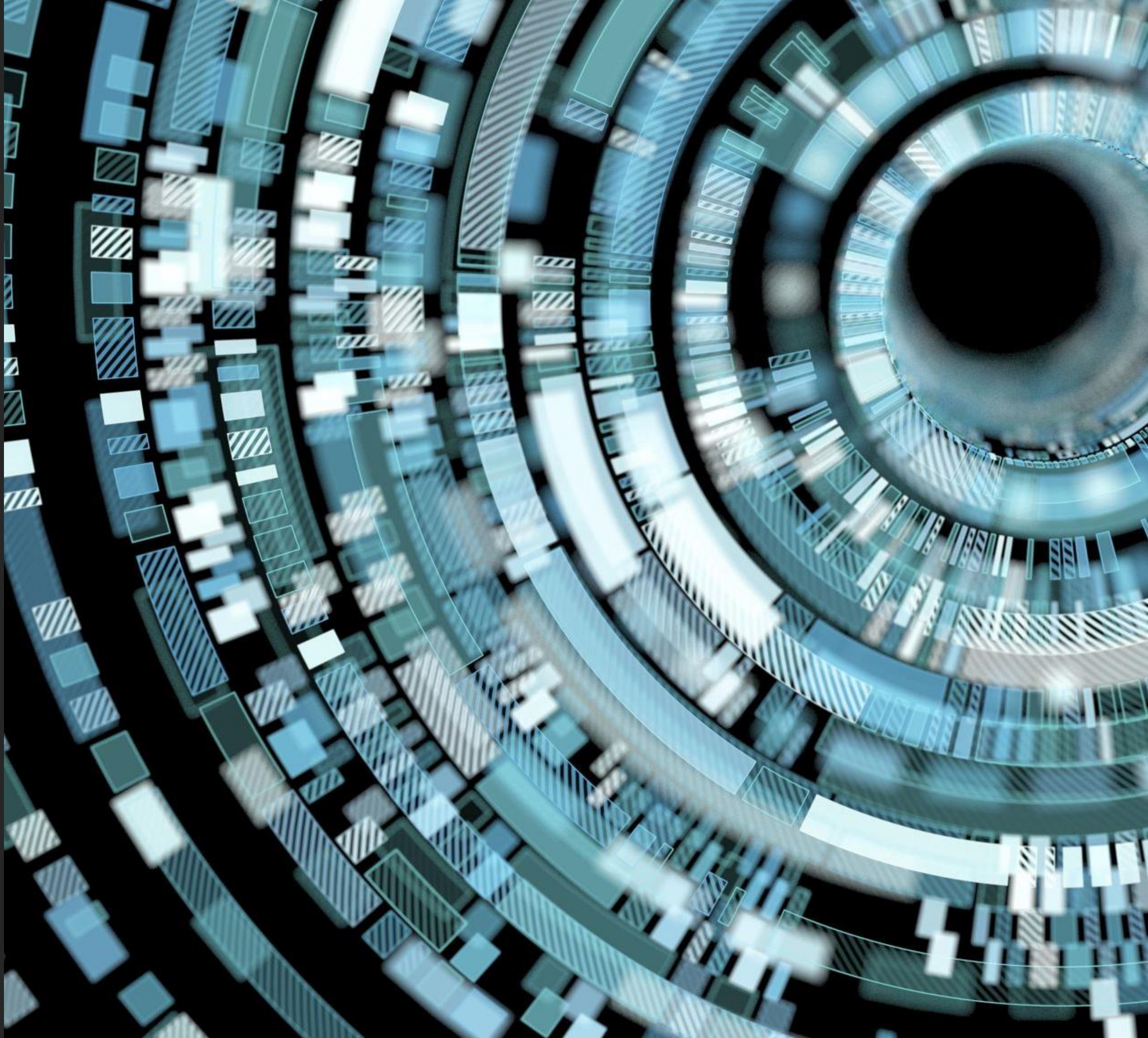
Checking Security of Java Bytecode by Abstract Interpretation

BARBUTI, BERNARDESCHI, DE FRANCESCO

LORENZO BELLOMO



Outline of the seminar

1. Introduction and recap
2. Language syntax and semantics
3. Introducing security
4. The new concrete semantics
5. Abstract interpretation and abstract semantics
6. Future (and past) works
7. Conclusions

Information Leakage

It is essential to be able to download code from Internet (i.e Applications…)

Downloaded code is *dangerous* by nature

There are two options to reduce this problem:

- Make the application unable to access local files (not interesting)
- Allow access, but deny the possibility of data *leakage*

Solving this problem is *hard*

Recap: Information Flow

Secure information flow means that *high level* information does NOT flow towards *lower levels*

Two types of information flow:

- *Explicit flow*: Performed through an assignment
- *Implicit flow*: Nested inside conditional branches (and possibly following jumps in the code)

Information flow sources

So if: `security_lvl(x)<security_lvl(y)`

Explicit flow is: `x = y;`

Implicit flow is:

```
if(y == 5) then x = 0 else x = 1;
```

But leakage is also possible through *behaviour analysis.*

Think about `while(y == 0) do skip;`

And in this case we are dealing with bytecode, so we should also take care of leakages to the *operand stack* and *program counter*!

The JVML0 language (1)

- op: pops 2 elements from the operand stack, performs the operation and pushes the result back (simple arithmetic)
- pop: discards the top of the stack
- push k: pushes the constant k on top of the stack
- load x: pushes the content of variable x on top of the stack
- store x: pop from the stack and put this value in variable x
- continues…

The JVML0 language (2)

- if j: pop from the stack, and if not 0 jump to j
- goto j: jump to address j
- jsr j: at address p, jump to address j and push return address p + l onto the operand stack
- ret x: jump to address stored in x
- halt: stop

Some notations

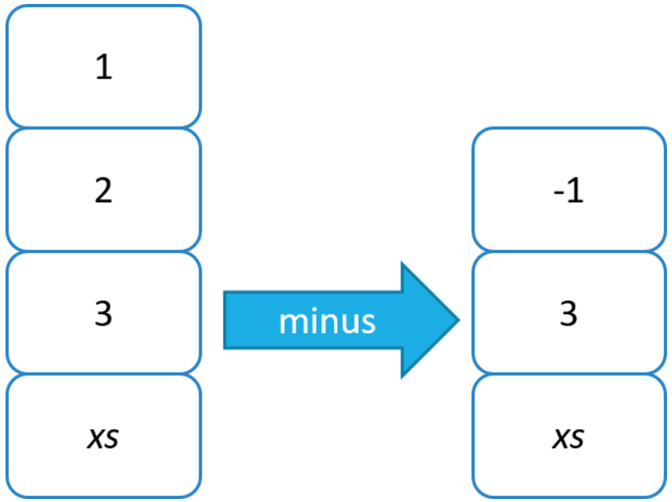
The stack is denoted as a sequence *w*, where w[i] is the i-th element and w[1] is the top

The program execution is a triple $\langle i, m, s \rangle$, where:

- *i:* it is the position of the program counter
- *m:* it is the memory for the variables
- *s:* it is the operand stack

For example, the stack in figure is written like: $1 \cdot 2 \cdot 3 \cdot xs$

If I apply the minus operator then this stack transforms



$$\text{op}\ \frac{c[i] = op}{\langle i, m, k_1 \cdot k_2 \cdot s \rangle \longrightarrow^e \langle i+1, m, (k_1\ op\ k_2) \cdot s \rangle}$$

$$\text{pop}\ \frac{c[i] = pop}{\langle i, m, k \cdot s \rangle \longrightarrow^e \langle i+1, m, s \rangle}$$

$$\text{push}\ \frac{c[i] = push\ k}{\langle i, m, s \rangle \longrightarrow^e \langle i+1, m, k \cdot s \rangle}$$

$$\text{load}\ \frac{c[i] = load\ x \qquad m(x) = k}{\langle i, m, s \rangle \longrightarrow^e \langle i+1, m, k \cdot s \rangle}$$

$$\text{store}\ \frac{c[i] = store\ x}{\langle i, m, k \cdot s \rangle \longrightarrow^e \langle i+1, m[k/x], s \rangle}$$

$$if_{false}\ \frac{c[i] = if\ j}{\langle i, m, 0 \cdot s \rangle \longrightarrow^e \langle i+1, m, s \rangle}$$

$$if_{true}\ \frac{c[i] = if\ j}{\langle i, m, k \neq 0 \cdot s \rangle \longrightarrow^e \langle j, m, s \rangle}$$

$$\text{goto}\ \frac{c[i] = goto\ j}{\langle i, m, s \rangle \longrightarrow^e \langle j, m, s \rangle}$$

$$\text{jsr}\ \frac{c[i] = jsr\ j}{\langle i, m, s \rangle \longrightarrow^e \langle j, m, (i+1) \cdot s \rangle}$$

$$\text{ret}\ \frac{c[i] = ret\ x}{\langle i, m, s \rangle \longrightarrow^e \langle m(x), m, s \rangle}$$

The standard semantics

Control Flow Graph

Control flow graph G=(V,E)

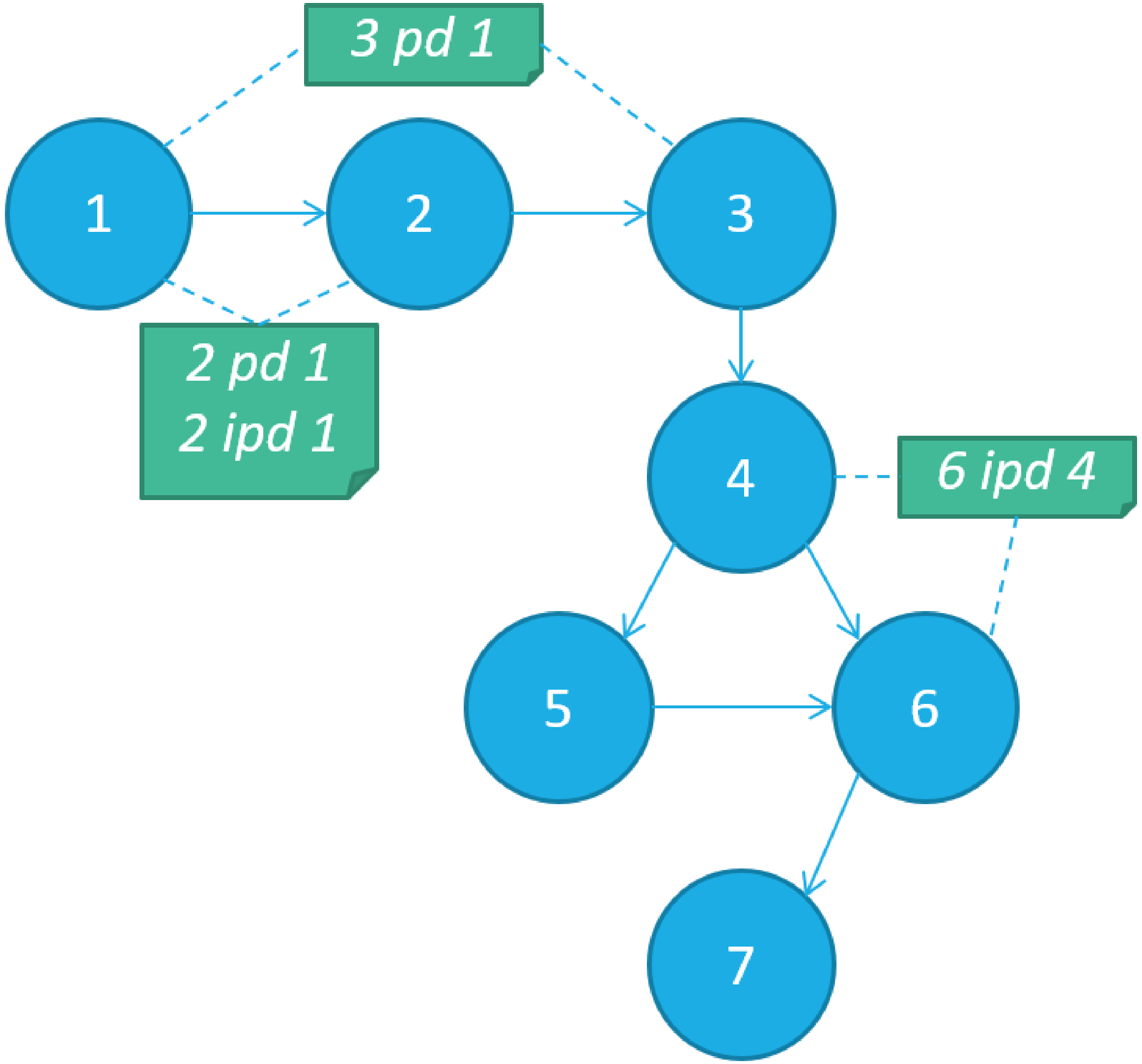
- V is the *instruction set*
- E is the set of edges, where edge *(x,y)* exists iff. the instruction at address y *can be executed after* the one at address x

Given and edge (x,y), we say that *y post dominates x* if every path starting from x eventually gets to y (y pd x).

y is said to *immediately postdominate* x (*y ipd x*) iff *y pd x* and there is no node r such that *y pd r pd x*

One example

```
1. push 1
2. push 2
3. load y
4. if 6
5. pop
6. store x
7. halt
```



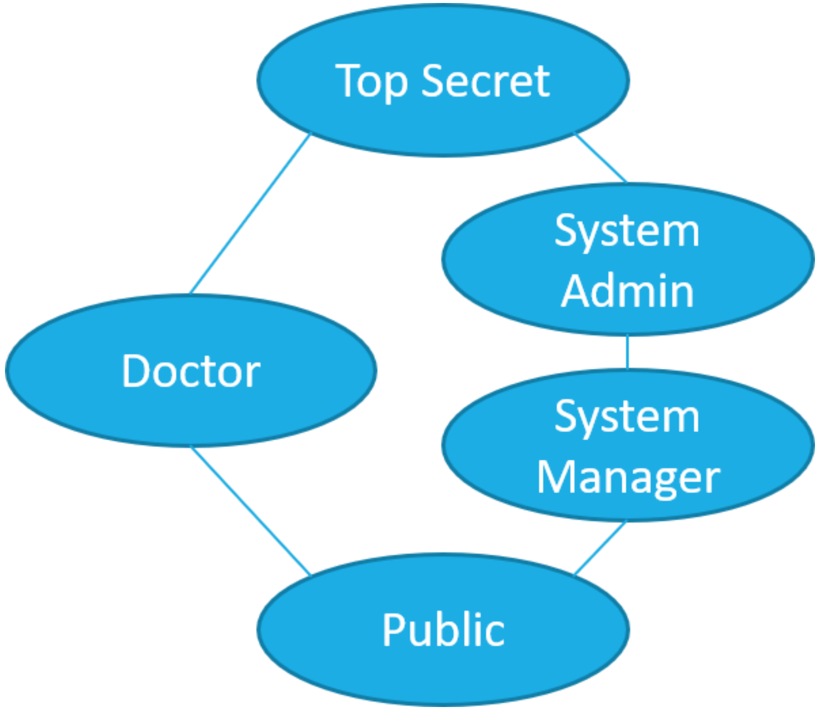
Why the graph?

Every time there is a branching instruction (i.e. `ret` or `if`) an *implicit flow* starts

The implicit flow affects every instruction from *i to ipd(i)*

The graph can be *statically analyzed* to identify implicit flows

We add a fake n+1 node so that any i has at least one j such that *j=ipd(i)*



Introducing security

We are going to need to express *security levels, partially ordered*

Given two security levels σ, τ, if σ has a *lower security level* with respect to τ then: $\sigma \sqsubset \tau$

Their *least upper bound* is $\sigma \sqcup \tau$

In the example:

- $P \sqsubset D,\ SM \sqsubset SA,\ SM \sqsubset TS$ …
- $TS = SA \sqcup TS,\ P = P \sqcup P,\ TS = D \sqcup SM$ …

σ-Security

Every variable has its own security level σ

With respect to σ, the remaining variables can be split in two groups:

- $\Lambda_{\sqsubseteq\sigma}$: The subset of variables with security level $\tau \sqsubseteq \sigma$
- $\Lambda_{\not\sqsubseteq\sigma}$: The subset of variables with security level $\tau \not\sqsubseteq \sigma$ (both $\sqsupset$ and unrelated)

σ*-Security* is parametric, and guarantees that the information in $\Lambda_{\not\sqsubseteq\sigma}$ is kept secret

But it makes no assumptions on $\Lambda_{\sqsubseteq\sigma}$

σ-Security limit

Which means that in $\Lambda_{\sqsubseteq\sigma}$ anything could happen

A variable $x \in \Lambda_{\sqsubseteq\sigma}$ with security level higher then $y \in \Lambda_{\sqsubseteq\sigma}$ might depend from it

In our example before, if we consider *SystemAdmin* as σ, then any dependency is allowed between *Public* and *System Manager*

It is still considered σ*-Secure*

Not σ-Secure programs

```
1. load y
2. if 5
3. push 1
4. goto 6
5. push 0
6. store x
7. halt
```

```
1. load y
2. if 1
3. halt
```

```
1. load y
2. if 5
3. push 1
4. goto 6
5. push 0
6. halt
```

```
1. load y
2. if 4
3. halt
4. halt
```

Assuming:
$x \in \Lambda_{\sqsubseteq\sigma}, y \in \Lambda_{\sqsubseteq\tau}$
$\sigma \sqsubset \tau$

Stack depends on y

Towards Concrete Semantics

Extension of the previously seen *semantics*

It is executed under a *Security Environment* (the security level)

It gets *upgraded* when there is a branching (security level grows)

It gets *downgraded* whenever the branching ends (security level falls back)

Branching, security levels

Anytime there is a branching, the semantics stores a couple $(\sigma, ipd(i))$

Anytime a branch merges, the couple is *discarded*

Nested branchings are handled in a *LIFO* policy

So we need to add another component to our semantics, in order to handle *ipd couples*

Concrete Semantics Building Blocks

We modeled the program execution as a triple

Now we need a *quadruple* $\langle i, M, S, \rho \rangle$, where:

- *i:* it is the position of the *program counter*
- *M:* it is the *memory*, and records associations from names to σ-*values* (couples (v, σ) of *values* and *security levels*)
- *S:* it is the *operand stack*, made of σ-*values*
- ρ: The newly added *ipd stack* (the LIFO queue for *branchings*)

$$\text{ipd}\ \frac{\rho = (i,\tau)\cdot\rho\prime}{\sigma \models \langle i, M, S, \rho\rangle \longrightarrow \tau \models \langle i, M, S, \rho\prime\rangle}$$

New Rule with max priority: it ensures security levels stay consistent

$$\text{op}\ \frac{c[i] = op \qquad i\ not\_in\ \rho}{\sigma \models \langle i, M, (k_1,\tau_1)\cdot(k_2,\tau_2)\cdot S, \rho\rangle \longrightarrow \sigma \models \langle i+1, M, (k_1\ op\ k_2, \tau_1 \sqcup \tau_2)\cdot S, \rho\rangle}$$

$$\text{pop}\ \frac{c[i] = pop \qquad i\ not\_in\ \rho}{\sigma \models \langle i, M, (k,\tau)\cdot S, \rho\rangle \longrightarrow \sigma \models \langle i+1, M, S, \rho\rangle}$$

Security Environment

$i \notin \rho$ is to give priority to ipd rule, which guarantees security correctness

Concrete Semantics (1)

$$\text{push}\ \frac{c[i] = push\ k \qquad i\ not\_in\ \rho}{\sigma \models \langle i, M, S, \rho \rangle \longrightarrow \sigma \models \langle i+1, M, (k, \sigma) \cdot S, \rho \rangle}$$

PUSH gives the security level of the environment

$$\text{load}\ \frac{c[i] = load\ x \qquad M(x) = (k, \tau) \qquad i\ not\_in\ \rho}{\sigma \models \langle i, M, S, \rho \rangle \longrightarrow \sigma \models \langle i+1, M, (k, \tau \sqcup \sigma) \cdot S, \rho \rangle}$$

LOAD needs to account for the memory too

$$\text{store}\ \frac{c[i] = store\ x \qquad i\ not\_in\ \rho}{\sigma \models \langle i, M, (k, \tau) \cdot S, \rho \rangle \longrightarrow \sigma \models \langle i+1, M[(k, \tau)/x], S, \rho \rangle}$$

STORE binds the memory

Concrete Semantics (2)

Operator $\odot$ only updates ρ if address i is NOT already on top of the stack

$$if_{false}\ \frac{c[i] = if\ j \qquad i\ not\_in\ \rho}{\sigma \models \langle i, M, (0,\tau)\cdot S, \rho\rangle \longrightarrow \tau \models \langle i+1, upgrade_M(M,i,\tau), upgrade_S(S,\tau), (ipd(i),\sigma)\odot\rho\rangle}$$

$$if_{true}\ \frac{c[i] = if\ j \qquad i\ not\_in\ \rho}{\sigma \models \langle i, M, (k\neq 0,\tau)\cdot S, \rho\rangle \longrightarrow \tau \models \langle j, upgrade_M(M,i,\tau), upgrade_S(S,\tau), (ipd(i),\sigma)\odot\rho\rangle}$$

$upgrade_M(M,i,\tau)(x) = (k, \sigma \sqcup \tau)$
$upgrade_S(S,\tau)(x) = (k, \sigma \sqcup \tau)$
Only applied to variables in the implicit flow path

Environment is upgraded from σ to τ

Concrete Semantics (3)

$$\text{goto}\ \frac{c[i] = goto\ j \qquad i\ not\_in\ \rho}{\sigma \models \langle i, M, S, \rho\rangle \longrightarrow \sigma \models \langle j, M, S, \rho\rangle}$$

$$\text{jsr}\ \frac{c[i] = jsr\ x \qquad i\ not\_in\ \rho}{\sigma \models \langle i, M, S, \rho\rangle \longrightarrow \sigma \models \langle j, M, (i+1, \sigma)\cdot S, \rho\rangle}$$

JSR records on the stack that the security level is σ

$$\text{ret}\ \frac{c[i] = ret\ x \qquad M(x) = (j, \tau) \qquad i\ not\_in\ \rho}{\sigma \models \langle i, M, S, \rho\rangle \longrightarrow \sigma \sqcup \tau \models \langle j, upgrade_M(M, i, \sigma \sqcup \tau), upgrade_S(S, \sigma \sqcup \tau), (ipd(i), \sigma) \odot \rho\rangle}$$

Concrete Semantics (4)

Limitation of the concrete semantics

Problem with the provided *concrete semantics*: it depends on the *runtime values*

We would like to make the security analysis *values oblivious*

Need to *approximate* the behaviour, because *σ-Security* is based on *all* the executions

We need to *abstact* the process in order to obtain automation

Reminder, abstract interpretation

Approximate the concrete semantics of a system, by providing an *abstract semantics*

It is useful to obtain a broader view, when in need to *abstract* a process

It is based on the *behaviour* of the process to be analyzed

Concept of *approximation*

$$\boxed{+3,\ +5,\ +16} \xrightarrow{\alpha} \boxed{+} \xrightarrow{\gamma} \boxed{+\ \boxed{+3,\ +5,\ +16}}$$

Abstract Semantics

New abstract semantics built like the concrete one, but we *discard the values*

Given the old memory M formed by couples (v, σ), we now get $M^\natural$, formed by only security levels σ

Similarly, S becomes $S^\natural$, formed by σ instead of couples (v, σ).

The rules are all *identical* (but on new domains) with respect to the concrete ones

Only `if` and `ret` are *redefined*

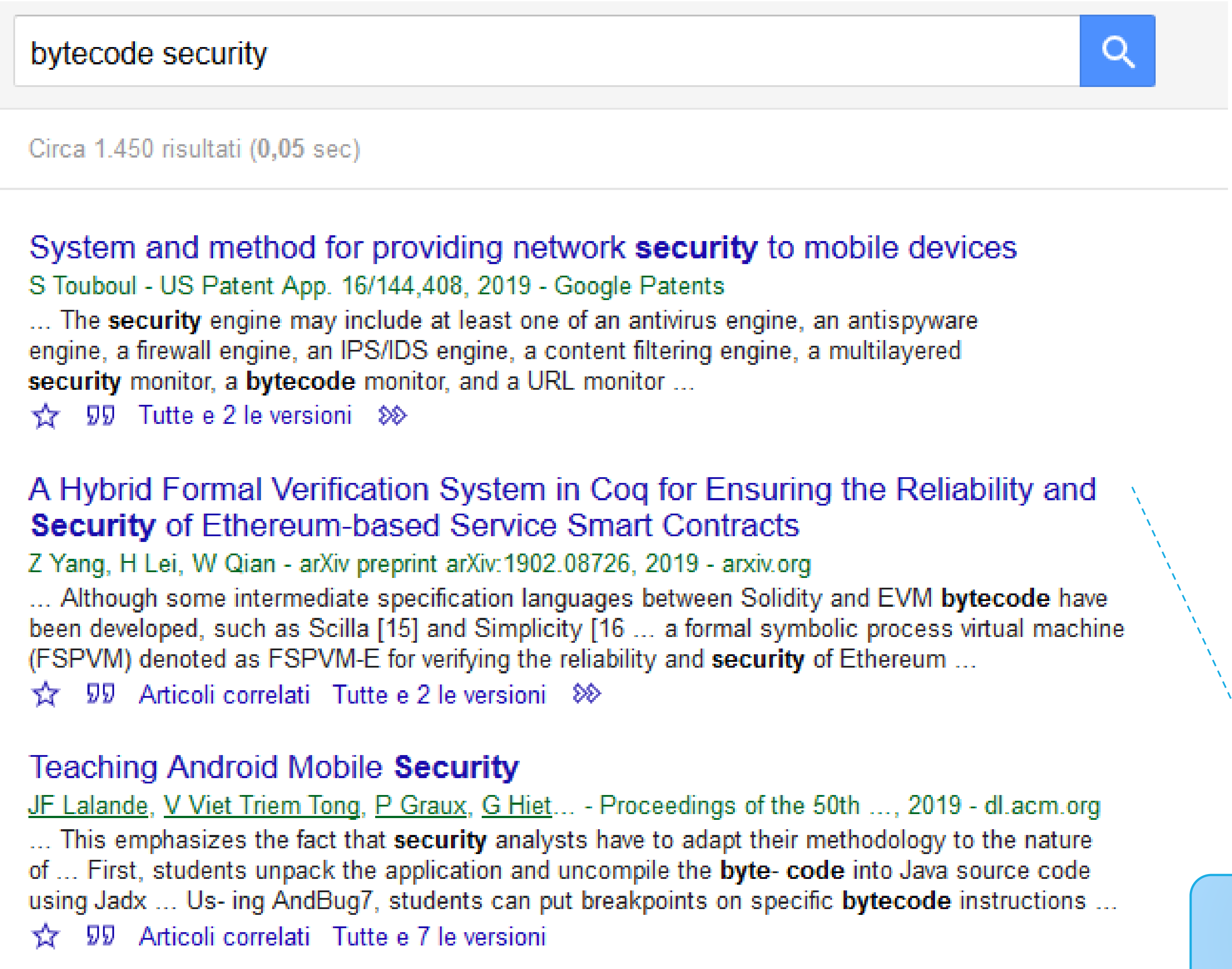
`if` checks both branches, `ret` explores all of them

G=(V,E)
I consider every edge in the Control Flow Graph

$$if^\natural \frac{c[i] = if\ x \qquad (i,j') \in E \qquad i\ not\_in\ \rho}{\sigma \models \langle i, M^\natural, \tau \cdot S^\natural, \rho \rangle \longrightarrow^\natural \tau \models \langle j', upgrade^\natural_M(M^\natural, i, \tau), upgrade^\natural_S(S^\natural, \tau), (ipd(i), \sigma) \odot \rho \rangle}$$

$$ret^\natural \frac{c[i] = ret\ x \qquad M^\natural(x) = \tau \qquad (i,j') \in E \qquad i\ not\_in\ \rho}{\sigma \models \langle i, M^\natural, S^\natural, \rho \rangle \longrightarrow^\natural \sigma \sqcup \tau \models \langle j', upgrade^\natural_M(M^\natural, i, \sigma \sqcup \tau), upgrade^\natural_S(S^\natural, \sigma \sqcup \tau), (ipd(i), \sigma) \odot \rho \rangle}$$

Abstract Semantics, if and ret


bytecode security

Circa 1.450 risultati (0,05 sec)

System and method for providing network **security** to mobile devices
S Touboul - US Patent App. 16/144,408, 2019 - Google Patents
... The **security** engine may include at least one of an antivirus engine, an antispyware engine, a firewall engine, an IPS/IDS engine, a content filtering engine, a multilayered **security** monitor, a **bytecode** monitor, and a URL monitor ...
Tutte e 2 le versioni

A Hybrid Formal Verification System in Coq for Ensuring the Reliability and **Security** of Ethereum-based Service Smart Contracts
Z Yang, H Lei, W Qian - arXiv preprint arXiv:1902.08726, 2019 - arxiv.org
... Although some intermediate specification languages between Solidity and EVM **bytecode** have been developed, such as Scilla [15] and Simplicity [16 ... a formal symbolic process virtual machine (FSPVM) denoted as FSPVM-E for verifying the reliability and **security** of Ethereum ...
Articoli correlati Tutte e 2 le versioni

Teaching Android Mobile **Security**
JF Lalande, V Viet Triem Tong, P Graux, G Hiet... - Proceedings of the 50th ..., 2019 - dl.acm.org
... This emphasizes the fact that **security** analysts have to adapt their methodology to the nature of ... First, students unpack the application and uncompile the **byte- code** into Java source code using Jadx ... Us- ing AndBug7, students can put breakpoints on specific **bytecode** instructions ...
Articoli correlati Tutte e 7 le versioni

Future works

Consider leaks from exceptions

Extend this method to the whole Java bytecode

Combine abstract interpretation and model checking (2002)

2019

2002

Conclusions

We were able to:

1. *Identify* an interesting subset of Java bytecode
2. Provide a *semantics* for given code
3. Provide a *security model*
4. Provide a *concrete semantics* for the code w.r.t. the security model
5. Provide an *abstract semantics* using *abstract interpetation*

References

- "Compile-time detection of information flow in sequential programs" Banatre, Bryce, Métayer (1994)
- "Combining Abstract Interpretation and Model Checking for Analysing Security Properties of Java Bytecode" Bernardeschi, de Francesco (2002)
- "Java bytecode verification by model checking" Basin, Friedrich, Posegga, Vogt (1999)
- "A lattice model of secure information flow" Denning (1976)