(sha3("withdraw(uint)")), amount);
}

function () payable {
    walletLibrary.delegatecall(msg.data);
}
```

Wallet library

```solidity
address owner;

function initWallet(address _owner) {
    owner = _owner;
}

function withdraw(uint amount) {
    if (msg.sender == owner) {
        owner.send(amount);
    }
}

function () payable {
    // receive money
}
```

How does delegatecall work?
Delegatecall takes the code in the library and executes it in the context of the wallet.

The message sent to the wallet library is:

```solidity
bytes4(sha3("initWallet(address)")), _owner
```

where:

- The first four bytes identify the method to be executed at the library. E.g., in this case,

  ```solidity
  bytes4(sha3("initWallet(address)")) = 0xe3710ed0
  ```

- `_owner` is the address passed as argument to the Wallet constructor.
Function dispatch in smart contracts

The contract takes the first four bytes of the message data (msg.data) and matches them to the first four bytes of the function’s hash.

If no function is matched, it executes the fallback function.
back to the code of the wallet and the wallet library...
Unprivileged write to storage

**Wallet**

```solidity
address walletLibrary;
address owner;

function Wallet(address _owner) {
    walletLibrary = // address of wallet library;
    walletLibrary.delegatecall(
        bytes4(sha3("initWallet(address)")), _owner);
}

function withdraw(uint amount) returns {
    walletLibrary.delegatecall(
        bytes4(sha3("withdraw(uint)")), amount);
}

function () payable {
    walletLibrary.delegatecall(msg.data);
}
```

**Wallet library**

```solidity
address owner;

function initWallet(address _owner) {
    owner = _owner;
}

function withdraw(uint amount) {
    if (msg.sender == owner) {
        owner.send(amount);
    }
}

function () payable {
    // receive money
}
```

- The wallet forwards msg.data to the library
- Attacker calls the wallet contract with

```solidity
msg.data = (bytes4(sha3("initWallet(address)")), _owner)
```
Unprivileged write to storage

An attacker exploited this bug to steal $30M worth of ether

- The wallet forwards `msg.data` to the library
- Attacker calls the wallet contract with

```solidity
msg.data = (bytes4(sha3("initWallet(address)")), _owner)
```
November 2017: Parity Wallet Bug #2

The DAO Attacked: Code Issue Leads to $60 Million Ether Theft

The DAO, the distributed autonomous organization that uses ether, has reportedly been hacked, sparking a billion-dollar hack. A leaderless organization comprised of a series of smart contracts, the DAO has lost 3.6m ether, which is currently sitting in a black hole.

The DAO Falls Victim to Attack Leading Ethereum Crash Over 20%

The event is still ongoing as hackers continue to drain ether. According to reports, more than 3.5 million ETH from the smart contracts were stolen.

Wallet bug freezes more than $150 million worth of Ethereum

A bug in Parity, a popular wallet for the cryptocurrency and decentralized application platform Ethereum, may have resulted in more than $150 million worth of ether being permanently frozen.

Security alert, warning of a vulnerability in Parity’s wallet software.

Million worth of ether have been stolen worth nearly $35 million at current price levels. The amount of the stolen ether has been confirmed by Etherscan.io.
Locked ether

Wallet

```
address walletLibrary;
address owner;

function Wallet(address _owner) {
    walletLibrary = // address of wallet library
    walletLibrary.delegatecall(
        bytes4(sha3("initWallet(address)")), _owner);
}

function withdraw(uint amount) returns {
    walletLibrary.delegatecall(
        bytes4(sha3("withdraw(uint)")), amount);
}

function () payable {
    walletLibrary.delegatecall(msg.data);
}
```

However, in Ethereum, smart contracts can be deleted!
Locked ether

A user froze $170M by “accidentally” deleting the wallet library.

An attacker deletes the library.

No withdraws are possible.
Plan for Today

I. Security issues in smart contracts
   - The DAO hack
   - Parity bug #1
   - Parity bug #2

II. Security properties
   - Informal semantics of smart contracts
   - Writes after calls, unrestricted writes, locked ether

III. Automated security analysis
   - Securify https://securify.ch
Low-level code

- Solidity
- Vyper

High-level languages

compile

Low-level code

- Stack-based
- No types
- No functions

EVM code
# Ethereum Virtual Machine (EVM)

<table>
<thead>
<tr>
<th>Operation type</th>
<th>Description</th>
<th>OPCODEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>Encode calculations and numerical expressions</td>
<td>Add, Mul, Sub, Div, LT, EQ</td>
</tr>
<tr>
<td>Control-flow</td>
<td>Encode conditional jumps</td>
<td>Jump, JumpZ</td>
</tr>
<tr>
<td>Crypto</td>
<td>Compute hash functions</td>
<td>SHA3</td>
</tr>
<tr>
<td>Environment</td>
<td>Fetch block and transaction information</td>
<td>Balance, Caller, Callvalue, Calldataload, Gas, Gaslimit Timestamp, Difficulty, Blockhash</td>
</tr>
<tr>
<td>Memory / storage</td>
<td>Read from / write to memory and storage</td>
<td>MLtore, MLoad, SStore, SLoad</td>
</tr>
<tr>
<td>System</td>
<td>Message call into a contract</td>
<td>Call</td>
</tr>
</tbody>
</table>

[http://yellowpaper.io/]
System state

**Storage ($S$)**
- Persistent (initial storage is defined by the constructor)

**Memory ($M$)**
- Not-persistent (re-initialized before executing a transaction)

**Stack ($Q$)**
- Each element is 256 bits
- Size limited to 1024 elements

**Block information ($B$)**
- Number, timestamp, etc.
- Fixed for a given transaction
**Contract semantics**

**State**

\[
\sigma = (S, M, Q, B)
\]

- Storage, memory, and stack may change as the contract executes for a given transaction

**Transaction** \( T = (\text{caller}, \text{data}) \)

- Transaction sender (caller)
- Transaction data (data)

**Trace**

\[
(\sigma_0, op_0) \rightarrow_T (\sigma_1, op_1) \rightarrow_T \cdots \rightarrow_T (\sigma_{n-1}, op_{n-1}) \rightarrow_T (\sigma_n)
\]

- Each \( op_i \) is the next EVM opcode to be executed
- The set of all traces for a given contract defines the contract’s semantics
**Storage changes after calls**

A contract **does not change storage after calls** iff for any two traces that are identical up to a Call instruction:

\[
(\sigma_0, op_0) \rightarrow_T \cdots (\sigma_i, Call) \rightarrow_T (\sigma_j, op_j) \rightarrow_T \cdots \rightarrow_T \sigma_n
\]

\[
(\sigma'_0, op'_0) \rightarrow_{T'} \cdots (\sigma'_i, Call) \rightarrow_{T'} (\sigma'_j, op'_j) \rightarrow_{T'} \cdots \rightarrow_{T'} \sigma'_n
\]

if \(T.n.data = T'.n.data\) and \(\forall k \in [0..i]. \sigma_k = \sigma'_k \land op_k = op'_k\), then we have \(S_n = S'_n\), where \(\sigma_n = (S_n, M_n, Q_n, B)\) and \(\sigma'_n = (S'_n, M'_n, Q'_n, B)\).

- \(\sigma_j\) and \(\sigma'_j\) are the states after the Call returns, and \(\sigma_n\) and \(\sigma'_n\) are the final states

- the condition \(S_n = S'_n\) stipulates that the final state of the storage must not depend on how the Call executes
Unrestricted writes

- Consider a write to storage at a particular offset
- Intuitively, a write to storage offset is unrestricted if any user can trigger a write to that offset

More formally, a write to offset $o$ is unrestricted iff for any user address $a$, there is a transaction $T = (a, \_)$ and a trace

$$(\sigma_0, op_0) \to (a, \_) \cdots \to (a, \_) (\sigma_i, op_i) \to (a, \_) \cdots$$

such that $op_i = SStore(o, \_)$
Locked ether

A contract can receive ether if there is a trace where the contract’s balance increased:

\[(\sigma_0, op_0) \rightarrow_T \cdots \rightarrow_T (\sigma_n)\]

such that \(\sigma_0(Balance) < \sigma_n(Balance)\)

A contract cannot transfer ether if for any trace with Call instruction, the ether amount provided as input to the Call is zero. That is, for any trace:

\[(\sigma_0, op_0) \rightarrow_T \cdots \rightarrow_T (\sigma_i, op_i) \rightarrow_T \cdots\]

if \(op_i = Call(_, _, x, _)\) then \(x = 0\)

A contract locks ether iff it can receive ether and cannot transfer ether
More security properties...

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>🐷</td>
<td>Unexpected ether flows</td>
</tr>
<tr>
<td>🐛</td>
<td>Insecure coding, such as unprivileged writes</td>
</tr>
<tr>
<td>🔐</td>
<td>Use of unsafe inputs (e.g., reflection, hashing, ...)</td>
</tr>
<tr>
<td>⌚️</td>
<td>Reentrant method calls (e.g., DAO bug)</td>
</tr>
<tr>
<td>⬅️</td>
<td>Manipulating ether flows via transaction reordering</td>
</tr>
</tbody>
</table>
Challenges

Problem: Cannot enumerate all possible contract behaviors...
Automated techniques

Testing
- Report true bugs
- Can miss bugs

Dynamic analysis
- Report true bugs
- Can miss bugs

Automated verification
- Can report false alarms
  - No missed bugs

Note that properties such as unrestricted writes cannot be checked on a single trace.
Plan for Today

I. Security issues in smart contracts
   - The DAO hack
   - Parity bug #1
   - Parity bug #2

II. Security properties
   - Informal semantics of smart contracts
   - Writes after calls, unrestricted writes, locked ether

III. Automated security analysis
   - Securify https://securify.ch
Demo
Released in Fall 2017, so far:

- 95% positive feedback
- > 8K uploaded smart contracts
- > 600 users signed up for updates

Interesting discussions on Reddit

- mcgravier 22 points 12 days ago
  Seems almost too good to be true :) What are the limitations and how exactly does it work under the hood?

- AlexanderSupersloth 12 points 12 days ago
  Please, someone, humour a layman: how can a Turing complete language be formally verified?

  I thought formally verifiable languages were necessarily not Turing complete, and we can therefore not formally verify Solidity.

- pirapira Ethereum - Yoichi Hirai 3 points 11 days ago
  It's great that the authors of the tool are aware they are approximating the set of behaviors in the growing direction. That's the way to go if they seek safety properties without false-negatives. I'm interested how they compare their EVM semantics against other EVM implementations in the wild.
Under the hood

EVM Binary

00: 60
02: 5b
04: 42
06: 80
08: 90
0a: 56

Decompile

00: x = Balance
02: y = 0x20
04: If (x == 0x00)
06: MStore(y, x)
08: z = y
0a: goto 0x42

Securify Intermediate Representation

Static Analysis

MemTag(0x20, Balance)
MemTag(0x40, Const)
VarTag(z, Const)
VarTag(k, Gas)
Assign(s, 0x20)
Call(s{0x20}, k{Gas})

Securify Semantic Representation

Infer

Security patterns

Securify Report
Compliance and violation patterns

Insecure behaviors with respect to a property

Secure behaviors with respect to a property
Compliance and violation patterns

**Insecure** behaviors with respect to a property

**Violation** pattern (under-approximates unsafe behaviors)

**Secure** behaviors with respect to a property

**Compliance** pattern (under-approximates safe behaviors)
Under the hood

```plaintext
00: x = Balance
02: y = 0x20
04: If (x == 0x00)
06: MStore(y, x)
08: z = y
0a: goto 0x42
```

EVM Binary

Securify Intermediate Representation
From EVM to CFG over SSA

Control flow graph (CFG)
- Each node is a basic block (a sequence of instructions without jumps)
- An edge represents a jump from one basic block to another

Static single assignment form (SSA)
- Each variable is assigned exactly ones
Partial evaluation

\[
x := 10 \\
y := x + 20 \\
if (y > 0) \text{ goto L1} \\
<\text{else branch}> \\
\text{return} \\
<\text{then branch}>
\]

\[
x := 10 \\
y := x + 20 \\
<\text{then branch}> \\
\text{return} \\
<\text{else branch}>
\]
Partial evaluation

- Resolve jumps and improve the precision of the constructed CFG
- Resolve write offsets to storage/memory to improve analysis precision
Under the hood

- **Securify Intermediate Representation**
  - 00: x = Balance
  - 02: y = 0x20
  - 04: If (x == 0x00)
  - 06: MStore(y, x)
  - 08: z = y
  - 0a: goto 0x42

- **Securify Semantic Representation**
  - MemTag(0x20, Balance)
  - MemTag(0x40, Const)
  - VarTag(z, Const)
  - VarTag(k, Gas)
  - Assign(s, 0x20)
  - Call(s{0x20}, k{Gas})
## Semantic facts

<table>
<thead>
<tr>
<th>Semantic fact</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flow dependencies</strong></td>
<td></td>
</tr>
<tr>
<td>MayFollow ((l, l'))</td>
<td>The instruction at label (l) may follow that at label (l')</td>
</tr>
<tr>
<td>MustFollow ((l, l'))</td>
<td>The instruction at label (l) must follow that at label (l')</td>
</tr>
<tr>
<td><strong>Data dependencies</strong></td>
<td></td>
</tr>
<tr>
<td>MayDepOn ((x, t))</td>
<td>The value of (x) may depend on tag (t)</td>
</tr>
<tr>
<td>MustDepOn ((x, t))</td>
<td>The value of (x) must depend on tag (t)</td>
</tr>
<tr>
<td>DepBy ((x, t))</td>
<td>For different values of (t) the value of (x) is different.</td>
</tr>
</tbody>
</table>

A tag can be an instruction (e.g. Caller) or a variable.
Inference rules: *MayFollow*

Datalog program that defines the meaning of predicate *MayFollow*

\[
\begin{align*}
\text{MayFollow}(i, j) & \leftarrow \text{Follow}(i, j) \\
\text{MayFollow}(i, j) & \leftarrow \text{Follow}(i, k), \text{MayFollow}(k, j)
\end{align*}
\]

Derive input by declaring a predicate *Follow*(i, j) for each edge (i, j) in the CFG

1: x := 10
2: y := x + 20
3: y--; 4: return
5: y = 0
6: return

\[
\begin{align*}
\text{Follow}(1,2) & \\
\text{Follow}(2,3) & \\
\text{Follow}(2,5) & \\
\text{Follow}(3,4) & \\
\text{Follow}(5,6) &
\end{align*}
\]
Additional input facts

1: \( x = \text{Balance} \)
2: \( \text{Mstore}(0x20, x) \)
3: \( y = \text{MLoad}(0x20) \)
4: \( z = x + y \)

\[\text{Follow}(1,2)\]
\[\text{Follow}(2,3)\]
\[\text{Follow}(3,4)\]

\[\text{Assign}(x, \text{Balance})\]
\[\text{IsConst}(0x20)\]
\[\text{MStore}(2,0x20, x)\]
\[\text{MLoad}(3, y, 0x20)\]
\[\text{Op}(4, z, x)\]
\[\text{Op}(4, z, y)\]
Additional input facts

1: \( x = \text{Balance} \)
2: \( \text{Mstore}(0x20, x) \)
3: \( y = \text{MLoad}(0x20) \)
4: \( z = x + y \)

Derived from the Balance instruction

Input facts:

- \( \text{Follow}(1,2) \)
- \( \text{Follow}(2,3) \)
- \( \text{Follow}(3,4) \)
- \( \text{Assign}(x, \text{Balance}) \)
- \( \text{IsConst}(0x20) \)
- \( \text{MStore}(2,0x20, x) \)
- \( \text{MLoad}(3, y, 0x20) \)
- \( \text{Op}(4, z, x) \)
- \( \text{Op}(4, z, y) \)
Additional input facts

Code

1: x = \text{Balance}
2: \text{Mstore}(0x20, x)
3: y = \text{MLoad}(0x20)
4: z = x + y

Input facts

\text{Follow}(1,2)
\text{Follow}(2,3)
\text{Follow}(3,4)
\text{Assign}(x, \text{Balance})
\text{IsConst}(0x20)
\text{MStore}(2,0x20, x)
\text{MLoad}(3, y, 0x20)
\text{Op}(4, z, x)
\text{Op}(4, z, y)

Memory operations
Additional input facts

1: \( x = \text{Balance} \)
2: \( \text{Mstore}(0x20, x) \)
3: \( y = \text{MLoad}(0x20) \)
4: \( z = x + y \)

Capture that \( z \) is derived from \( x \) and \( y \)
Partial inference rules: *MayDepOn*

\[
\text{MayDepOn}(x, t) \leftarrow \text{Assign}(x, t)
\]
\[
\text{MayDepOn}(x, t) \leftarrow \text{Op}(_, x, x'), \text{MayDepOn}(x', t)
\]
\[
\text{MayDepOn}(x, t) \leftarrow \text{MLoad}(l, x, o), \text{isConst}(l, o), \text{MemTag}(l, o, t)
\]
\[
\text{MayDepOn}(x, t) \leftarrow \text{MLoad}(l, x, o), \neg\text{isConst}(l, o), \text{MemTag}(l, _, t)
\]

\[
\text{MemTag}(l, o, t) \leftarrow \text{MStore}(l, o, x), \text{isConst}(o), \text{MayDepOn}(x, t)
\]
\[
\text{MemTag}(l, \top, t) \leftarrow \text{MStore}(l, o, x), \neg\text{isConst}(o), \text{MayDepOn}(x, t)
\]
\[
\text{MemTag}(l, o, t) \leftarrow \text{Follows}(l, l'), \text{MemTag}(l', o, t), \neg\text{MStore}(l, o, _)
\]

- *MayDepOn* does not include the label \(l\) because variables are assigned exactly ones (do not change values)
- *MemTag* includes the label \(l\) as the dependencies for a given offset do change as the offset changes
Derived semantic facts

1: \( x = \text{Balance} \)
2: \( \text{Mstore}(0x20, x) \)
3: \( y = \text{MLoad}(0x20) \)
4: \( z = x + y \)

\[ \begin{align*}
\text{MayDepOn}(x, \text{Balance}) \\
\text{MayDepOn}(y, \text{Balance}) \\
\text{MayDepOn}(z, \text{Balance}) \\
\text{MemTag}(2, 0x20, \text{Balance}) \\
\text{MemTag}(3, 0x20, \text{Balance}) \\
\text{MemTag}(4, 0x20, \text{Balance})
\end{align*} \]
Summary: Static analysis via Datalog

Static analysis = fixed-point computation
- Pointer analysis
- Data-flow analysis
- Taint analysis
- (many others)

Express analysis declaratively in Datalog

Benefits:
- Declarative (concise spec of the analysis)
- Modular (can “glue” different pieces by merging the rules)
- Leverage existing scalable Datalog solvers

Checkout this project: https://flix.github.io/
Under the hood

```
MemTag(0x20, Balance)
MemTag(0x40, Const)
VarTag(z, Const)
VarTag(k, Gas)
Assign(s, 0x20)
Call(s{0x20}, k{Gas})
```

**Securify** Semantic Representation
Securify pattern language

\begin{align*}
(L&als) & \quad l \ ::= \text{(labels)} \\
(Var&es) & \quad x \ ::= \text{(variables)} \\
(T&ags) & \quad t \ ::= l \mid x \\
(Instr&) & \quad n \ ::= \text{Instr}(l,x,\ldots) \\
(Facts) & \quad f \ ::= \text{MayFollow}(l,l) \mid \text{MustFollow}(l,l) \\
& \quad \quad \quad \quad \mid \text{MayDepOn}(x,t) \mid \text{MustDepOn}(x,t) \\
& \quad \quad \quad \quad \mid \text{DetBy}(x,t) \\
(Patterns) & \quad p \ ::= f \mid \text{all n.p} \mid \text{some n.p} \mid p \&\& p \mid p \parallel p \mid \lnot p
\end{align*}
Example patterns: Restricted write

**Compliance pattern**

\[ \text{all } S\text{Store}(l, x, \_). \text{DetBy}(x, \text{Caller}) \]

**Violation pattern**

\[ \text{some } S\text{Store}(l, x, \_). \]
\[ \neg \text{MayDepOn}(x, \text{Caller}) \land \neg \text{MayDepOn}(l, \text{Caller}) \]

- The remaining patterns are encoded similarly
- Proofs establish a formal relation between the patterns and the security properties
Recap

Decompile

Securify Intermediate Representation

EVM Binary

00: 60
02: 5b
04: 42
06: 80
08: 90
0a: 56

\[ \text{x = Balance} \]
\[ \text{y = 0x20} \]
\[ \text{If (x == 0x00)} \]
\[ \text{MStore(y, x)} \]
\[ \text{z = y} \]
\[ \text{goto 0x42} \]

Static Analysis

Securify Semantic Representation

MemTag(0x20, Balance)
MemTag(0x40, Const)
VarTag(z, Const)
VarTag(k, Gas)
Assign(s, 0x20)
Call(s{0x20}, k{Gas})

Infer

Securify Report
Talk to me if you are looking for research projects / master thesis projects and summer internships in the area blockchain security.

Or send an email to: petar@chainsecurity.com
Summary

Three critical security issues
- The DAO exploit, Parity bugs #1 & #2

Properties and techniques
- Important to formally define the security properties
- Detecting / proving relevant properties is challenging via basic techniques (testing, dynamic analysis)

Automated verification of smart contracts
- Declarative static analysis, inference of semantic facts
- Property verification using compliance and violation patterns
Next lecture

April 2\textsuperscript{nd}:  \textbf{Easter break}

April 9\textsuperscript{th}:  \textbf{Start of Part III}
\textit{Machine learning attacks and defenses}

1. Introduction
2. Attacks: adversarial examples
3. Analysis and defenses
References

Securify: https://securify.ch

Interesting projects in the are of declarative static analysis:

- Flix
- PQL
- Pidgin