Secure Information Flow Summary

§1 Introduction

In the old days, program correctness meant that software outputs did not go wrong (a program not crashing), while computer security was to prevent bad things from happening (avoid unauthorized accesses to secure sensitive resources). For this context, a buffer overflow is a problem of program correctness.
On the other hand, security comes with the following definitions

- Security: absence of unauthorized access to or handling of system state
- Confidentiality: prevention of unauthorized disclosure of information
- Integrity: prevention of unauthorized change of information

Conventional security means that a program works as a black box that interacts with the environment, by the usage of encryption, firewalls or process level privileges. Usually, Operative Systems enforce security at the system call layers, but it is still hard to control applications when they're not making system calls. That's why modern attacks work at application level, by using exploits or protocol vulnerabilities.
Modern applications process sensitive data, which could be unknowingly shared by untrusted libraries or third party applications. Conventional security is not always enough, which is reason enough to apply a more low level solution, with languages, compilers and runtime systems methods.
Examples of security threats are, scarewares (two step malware), malwares, fraud or identity theft.
Security and program correctness are undecidable problems by several factors, but there are many tools developed for verifying program correctness that can also be used for program security, such as

- Model checking: it simulates the run of a program for every possible input until either an attack is found or some resource usage limit is reached.
- Static analysis (type system, abstract interpretation): it relies on a provably sound abstraction of the program semantics, which makes the security property of interest decidable in an abstract domain. It doesn't give a certain answer though, because if the program is verified, then it is safe, but if the program is not validated, the there is either a flaw or the abstraction is not enough.
- Dynamic analysis (runtime monitoring): same as static analysis, but the program run is verified during execution

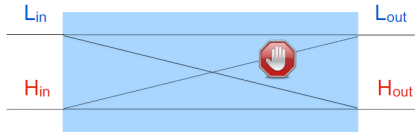
The main Goal here is to verify whether or not a program enables information flow from high variables to low variables.

Lattices

Lattices are a structure that enforces order or partial ordering. All finite lattices (with finite elements) are complete, which means there are partial ordering, joining and merging operations and TOP and BOTTOM elements defined.

Non interference: Secret inputs of programs must not influence public outputs. For type systems we assume a basic programming model (Batch Job model):

- All inputs are specified at the beginning of programs
- Confidentiality is defined by partitioning a programs variables in security levels.
- Low variables are public information, high variables are private information.



Mechanisms for signaling information through a computing system are know as channels. Channels that exploit a mechanism whose primary purpose is not transferring information are called covert channels. Examples are

- Termination channels: signal information through the termination or non termination of a computation ("while h=1 do skip")
- Timing channels: signals information through the time at which an action occurs rather than through the data associated with the action ("if h=1 then C_long else skip")
- Probabilistic channel: signal information by changing the probability distribution of observable data (multithread program with "leak=secret —— leak=random(100) ", which thread goes first?)
- Power channels ad resource exhaustion channel.

2 Information Flow Control (IFC?)

In general we can observe that it is easy to check information release, and hard to check information propagation. For verifying whether a program guarantees confidentiality of secret data we need a formal definition, precise description of the program semantic and verification techniques.
Definitions

- L-equivalence: Two memories u and v are L-equivalent (written $u \sim_L v$) if they have the same values for all L variables.
- Non-interference for type systems: Program c satisfies non-interference if
  $\forall u, v, u', v'\ u \sim_L v, (c,u) \to u', (c,v) \to v' \implies u' \sim_L v'$
  This definition is termination insensitive, timing insensitive and doesn't take into account intermediate observations.

Operational semantics are a formal description for a program execution. A program c is executed under a memory u which maps identifiers to values. Expressions are evaluated atomically, letting $u(e)$ denote the value of e in memory u. Structural operational semantics is defined in term of transition relations between configurations. A configuration is either a pair (c,u) or a memory u.

$$\textsc{Update}\ \frac{x \in dom(\mu)}{(x := e, \mu) \to \mu[x \mapsto \mu(e)]}\quad \text{(Memory update)} \qquad \textsc{No-Op}\ (\text{skip}, \mu) \to \mu$$

$$\textsc{Branch True}\ \frac{\mu(e) \neq 0}{(\text{if } e \text{ then } c_1 \text{ else } c_2, \mu) \to (c_1, \mu)} \qquad \textsc{Branch False}\ \frac{\mu(e) = 0}{(\text{if } e \text{ then } c_1 \text{ else } c_2, \mu) \to (c_2, \mu)}$$

$$\textsc{Loop True}\ \frac{\mu(e) \neq 0}{(\text{while } e \text{ do } c, \mu) \to (c; \text{while } e \text{ do } c, \mu)} \qquad \textsc{Loop False}\ \frac{\mu(e) = 0}{(\text{while } e \text{ do } c, \mu) \to \mu}$$

$$\textsc{Sequence 1}\ \frac{(c_1, \mu) \to \mu'}{(c_1; c_2, \mu) \to (c_2, \mu')} \qquad \textsc{Sequence}\ \frac{(c_1, \mu) \to (c_1', \mu')}{(c_1; c_2, \mu) \to (c_1'; c_2, \mu')}$$

Type system

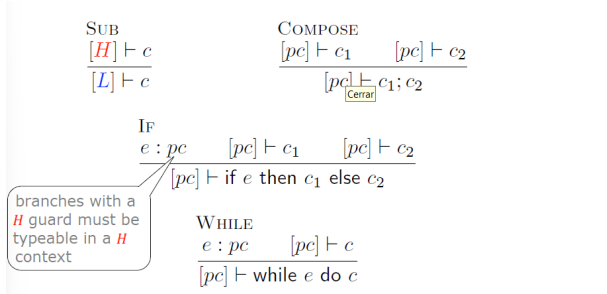
Type systems are a well-known toll to enforce non-interference, by analyzing just the syntax of a program. As mentioned, if a program is accepted, then it is secure, and if not, it might be insecure or the type system might not be precise enough.

Overview

- First, we classify expression as "High" if it has any H variable (or in any case really), and L otherwise
- Next, prevent explicit flows by forbidding a H expression from being assigned to a L variable
- Finally, prevent implicit flows by forbidding a guarded command with a H guard from assigning to L variables
- Notes
  - Any expression ca be typed H
  - Assignments to H variables or empty commands can be typed in any context (H or L)

$$\textsc{Exp-High}\ exp : H \qquad \textsc{Exp-Low}\ \frac{h \notin Vars(exp)}{exp : L}$$

$$\textsc{Skip}\ [H] \vdash \text{skip} \qquad \textsc{Assign-L}\ \frac{e : L}{[L] \vdash l := e} \qquad \textsc{Assign-H}\ [H] \vdash h := e$$



Lemmas

- Simple Security: if $e : \tau$, then e contains only variables of level τ or lower
- Confinement: If $[\tau] \vdash c$, then c assigns only to variables of level τ or higher
- Invariance: If $(c, u) \to u'$ and x is notr assigned to in c, then $u(x) = u'(x)$
- Memory: If $(c, u) \to u'$ then $\text{dom}(u) = \text{dom}(u')$

3 Declassification

Information release: Some applications crucially rely on intended information leaks, like password checking or location information. Allowing leaks might compromise confidentiality, specially when non interference is violated.

Example: Average Salary

Intention: Release average salary of a company

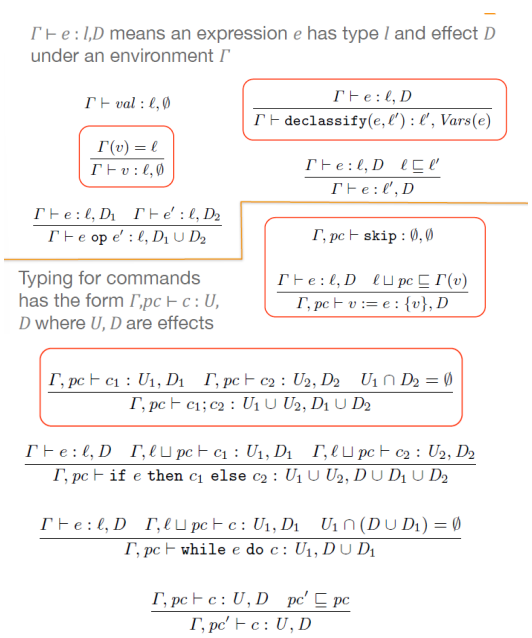
- Without declassification
  - Command: avg = $(h_1 + \cdots + h_n)/\ n$;
  - Rejected by non interference
- With declassification
  - Command: avg = declassify($(h_1 + \cdots + h_n)/\ n$, L);
  - Only declassified data and no further information is released.
  - Expressions under declassify are called "escape hatches"

Delimited release

The only way to learn information is via escape hatches. Definition: Let command c achieve delimited release if
$\forall u_1, u_2, v_1, v_2\ u_1 \sim_L u_2 \wedge (u_1, c) \to v_1 \wedge$
$(u_2, c) \to v_2 \wedge \forall i, u_1(e_i) = u_2(e_i) \implies v_1 \sim_L v_2$.
For programs without declassification, delimited release coincides with non-interference.
Security type and effect system for delimited release is similar to the one for non interference. Extensions are

- the instruction declassify(h, L) is given the type L.
- prevent new information from flowing into variables used in hatch expressions.

Type System for delimited release



Notes

- Programs that input high data are always rejected
- Delimited release allows us to characterize what information is released, but not where the release occurs

4 Declassification, Integrity and DLM

Definition: Localized delimited release allows release of data any time after it is declassified.
Definition: Gradual release states that between to declassify commands no secret data must be released.

Attacker knowledge

Attackers

- can observe L-projection of initial memory.
- can observe L-events (assignments).
- can infer secret values based on observations

Formally, for a program c, the L-projection of initial memory M_L^0 and a sequence of low events $\vec{l}$; the current knowledge is the set of all memories that lead to observation $\vec{l}$ with initial memories that agree with M_L^0; the initial knowledge is the set of all memories that lead to termination with initial memories that agree with M_L^0.
If initial knowledge is equal to the current knowledge, non interference is satisfied.
Integrity tells if we can safely answer where the data is coming from. We can define a security lattice of Confidentiality and Integrity, by replacing the usual $\sim_L$ relation with $\sim_l$, which means that u and v are equivalent at level l or lower.

Robust declassification:H-integrity code is a program in which some statements are missing, replaced by holes. These holes are instantiated with attacks a. Definition
$\forall u, v, a, a'\ (c[a], u) \sim_{L,L} (c[a], v) \implies (c[a'], u) \sim_{L,L} (c[a'], v)$

6 Control Flow

Control Flow Graph is defined as a graph in which every node is a statement or a basic block, and edges represent the control flow of a program. It can also define a function v which defines the condition under an edge is followed.
A complete lattice defines the five-tuple $\mathbf{L} = (L, \sqsubseteq, \sqcup, \sqcap, \bot\ \top)$. A semi lattice defines only the first 4 elements of a complete lattice plus a set of functions F that have the following properties: include the identity function, follow monotonicity, are closed under composition, and meet pointwise infimum.
A fixed point x for a operator f in a semi-lattice L holds that $f(x) = x$. A Maximal Fixed Point x holds that
$\forall y \in L : f(y) = y \to \sqsubseteq x$

foreach $n \in CFG$ **do**
  $A[n] = \bot$
od
do
  $change = false$
  foreach $n \in CFG$ **do**
    $temp = \bigsqcap_{q \in pred(n)} F_q(A[q])$
    if $temp \neq A[n]$
      $change = true$
      $A[n] = temp$
    fi
  od
until $!change$

Reaching Definitions can be the subset of functions F. Here

- Gen(n) = Def(n)
- Kill(n) = All definitions in n that are also defined somewhere else
- IN(n) = Union of all OUT(p) for p being a predecessor of n. It's initialized as empty for all n.
- OUT(n) = (IN(n) - Kill(n)) $\bigcup$ Gen(n)

This is done in a post-order manner until there's no change in the sets.
A data dependence graph has the same nodes as a CFG, but the edges are defined as a data dependency between node x and y $(x \to y)$ if:

- There's a variable v that meets $v \in \text{def}(x)$ and $v \in \text{use}(y)$.
- v is part of the reaching definitions (IN) of y.

Slicing on a code is mainly done by a certain criteria, and it is valid if the slice is still a valid program and when both original program and slice have the same halting behaviors.
A Program Dependence graph is the merge of both Data Dependence graph (done by reaching definitions) and Control Dependence graph.

7 Slicing-Based IFC

A node x is post-dominated by node y if all paths from x to the exit node through y.

A control dependence graph is build by using all nodes in a control flow graph, and the edges from x to y are defined if

- there exists a path p from x to y in the CFG, such that y post-dominates every node in p, except for x
- x is not post-dominated by y

A post dominator tree can also be build by reversing the original CFG, and computing the dominator tree, that is build by the inmediate dominators of every node.

Slicing can be done in a Forward or Backward manner (following successors and predecessors).

Summary edges are added into program dependence graphs that have interprocedural dependencies. In this case, we apply Two phase slicing

- In the first phase: Do not descend into called methods, mark omitted edges for later phase. Traverse summary edges instead.
- In the second phase: Starting with the omitted edges, do not reascend into calling method. Still traverse summary edges.

Static and Dynamic Analysis

As stated before, from static analysis one can tell for sure if a program is safe, but in the contrary case, the conclusion is ambiguous. Dynamic analysis always will give a certain answer, and will extract dependencies at runtime

Taint analysis is done to check if untrusted sources of data could get to critical code. A sensitive update is done when a public variable is updated in a high context

In this context, the flow sensitive type system is presented

$$pc \vdash \Gamma\ \{\texttt{skip}\}\ \Gamma \qquad \frac{\Gamma \vdash e : t}{pc \vdash \Gamma\ \{x := e\}\ \Gamma[x \mapsto pc \sqcup t]}$$

$$\frac{pc \vdash \Gamma\ \{c_1\}\ \Gamma' \qquad pc \vdash \Gamma'\ \{c_2\}\ \Gamma''}{pc \vdash \Gamma\ \{c_1; c_2\}\ \Gamma''}$$

$$\frac{\Gamma \vdash e : t \qquad pc \sqcup t \vdash \Gamma\ \{c_i\}\ \Gamma' \qquad i = 1, 2}{pc \vdash \Gamma\ \{\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\}\ \Gamma'}$$

$$\frac{\Gamma \vdash e : t \qquad pc \sqcup t \vdash \Gamma\ \{c\}\ \Gamma}{pc \vdash \Gamma\ \{\texttt{while}\ e\ \texttt{do}\ c\}\ \Gamma} \qquad \frac{\Gamma \vdash e : t \qquad pc \sqcup t \sqsubseteq \ell}{pc \vdash \Gamma\ \{\texttt{output}_\ell(e)\}\ \Gamma}$$

A monitor μ inspects the program execution at each step, reacts according to a policy, and can terminate the program execution, suppress output or print default values.
Properties

- Definition <<c(ode), m(emory)> $|_\mu$ cfmg>
- No look ahead: c' does not affect the monitor decisions regarding c in c; c'. Monitor cannot skip or rewrite instructions ahead of time.
- No look aside
- Soundness: Attacker cannot learn the secret in polynomial time in size of secret (termination-insensitive noninterference)
- Permisiveness: If a program type checks then the monitor does not modify its behavior

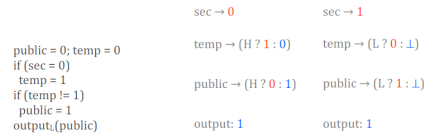
A monitor μ cannot fulfill these four properties at the same time. Semantics have the form $\frac{1}{2 \to 3}$, where 1 is an optional pre condition, and $2 \to 3$ is the change in the Context and Computer stack (Γ and Γ_s).

10 Secure Multiexecution

Secure multiexecution is a form of assuring security by executing a same program in both Low and High context. For this, we present a two level lattice, with the definitions of "can see" the content of a high context, and "there is a presence" of high context. Here declassification works as a black-box approach, named stateful declassification. For information release, we present the following functions

- Policies:of the form $\pi(\text{event}n) : Nothing|Project(n)$
- Project(event n): it presents to the low observer the content of an event n.
- Nothing: Event not visible for low observers.
- Leaks, which can be of the form r(state s, event n) = (s', Unchanged — Release n')
  - Unchanged: no new info is released.
  - Release n': releases info n' to observers.

Faceted execution



11 Hybrid Information Flow Control

Explicit Flow implies a flow due to assignments. In program analysis is the data dependency.
Implicit flow comes from the control structure. One way to determine if there is a implicit flow, is by reducing the context of the inmediate post dominator of the statements to prove.
For intra-procedural implicit flow, there is a notation for a pc-stack, which pushes into it the pc of the called method, and then pops off when the method returns.

12 Reactive Non-Interference

In a Batch-Job model input is given as a content or stream of low and high variables, and the output is content of only low variables. For this, we need a new definition for non-interference. We define reactive systems as a five-tuple (Consumers, Producers, Input, Output, transition)
Some relations between streams are

- $\approx^{NC}$: where both streams don't conflict in each others events.
- $\approx^{ID}$: where both streams have the are not conflicting and are still in relation if one of them has a unimportant event.
- $\approx^{CP}$: where both streams are not conflicting but now do care about unimportant events.
- $\approx^{CPCT}$: all from before, but now size is also taken into care.