Ethainter:
A Smart Contract Security Analyzer for Composite Vulnerabilities

Lexi Brent, Neville Grech, Sifis Lagouvardos,
Bernhard J Scholz, Yannis Smaragdakis
(PLDI 2020)

Presented by Jaehwang Jung

Ethereum primer

- Smart contract = distributed database with a program to make transaction
- Ethereum network
  - Accounts identified by its addresses.
    - Ethereum balance
    - optional **code** (→ contract)
    - **storage**: 256-bit address space (very sparse), each slot contains 256-bit word
  - Send messages to transfer Ethereum balance or to call contract functions
- Running a contract
  - Launch a local instance of Ethereum Virtual Machine (EVM)
  - Run the code with the input (code = EVM bytecode)
  - Propose the transaction result to the Ethereum network
- Usually the contract is written in high-level languages like Solidity

Smart contract example: Token contract

```
contract Token {
    mapping (address => uint256) balances;
    ...
    function transfer(address to, uint256 value) {
        require(value <= balances[msg.sender]);

        balances[msg.sender] = balances[msg.sender].sub
        balances[to] = balances[to].add(value);
    }
    ...
}
```

```
0 |-> PUSH ( 1 , 128 )
2 |-> PUSH ( 1 , 64 )
4 |-> MSTORE
5 |-> PUSH ( 1 , 4 )
7 |-> CALLDATASIZE
8 |-> LT
9 |-> PUSH ( 2 , 87 )
12 |-> JUMPI
13 |-> PUSH ( 1 , 0 )
15 |-> CALLDATALOAD
...
```

Security-critical operations

- `selfdestruct(addr)`: destroy the contract and send eth balance to `addr`
  - just to remove the contract
  - to deploy new version of contract: bug fix, new feature, … (contract code is immutable!)
- `delegatecall(addr)`: take the code from `addr` and run it on the current contract (modifies the current contract's state)
  - proxy pattern, ...

Very dangerous.

These operations must not be accessible to users that are not the "owner" of the contract.

Guards

```
contract OwnedContract {
    address owner;

    constructor() public {
        owner = msg.sender;
    }
    function kill() public {
        require(msg.sender == owner);
        selfdestruct(owner);
    }
}
```

Composite Vulnerabilities

```
contract Victim {
    address owner;
    mapping(address => bool) admins;

    function kill() public {
        require(admins[msg.sender]);
        selfdestruct(owner);
    }
    function registerAdmin(address admin) public {
        admins[admin] = true;
    }
}
```

Challenges of analyzing Ethereum smart contracts

- Composite vulnerabilities
  - Guards can be rendered useless! e.g. `Victim` contract
- EVM bytecode: How to extract high-level properties from low-level bytecode?
  - Security-critical operations
    - Not all sensitive operations are as simple as `selfdestruct`, e.g. `registerAdmin()`
  - Guards
    - Not all checks are guards.
    - Guard are complex
      - They often use storage data structure lookups with `msg.sender` variable e.g. `require(admins[msg.sender])`.
      - … why is this complex?

```
0 |-> PUSH ( 1 , 128 )
2 |-> PUSH ( 1 , 64 )
4 |-> MSTORE
5 |-> PUSH ( 1 , 4 )
7 |-> CALLDATASIZE
8 |-> LT
9 |-> PUSH ( 2 , 87 )
12 |-> JUMPI
13 |-> PUSH ( 1 , 0 )
15 |-> CALLDATALOAD
...
```

Compiling data structures to EVM bytecode

Entries of mappings are scattered over the entire 256-bit storage space and their locations are determined by the hash of key.

```
mapping(address => bool) admins;
require(admins[msg.sender]);
```

Address of `admins[msg.sender]` in the storage

= `hash(msg.sender ++ slot(admins))` → 256-bit address

SHA3

Byte array concatenation

ID of each storage variable, e.g. 3

Ethainter

Solution: specialized taint analysis

- Composite vulnerabilities → Taint propagation via storage
  - If taint propagates into a guard, that guard becomes useless (non-sanitizing)
- Too low-level → Infer high-level properties from the bytecode:
  - Detect sinks (security-critical operations) and sanitizers (guards)
  - Detect variables related to data structure operations

Taint source and sink

$$(\textsc{LoadInput})\quad \frac{x \;:=\; \texttt{INPUT}()}{\downarrow^{I} x}$$

$\downarrow^{I} x$	Variable x is tainted from input.
$\downarrow^{T} x$	Variable x is tainted from storage.

$$(\textsc{Operation-1})\quad \frac{x \;:=\; \texttt{OP}(y, *) \qquad \downarrow^{I/T} y}{\downarrow^{I/T} x}$$

$$(\textsc{Violation})\quad \frac{\texttt{SINK}(x) \qquad \downarrow^{*} x}{\textsc{violation}}$$

Storage taint (1)

$\downarrow^I x$	Variable x is tainted from input.
$\downarrow^T x$	Variable x is tainted from storage.

from local variable f to storage address t

$C(x) = v$ Variable x is inferred to have constant value v .

$$(\textsc{StorageWrite-1})\ \frac{\mathtt{SSTORE}(f, t) \quad \downarrow^* f \quad C(t) = v}{\downarrow^T S(v)}$$

Storage location with constant address v is tainted.

Storage taint (2)

If a storage write destination is tainted, all **statically known** storage locations get tainted.

$$(\textsc{StorageWrite-2})\ \frac{\mathsf{SSTORE}(f,t) \qquad \downarrow^{*} f \qquad \boxed{\downarrow^{*} t}}{\forall i : \downarrow^{T} S(i)}$$

from storage address f to local variable t

$$(\textsc{StorageLoad})\ \frac{\mathsf{SLOAD}(f,t) \qquad \downarrow^{T} S(v) \qquad C(f) = v}{\downarrow^{T} t}$$

Guards (1)

A guard sanitizes input taints.

$\downarrow^I y$ $\quad x \;:=\; \mathsf{GUARD}(p, y)$ $\quad \Rightarrow \quad$ $\cancel{\downarrow^I x}$

tainted from input. *checked in predicate p to sanitize y*

A guard sanitizes input taints, … **unless...**

$$(\textsc{Guard-2})\ \frac{x \;:=\; \mathsf{GUARD}(p, y) \qquad \downarrow^I y \qquad \boxed{\updownarrow p}}{\downarrow^I x}$$

guard predicate p fails to sanitize.
(Non-sanitizing guard)

Guards (2)

$$(\textsc{Guard-1})\quad \frac{x \;:=\; \mathsf{GUARD}(*, y) \qquad \downarrow^{T} y}{\downarrow^{T} x}$$

Guards do not sanitize storage taints.

Non-sanitizing guards (1)

Variable z is an alias for a constant storage slot v.

$$(\textsc{Uguard-T})\ \frac{p\ :=\ (\mathtt{sender}\ =\ z) \qquad z \sim S(v) \qquad \downarrow^{T} S(v)}{\ddagger p}$$

A guard predicate p is non-sanitizing
if it refers to a tainted storage location.

Non-sanitizing guards (2)

Variable y and z are not related to data structure entries that are related to `msg.sender`.

$$\text{(UGUARD-NDS)}\ \frac{p \ := \ (y \ = \ z) \qquad \neg\mathrm{DS}(y) \qquad \neg\mathrm{DS}(z)}{\ddagger p}$$

A guard predicate p is non-sanitizing
if it does not lookup a storage data structure with `msg.sender`.

Modeling data structures lookups for guards

Guards usually perform data structure lookups using `msg.sender` as a key.

$$(\text{DS-SenderKey}) \quad \frac{}{\mathsf{DS}(\mathsf{sender})}$$

If a value is related to data structure entries that are related to `msg.sender`, then the derived addresses/values are also related.

$$(\text{DS-Lookup}) \quad \frac{x \ := \ \mathsf{HASH}(y) \qquad \mathsf{DS}(y)}{\mathsf{DSA}(x)}$$

$$(\text{DS-AddrOp-1}) \quad \frac{\mathsf{DSA}(y) \qquad x \ := \ \mathsf{OP}(y, *)}{\mathsf{DSA}(x)}$$

$$(\text{DSA-Load}) \quad \frac{\mathsf{DSA}(x) \qquad \mathsf{SLOAD}(x, y)}{\mathsf{DS}(y)}$$

Inferring guards

```
function kill() public {
    require(admins[msg.sender]);
    selfdestruct(owner);
}
```

If a check dominates a use of a tainted variable, then it is considered a guard for that variable.

Inferring sinks

```
function kill() public {
    require(admins[msg.sender]);
    selfdestruct(owner);
}
function registerAdmin(address admin) public {
    admins[admin] = true;
}
```

A variable is sink if:

it's used as a guard for another tainted variable

it's read from storage

$$\frac{* \;:=\; \mathsf{GUARD}(\mathsf{sender} = z, x) \qquad \downarrow^{I/T} x \qquad z \sim S(*)}{\mathsf{SINK}(z)}$$

Balancing completeness and precision (1)

$$(\textsc{StorageWrite-2})\ \frac{\mathsf{SSTORE}(f, t) \qquad \downarrow^{*} f \qquad \downarrow^{*} t}{\forall i : \downarrow^{T} S(i)}$$

Tainted write propagates the taint to all constant storage locations.
→ **more complete, but less precise**

Balancing completeness and precision (2)

$$(\textsc{UGuard-T})\ \frac{p\ \texttt{:=}\ (\texttt{sender}\ \texttt{=}\ z) \qquad \boxed{z \sim S(v)} \qquad \downarrow^{T} S(v)}{\nmid p}$$

Only the statically known taints nullify a guard.
→ more guards are considered effective
→ less complete, but more precise

Balancing completeness and precision (3)

$$\text{(DS-LOOKUP)}\ \frac{x := \mathsf{HASH}(y) \qquad \mathrm{DS}(y)}{\mathrm{DSA}(x)} \qquad \text{(DS-ADDROP-1)}\ \frac{\mathrm{DSA}(y) \qquad x := \mathsf{OP}(y, *)}{\mathrm{DSA}(x)} \qquad \text{(DSA-LOAD)}\ \frac{\mathrm{DSA}(x) \qquad \mathsf{SLOAD}(x, y)}{\mathrm{DS}(y)}$$

→ Over-approximate the relevance to data structure lookups

$$\text{(UGUARD-NDS)}\ \frac{p := (y = z) \qquad \neg\mathrm{DS}(y) \qquad \neg\mathrm{DS}(z)}{\ddagger p}$$

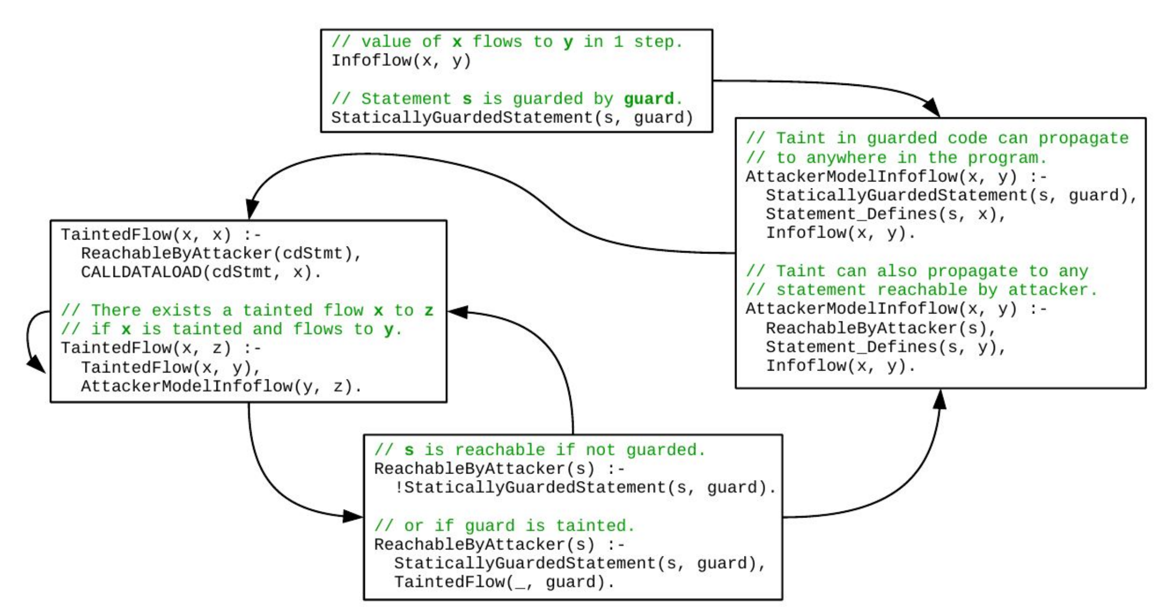
More guards are considered to be related to data structure lookups

→ more guards are considered effective

→ **less complete, but more precise**

Implementation

- Complex flow model to support real-world contracts (hundreds of rules)
- Souffle Datalog engine
- other various analysis: const prop, control dependency, …



Evaluation

- Ethainter is an effective static analysis, practically relevant, flagging a usefully large number of contracts, with high precision and completeness.
  - Manually inspected randomly chosen contracts.
    - Much better results than other tools like Securify V1, V2 (static analysis), teEther (symbolic execution)
  - Automated exploitation generation for some simple types of vulnerabilities
- Ethainter analyzes all 240K unique contracts on the blockchain, corresponding to a total of 38 million lines of 3-address code, in 200 CPU hours (5 secs / contract).

- Good balance between precision and completeness

  - Significantly less precise/complete if the current balance is broken.

Summary

Ethainter detects composite vulnerabilities in smart contracts using information flow analysis for tracking tainted values

- which **models key domain concepts**, such as sanitization via guards and taint through persistent storage,
- and is **finely tuned** for precision, completeness and scalability.