Security: Applications & Information Flow

Klaus v. Gleissenthall

VRIJE UNIVERSITEIT AMSTERDAM

Where are we?

- We're in the business of proving things about programs
- First we used Hoare Logic and VCGen
- But this requires supplying loop invariants
- This can be quite tedious!
- Next, we wanted to compute loop invariants (semi-)automatically
- We looked at Horn clauses
- Horn clauses represent constraints on unknown relations called **queries**
- We used weakest preconditions to translate programs into Horn clauses
- In particular, this translation uses queries to represent loop invariants

Where are we?

- Next, we've looked at solving Horn clauses, that is, compute solution for the unknown queries
- The solutions to the queries give us the missing loop invariants
- We then looked at solving Horn clauses
- First, we start with a solution that maps all queries to $\perp$
- We then used the **post** operator to compute strongest postconditions
- For each clause, we compute the strongest postcondition of the body wrt. the head
- We can think of this process as computing the set of reachable states
- The strongest postcondition gives us a new state for the solution of the head query
- Unfortunately, if there is an unbounded number of states, the algorithm never terminates

Where are we?

- To solve this problem we've used abstraction
- Instead of the concrete **post**, we compute an abstract post **post#**
- **post#** expresses the post-condition over a finite vocabulary
- The finite vocabulary is given by a set of predicates **P** (e.g., $x \geq 0$, $x \geq 1$)
- **post#** computes the strongest post-condition that can be expressed as a

  conjunction of predicates in **P**
- Restricting the vocabulary ensures that the algorithm always terminates
- It may however be that the computed solution is not strong enough to

  prove our desired post-condition

Where are we?

- What have we gained?
- Instead of invariants, we now have to supply the building blocks: predicates
- We can just throw a bunch of predicates at it and “see what sticks”
- It doesn’t matter if we supply wrong predicates, as these will just not be used
- However, two many predicates and a large program may make the algorithm slow
- Our algorithm will also guess the Boolean structure of the invariant
- That is, we don’t have to worry about where to put ||

Where are we?

- Have we solved the problem of computing loop invariants automatically?
- No! If we’re missing the right predicates, the verification will still not work
- Not always easy to guess, that is, requires some ingenuity
- Works really well for: large programs, where the invariants have predictable structure

- Does it make verification easier?
- Don’t forget, there can be no perfect solution as we’re tackling an undecidable problem.

Where are we?

- After all, we’re proving that for *any* possible inputs the program is correct
- These are very strong guarantees, since, as Dijkstra said:

“Program testing can be used to
show the presence of bugs, but never to show their absence!”

- With verification, we can show their absence, but it’s now always easy

Verification for Security

- Now: How can we use what we've learned for proving security properties?
- First, often proving functional properties alone is good for security!
- Example: verified crypto library https://dl.acm.org/doi/pdf/10.1145/3133956.3134043
- Next, we can use reasoning about arrays to prove memory safety
- Memory errors are still among the most exploited and problematic security concerns!
- However, systems code often contains complicated pointer operations and difficult data-structures

Verification for Security

- One direction: pick abstractions that are tailored to memory safety
- Example: Facebook's Infer https://fbinfer.com/
- Try to detect Null-pointer exceptions in mobile applications
- Crashes in mobile apps are bad as no one ever updates, so old versions stick around
- Had some success using this in production
- Galois: https://galois.com/ Company that uses formal verification for security & more
- Amazon/AWS has a huge formal verification group
- https://www.youtube.com/watch?v=g-DH_b5bFd4
- http://www0.cs.ucl.ac.uk/staff/b.cook/oneclick.pdf
- Microsoft has the Dafny verifier: https://github.com/dafny-lang/dafny (VCGen)
- SLAM: https://www.microsoft.com/en-us/research/project/slam/ (Predicate Abstraction)

Applications

- Alternatives for memory safety: Rust (that is, safe subset ;)), Sanitiziers
- Current project: a verified sanitizer that provably detects memory errors
- Can be used to correctly patch existing vulnerabilities

- Some more applications:
    - Smart contracts
    - Some devastating hacks!
    - Lots of money on the line

Applications: Smart Contracts

- Smart contracts are (usually simple, imperative) programs that manage how assets are distributed
- Need to make sure that they don't contain bugs
- Lots of work, and a bunch of start ups:
  - https://veridise.com/
  - https://www.certora.com/
  - https://files.sri.inf.ethz.ch/website/papers/sp20-verx.pdf

Other Applications

- Proving Security of Crypto Algorithms:
    - https://github.com/EasyCrypt/easycrypt
- Verified enclave
    - https://www.microsoft.com/en-us/research/project/komodo/
- Operating Systems:
    - https://sel4.systems/
- Compiler
    - https://compcert.org/compcert-C.html
- Distributed Systems:
    - https://www.microsoft.com/en-us/research/publication/ironfleet-proving-practical-distributed-systems-correct/
    - https://dl.acm.org/doi/10.1145/3290372

My Research on Verification

- Verify that a distributed system is correct
- Pretend Synchrony: https://dl.acm.org/doi/10.1145/3290372
    - Implement a verified key-value store using Paxos
    - There are other verified implementations, but they require a lot of manual proof
    - Pretend Synchrony exploits insights about algorithms
    - For the purpose of the proof, we can pretend that the algorithm is synchronous
    - Our proofs require a lot less work than previous ones
    - Builds on VCGen, but need some tricks to reason about concurrency

My Research on Verification

- Iodine: https://gleissen.github.io/papers/iodine.pdf
  - Verify that hardware doesn't accidentally leak sensitive information (via timing)
  - Builds on predicate abstraction / Horn clauses
  - But now: the programs we verify are the hardware (written in Verilog!)
  - We can use the same technique by translating into Horn clauses
- Xenon: https://gleissen.github.io/papers/xenon.pdf
  - Follow-up that makes the method easier to use and verifies some real RISCV processors
  - More on this problem now!

Some more Light & Inspirational Reading:

- https://www.quantamagazine.org/formal-verification-creates-hacker-proof-code-20160920/
- https://www.quantamagazine.org/computing-expert-says-programmers-need-more-math-20220517/

Now: Information Flow

- Let’s now look at a different set of security problems: side-channels & speculation
- Can be used to extract information from (otherwise safe) cryptographic functions
- Real problem in practice, in particular for cloud providers like Amazon/Azure etc.
- Allow for sneaky attacks that are hard to defend against!

- But let’s start with a simpler problem
- Say, our program contains variables that are **secret**
- Moreover, there is an attacker, that can **observe** certain other variables
- We want to make sure that the secrets can’t fall into the hands of the attacker

Information Flow

<u>Example:</u>

- Consider the following program
- Variable **secret** contains our confidential value
- The attacker can observe variable **obs**

```
x := secret;
y := x;
obs := y
```

Quiz:

- Is this program safe?

Information Flow

Example:

- Consider the following program
- Variable **secret** contains our confidential value
- The attacker can observe variable **obs**

```
x := secret;
y := x;
obs := y
```

Quiz:

- Is this program safe?
- No: There is a direct flow from **secret** to **obs** (via variables x, y)

Information Flow

<u>Example:</u>

- Let's look at another program
- Again, **secret** contains our confidential value

```
x := secret;
obs := 0;
if (x=1) {
  obs := 1;
}
```

Quiz:

- Does this program contain a direct flow?
- Is it safe?
- What can happen?

Information Flow

Example:

- Let's look at another program
- Again, **secret** contains our confidential value

```
x := secret;
obs := 0;
if (x=1) {
  obs := 1;
}
```

- The program contains an indirect flow
- Depending on whether we take the branch or not, **obs** will have value 0 or 1.
- This leaks information about **obs**: the attacker can find out if **secret** has value 1

Non-Interference

- When is a program safe, according to this notion?
- We can formalize this in the notion of <u>non-interference</u>
- Intuition: let's look back at our initial unsafe example
- Let's look at <u>two executions</u> of the program (left and right)
- In these two executions the **secrets** take different values
- All other values are the same
- The program is <u>safe</u>, if the observables have the same value
- That is, despite the **secrets** being different, an attacker cannot distinguish the two runs

x := **secret**;	x := **secret**;
y := x;	y := x;
obs := y	**obs** := y

Non-Interference

Example:

```
x := 3;          x := 5;
y := x;          y := x;
obs := y         obs := y
```

- Say we set the secret to **3** in the left run and **5** in the right run
- After executing the program **obs** is **3** in the left run and **5** in the right run
- Hence, an attacker can distinguish the two runs, and the program is not safe!

Non-Interference

Example:

```
x := secret;
obs := 0;
if (x=1) {
  obs := 1;
}
```

```
x := secret;
obs := 0;
if (x=1) {
  obs := 1;
}
```

Quiz:

- Let’s look at our second example
- Is the example safe according to our new notion of security?

Non-Interference

<u>Example:</u>

```
x := 1;
obs := 0;
if (x=1) {
  obs := 1;
}
```

```
x := 5;
obs := 0;
if (x=1) {
  obs := 1;
}
```

- No!
- If we pick secret 1 in the left run and 5 in the right run, the observations will differ
- In the left run, the attacker observes 1, in the right run 0
- This is called an <u>indirect flow</u>
- Note, we have to pick specific values for the difference to occur
- For the program to satisfy non-interference attacker observations need to be the same **for any** choice of secrets!

Non-Interference

Quiz:

- Remember our semantics relation $\langle s, \sigma \rangle \Downarrow \sigma'$, saying state σ changes to state σ' after executing statement s
- Let's write $\sigma \downarrow$**obs** for restriction of σ's domain to attacker observable varibles
- How can we formally define non-interference?

Non-Interference

Quiz:

- Remember our semantics relation $\langle s, \sigma \rangle \Downarrow \sigma'$, saying state σ changes to state σ' after executing statement s
- Let's write $\sigma\downarrow$**obs** for restriction of σ's domain to attacker observable varibles
- How can we formally define non-interference?

For all σ_L, σ_R: if $\sigma_L\downarrow$**obs**$=\sigma_R\downarrow$**obs** and $\langle s, \sigma_L\rangle \Downarrow \sigma_L'$ and $\langle s, \sigma_R\rangle \Downarrow \sigma_R'$ then $\sigma_L' \downarrow$**obs** $=$ $\sigma_R'\downarrow$**obs**

Non-Interference

Quiz:

```
obs := obs +1;
```

- Does the program above satisfy non-interference (as we defined it)?

Non-Interference

Quiz:

```
x := secret;
while (x≥1) {
   x:= x+1;
}
```

- Does the program above satisfy non-interference (as we defined it)?
- Could an attacker still distinguish two runs with different secrets?

Non-Interference

Quiz:

```
x := secret;
while (x≥1) {
   x:= x+1;
}
```

- Does the program above satisfy non-interference (as we defined it)?
- Could an attacker still distinguish two runs with different secrets?

- Yes! If an attacker can see whether the program terminates, they can distinguish the runs
- We defined termination insensitive non-interference
- One can define termination sensitive non-interference, which in addition requires the program to terminate

Self-composition

Quiz:

```
x := secret;        x := secret;
y := x;             y := x;
obs := y            obs := y
```

- Let's look at our example program
- Can we use our previous verification techniques to verify non-interference?

Self-composition

Quiz:

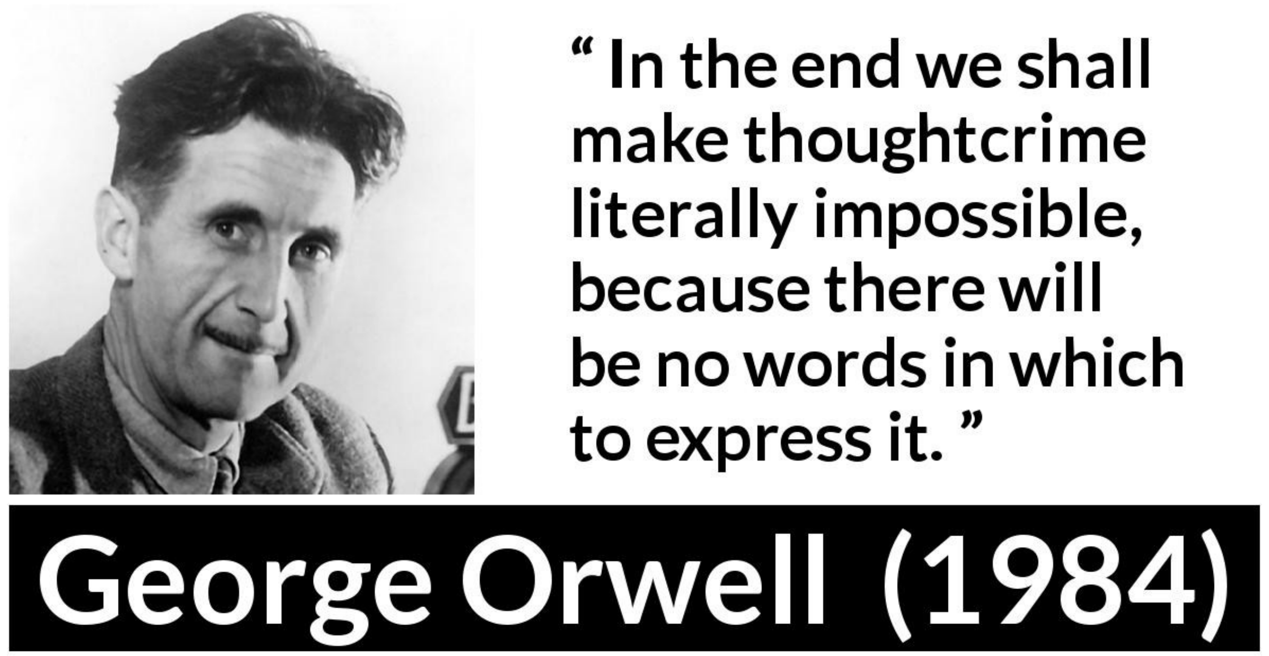
```
x := secret;        x := secret;
y := x;             y := x;
obs := y            obs := y
```

- Let's look at our example program
- Can we use our previous verification techniques to verify non-interference?
- Not directly! Non-interference talks about two runs
- This is called a hyper-property
- But, we can form a new program made up of two copies
- We can then use verification to prove non-interference
- This is called self-composition or product program

https://www-sop.inria.fr/lemme/Tamara.Rezk/publication/Barthe-DArgenio-Rezk-Journal.pdf

Information Flow Types

- We'll now take a look at an approach with a different philosophy
- Up to now, you can write *any* program and we'll try to prove correctness
- This is the verification approach (conferences like CAV)
- There's a different approach that comes from the PL community (PLDI, POPL)
- You're not allowed to write programs that might be wrong
- Idea: you cannot even express possibly incorrect programs
- Comes at the risk of ruling out programs that are correct but less obviously so
- Totalitarian programing: wrong programs are thought crime
- Another example: Rust
- For security, this is called language-based security



Information Flow Types

- We'll now define a type system such that if the program type checks, then it is non-interferent

- We'll do this for Nano:

Expressions: $e, e_1, e_2 \ni Exp ::= n \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2$ where $n \in \mathbb{Z}$, $x \in Vars$

Boolean Expressions: $b, b_1, b_2 \ni BExp ::= \top \mid \bot \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid e_1 = e_2 \mid e_1 \leq e_2$

Statement:

$s, s_1, s_2 \ni Stmt ::=$ skip	(no-op)
$\mid x := e$	(assignment)
$\mid s_1 ; s_2$	(sequential composition)
$\mid$ if b then s_1 else s_2	(if)
$\mid$ while b do s	(while)

Information Flow Types

- First, we'll note that our two levels of security **sec** and **obs** form a <u>lattice</u> $\mathcal{L}$
- That means we can define a relation $\sqsubseteq$ such that a $\sqsubseteq$ b, iff b <u>is at least as confidential as</u> a.
- You can think of b as requiring higher (or equal) "security clearance"

Quiz:

- How should $\sqsubseteq$ be defined for **sec** and **obs** ?

- We also define a join operator $\sqcup$ such that for all $a,b \in \mathcal{L}$: $a \sqsubseteq a \sqcup b$ and $b \sqsubseteq a \sqcup b$

Quiz:

- How should $\sqcup$ be defined for **sec** and **obs** ?

Information Flow Types

- Let's assume we have a function Γ that assigns a security label to each variable

Quiz:

```
x := secret;
y := x;
obs := 3
```

- Let's look again at the following example:
- What should we assign to Γ(**secret**)?
- What about Γ(**obs**)?
- What about Γ(x)?
- What about Γ(y)?
- Is the program safe?

Information Flow Types

- Let's assume we have a function Γ that assigns a security label to each variable

Quiz:

```
x := secret;
y := x;
obs := y
```

- What if, we change it to match our previous example
- What's the type of $\Gamma(y)$ now?
- Is the program safe?

- Now: formalize this in type system

Information Flow Types

- Let's define a typing judgement $\Gamma \vdash e : \ell$ saying that expression e has label $\ell \in \mathscr{L}$
- We define them using inference rules, one per expression

$$\underline{\text{var}} \quad \frac{\Gamma(x) = \ell}{\Gamma \vdash x : \ell} \qquad \underline{\text{num}} \quad \frac{}{\Gamma \vdash n : \textbf{obs}} \qquad \underline{\text{plus}} \quad \frac{\Gamma \vdash e_1 : \ell_1 \quad \Gamma \vdash e_2 : \ell_2}{\Gamma \vdash e_1 + e_2 : \ell_1 \sqcup \ell_2}$$

- Similarly, for binary a expression b, we get

$$\underline{\top} \quad \frac{}{\Gamma \vdash \top : \textbf{obs}} \qquad \underline{\text{or}} \quad \frac{\Gamma \vdash b_1 : \ell_1 \quad \Gamma \vdash b_2 : \ell_2}{\Gamma \vdash b_1 \vee b_2 : \ell_1 \sqcup \ell_2} \qquad \underline{\text{eq}} \quad \frac{\Gamma \vdash e_1 : \ell_1 \quad \Gamma \vdash e_2 : \ell_2}{\Gamma \vdash e_1 = e_2 : \ell_1 \sqcup \ell_2}$$

Information Flow Types

Quiz:

- Let's say $\Gamma(x) = \mathbf{obs}$ and $\Gamma(y) = \mathbf{sec}$

- Can we derive that $\Gamma \vdash x : \mathbf{obs}$?
- Can we derive that $\Gamma \vdash x : \mathbf{sec}$?
- What label ℓ do we get for $\Gamma \vdash 3 : \boldsymbol{\ell}$?
- What label ℓ do we get for $\Gamma \vdash x + 3 : \boldsymbol{\ell}$?
- What label ℓ do we get for $\Gamma \vdash y + 3 : \boldsymbol{\ell}$?
- What label ℓ do we get for $\Gamma \vdash x + y : \boldsymbol{\ell}$?

Information Flow Types

- All for today
- Next lecture: typing statements
- First a wrong attempt, then the correct one
- Then, how to use type systems to rule out side-channel (and speculation) attacks

Reading

- Lattices: https://ieeexplore.ieee.org/document/246638
- IFC Type systems: https://www.cs.cornell.edu/andru/papers/jsac/sm-jsac03.pdf
- Lecture notes: https://www.cs.uoregon.edu/research/summerschool/summer19/lecture_notes/myers1.pdf

- There are also videos available for these lectures:
- https://www.cs.uoregon.edu/research/summerschool/summer12/curriculum.html
- Under: Language-based security