Towards Fully Automatic Logic-Based Information Flow Analysis: An Electronic-Voting Case Study
Lecture Notes

Introduction

- Common information flow analysis approaches:
  - Logic Based - precise, not automatic, difficult to use for large programs
  - Over approximation - type-based, dependency graph or abstract interpretation, automatic, lack precision
- This approach: based on self composition and symbolic execution that uses abstract interpretation
- Approach summary:
  - Generate specifications for unbounded loops and recursive method calls
  - Use KEG tool to detect policy violations (information flow leaks)
- This approach is applied to two versions (one correct and one faulty) of an e-voting system

Symbolic Execution Review:

- Main idea: run program with symbolic values instead of concrete ones
- Result: Symbolic execution tree
  - Branches: Contain previous path conditions and added branch condition
  - Nodes: Symbolic State
  - No unbounded loops or recursive method calls means the tree is finite and covers all possible executions
  - Unbounded loops or recursive method calls means the tree will be infinite

Dynamic Logic

(Source: A Theorem Proving Approach to Analysis of Secure Information Flow by Ádám Darvas, Reiner Hähnle, and David Sands)

- $\langle p \rangle$ is the state that is reached by running program p
- $\langle p \rangle \Phi$ - p terminates in a state in which Φ holds.
- $\Phi \rightarrow \langle p \rangle \Psi$ is valid if for every state s satisfying precondition Φ a run of p starting at s terminates in a state in which Ψ holds.
- Dual operator $[\,]$: $[p]\Phi \equiv \neg \langle p \rangle \neg\Phi$
- $r \doteq l$ the value r of l.
- Secure information flow in DL: $\forall l.\ \exists r.\ \forall h.\ \langle p \rangle\ r \doteq l$. "When starting p with arbitrary values l, the value r of l after executing p is independent in the choice of h.
- Secure information flow in DL, another way: $\forall l, l', h, h'.(l \doteq l' \rightarrow \langle p\{l,h\}; p\{l',h'\} \rangle\ l \doteq l')$ "Running two instances of p with equal low-security values and arbitrary high-security values, the resulting low security values are equal too."

Noninterference

- Given a program p and a set V of variables partitioned into two sets H and L. $H \not\rightarrow L$ if there is no information flow from H to L.
- $H \not\rightarrow L$ iff any two execution states of p starting in initial states that coincide on L also terminate in two states that coincide on L
- Eq 1: Formalism (self composition), given p' a copy of p, $H \not\rightarrow L \equiv \{L \doteq L'\} p(V); p'(V')\{L \doteq L'\}$ (If the two low-security values are equal before execution of p and p', they will be equal after execution.)
- Issue with this formalization: requires p to be analyzed twice, but this can be done using symbolic execution.
- Notation:
    - SE_p is the Symbolic Execution tree of p
    - For each symbolic execution path i, the path condition of that path is denoted by pc_i.
    - For some symbolic input i and variable v, f_i^v maps from the symbolic inputs to the symbolic final value of v
    - N_p is the number of symbolic execution paths of SE_p
- We only need to symbolically execute p once and represent two executions of p and p' by two symbolic execution paths of SE_p with different symbolic inputs V and V'.
- SMT (Satisfiable Modulo Theory - good for inputting into theorem provers) formula (same meaning as Eq 1): $\wedge_{0 \le i \le j \le N_p}((\wedge_{v \in L} v \doteq v') \wedge pc_i(V) \wedge pc_j(V') \Rightarrow \wedge_{l \in L} f_i^l(V) \doteq f_j^l(V'))$
- Negation of this formula: $\vee_{l \in L} \vee_{0 \le i \le j \le N_p} Leak(H, L, l, i, j)$, where $Leak(H, L, l, i, j) = (\wedge_{v \in L} v \doteq v') \wedge pc_i(V) \wedge pc_j(V') \wedge \neg(f_i^l(V) \doteq f_j^l(V'))$
- If this leak function is satisfiable, there exists a forbidden information flow from some variables of H to a variable $l \in L$. Otherwise, p is secure with respect to the noninterference policy.
- Possible issue: if p contains unbounded loops or recursive method calls, SE_p becomes infinite. However, there is a method that represents loops and method calls as corresponding single nodes of a symbolic execution tree and use loop invariants and method contracts to contribute to relevant path conditions and to the representation of the tree. Therefore our goal is to specify these loop invariants and method contracts.
- Implementation: Used KEG (KeY Exploit Generation) software, which when given information flow policies, can detect leaks and generate exploits in the form of JUnit tests. How KEG works: symbolically execute method, compose all insecurity formulas, find models satisfying formulas, then generate JUnit tests from found models.

Generating Invariants using Abstract Interpretation

- JavaDL calculus used by KeY
    - Updates - used to cover state changes in variables and to model the heap memory as a program variable

    - Updates are created in symbolic execution whenever a field or variable changes its value
    - $x := t$ means program variable x is updated to t
    - $U \parallel U'$ are parallel updates
    - $\{\cdot\}\,\cdot$ applies updates to formulas. Example: $\{x := t\}(2x)$ is equal to $2t$
    - $heap := store(heap(a, i, t))$ updates $a[i] = t$
- Abstract Interpretation:
    - Lose precision within the analysis for more automation
    - Combined with symbolic execution, allows for abstract symbolic values, which represent items in a specific set of concrete values called the abstract element. The abstract elements make up the abstract domain.
    - Property: if a_1 and a_2 are symbolic values, $a_1 \sqcup a_2$ is a set which encompasses at least all the concrete values of the two input values.
    - Information loss:
        - $a_1 \sqcup a_2$ may contain more values than a_1 and a_2 individually
        - Abstracting a set of concrete values to an abstract value means trying to find an abstract value that represents all of the concrete values
    - Abstract functions:
        - Used to express within updates properties of variables
        - $\gamma_{\alpha,z}$ is an abstract function, where α is the abstract element and $z \in \mathbb{Z}$ identifies the abstract function, example: $x := \gamma_{>,1}$ sets x to some positive value
        - The description of each α is contained in the characteristic function X_α
- Generating Loop Invariants
    - Abstraction of variables: Loop is symbolically executed once, and symbolic program states are joined with the initial symbolic program state, for each variable, the value in the update is abstracted, then all abstract elements are joined. This is repeated until another iteration doesn't produce a weaker update.
    - Abstraction of arrays: Arrays are split into two parts: potentially modified and unmodified. If a sequence of array accesses in monotonously increasing or decreasing, this split moves each iteration in the same direction. The invariant can be specified this way: $\forall i.\ (initial \leq i \land i < id) \rightarrow \mathrm{X}_\alpha(arr[i])$ where id is the current index and $initial$ is the value of id before the loop.
    - The invariants are then translated into JML(java modeling language)

E-Voting Case Study

- Game: Adversary provides 2 vectors of choices of voters c_0 and c_1 such that they all yield the same result. Afterwards, the voters vote according to c_b for a secret bit b, and see if the adversary can tell which c_b they followed.
- For leak detection, they modify the method so that the results don't include the first vote in each c. Their method correctly detected this leak.