LBT Notes

- the 3 main security problems are
  - (information) leaks
  - buffer errors
  - injection
- are based on vulnerabilities known for decades but which still cause damage
  - a programming language is considered safe when it provides linguistic abstractions (and toolkits) to avoid these vulnerabilities
- *practically none of the languages used to date can be defined as safe*
  - a secure programming language is not easy to build, there are several problems
    - interoperability with the rest of the sw
    - must run on virtual machines already made and with runtime supports already written
    - the compilation must generate a target code that preserves the security properties

- how programming languages are implemented
  - two main methods
    - compilation
      - translate the source code into a target code executable by a machine (hw or virtual) and execute it
        - possibility of doing very refined static analyzes and optimizations on the code
    - interpretation
      - execute the code as is
        - practically ne ssuna
    - mixed
      - first compilation phase and then interpreters (like java which first compiles into bytecode and then interprets that)

- **LANGUAGE BASED TECHNOLOGY FOR SECURITY**
  - science that exploits the theory of languages to force software security to
    - design and develop *new* languages already with safety in mind in the design phase design
    - and develop tools and robe to secure the code already written

            - example: changes to the runtime of a language

- there are 3 phases of static security policy
    - enforcement: find and fix vulnerabilities before running the code
    - : identifies and blocks attacks at runtime
    - audit: recover from damage and assign blame

- PROBLEM
    - is possible to develop a safe language (guaranteed vulnerability free?), that is:
        - it must never fail it
        - must manage adverse conditions
        - it must be reasonably efficient
            - approach classic:
                - test tape
                - 10k pages of formal demonstration
    - **that is, we can develop a "science of security"**
        - can we have certified safety demonstrations?
        - can we evaluate security and make comparisons between two different programs? (more or less sure of ...)
        - are there any security properties that are proven unenforceable?
        - can we demonstrate that certain enforcement mechanisms can guarantee classes of policies and not others?

- the security problem is relatively recent
    - computers have become so 'crucial and so' connected to each other in recent times it
        - was difficult to make an attack because it was not worth it and because it was really difficult to reach the machine
    - almost all programming languages today they are designed to be as fast as possible, not to be safe. In the clash between "functionality" and "security" the first always won,
        - *the time has come to consider the trade-off between efficiency (execution time) and security in the design phase,*
        - hoping that it is the individual programmer who takes care of writing code sure it proved to be unsuccessful (it is too difficult)
            - the language and the whole toolchain need to lend a hand and that the dev is left alone to think about the logic of the program

- **NB:** cryptography is only a piece (relatively small in a sense) to achieve security

- **security weakness / flaw:** what is wrong or that could be done better
  - if a flaw**is**:
    - *accessible*: the attacker must be able to "see" it
    - *exploitable*: it can be exploited to do damage
  - then becomes a **security vulnerability**
    - **flaw** exploitable by an attacker

- DESIGN FLAW VS IMPLEMENTATION FLAW
  - **design flaw**: inherent flaw or vulnerability of the system design (design error)
  - **implementation flaw**: error introduced by human error (programmer) or by toolchain (compiler, ...)
  - configuration flaw
  - malicious exploiting of a desired functionality (email exploiting -> spam)

- **Implementation Flaws**
  - flaws that can be found by looking at the
    - typos code, errors distraction, names mismatch, ...
    - errors in the program logic, errors in accessing data (out of bound ..), etc.
  - flaws introduced by the underlying platform or by other services (os, compilers, ...)
    - **example**:
      - java code that models a current account
        - class *Account*
          - private field *amount*
          - public method *deposit (int amount)*
        - running the program in JAVA guarantees us a certain level of security
          - *amount* cannot be altered directly but only through the method offered by the class
      - let's say this code is compiled "towards C" (target language C)
        - class *Account* is translated as record (struct)
          - field amount

            - pointer to function for the function *deposit (int n)*
        - **security problems now are obvious**
            - anyone has direct access to the *amount* since C has no visibility policies,
            - it is even possible to change the reference of the function pointer and when it is invoked *deposit* any code will actually run
    - there is a big discrepancy between the abstractions that the programming language provides (JAVA) and the compiled language (C)

- you have to consider all the attack levels
    - take the example of before *Account* in JAVA

```
private int amount
public void decrease (int n) {
     if (amount <= n)
          amount = amount - n;
     else
          printf ("no founds");
}
public void increase (int n) {
     amount = amount + n
}
```

    - there are 3 flaws
        - **implementation flaw:** wrong if guard
        - **design flaw:** never trust the user: if n were negative?
        - **underlying flaw**: and if *amount + n* were too large to be saved in an int type data (64 bit)

- As mentioned we would like to treat security as a science
    - we would like a framework that defines the security of a system in a precise and formal way, so to
        - evaluate the security of the specific system
        - make comparisons in terms of security between distinct systems
        - improve the design and implementation of new systems

- To try to develop a science of security we will use the **security model approach,** consisting of
  - **system model:** description of the system of interest
    - how the system behaves in favorable and unfavorable conditions (wrong inputs, attacks, ..)
  - **threat model**
    - definition of the computational power and the degree of access to the system that attackers have
  - **security policies**
    - definition of the security properties that we would always like to have respected
    - ability to determine if the properties are respected for each behavior of the system ema or not
      1. when a property is compromised? is there 'a way to make the system behave in such a way as to compromise that property'?
    - **the CIA Factor**
      1. **Confidentiality:** no sensitive information is revealed
      2. **Integrity:** attackers cannot destroy the system
      3. **Availability:** attackers cannot make the system inaccessible to (authorized) users

- Security analysis
  - security models (system model, threat model and security policies) provide the basis for **security analysis,** i.e. the system evaluation process
    - does the system model guarantee the security policies when it collides with the threat model?

- **EXAMPLE: N**eedham-**S**chroeder Public Key Protocol
  - classic man in the middle attack between secure communications in which Eve manages to pretend to be A in the eyes of B and vice versa
    - encryption is not the solution, but it is *only* a small block of cybersec : communications as such are secure but not enough

  - **NS Security model**
    - **system model**
      - allowed actions
        - establishing communication

                - A chooses a random number N_a and sends the pair <A, N_a> encrypted with B's public key
                - B chooses a random number N_b and sends back to A the pair {N_a, N_b} encrypted with A's public key
                    - in this way B has shown that he can decrypt his public key, as he has accessed N_a
                - A sends N_b back to B to confirm his identity (encrypting it with the public key of B)
            - sending messages
                - once the identities are confirmed you send the message encrypting it with the recipient's public key
        - description of the system
            - the network is modeled as a collection of shared states isi among the main actors of the protocol
            - each message on the network specifies a particular state of the network
    - **threat model**
        - attackers can save and send back on the network
        - cannot alter the content of encrypted messages
        - can create encrypted messages using their private key (decryptable only with their own public key)
    - **security policies**
        - invariant of integrity: exchange of nonce (N_a, N_b)
        - secrecy: no attacker can decrypt and read anyone's secrets

- **NS**
    - **E** deceives **A** and convinces her to initiate a communication with her and subsequently forwards the messages a **B** convincing him to be **A**
        - **A** sends to **E** 's public key **E**
        - **E** 's public key **B** and sends it
        - **B** decrypts and sends to **E** (who he believes to be **A**) the message {N_a, N_b} encrypted with **A**
        - **E** 's secret key **A**, so it forwards the message to **A**
        - A qpoint **A** decrypts the message and as the last step in establishing communication **A - E** sends to **E** {K_b} encrypted with **E**

        - the nonces are random numbers, you don't know who they belong to: **A** sends his nonce and yes expects a pair of nonces behind, the first of which is his. To prove that it is really **A** (that is, that it has the private key necessary to decrypt) **A** decrypts and sends back to **E** the nonce that it believes to be from **E**, when in reality it is from **B**
    - **E** decrypts and reads N_b, policy violated

- the vulnerability 'and 'was found using a **model checker**

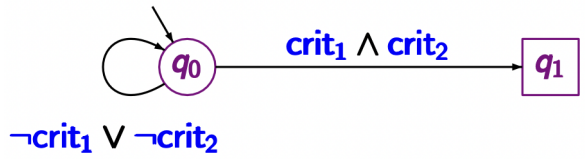
- **MODEL CHECKER as fast as possible**
  - nondeterministic finite state automaton

    NFA $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$
    - Q finite set of states
    - Σ alphabet
    - $\delta : Q \times \Sigma \to 2^Q$ transition relation
    - $Q_0 \subseteq Q$ set of initial states
    - $F \subseteq Q$ set of final states, also called accept states
  - **run** automaton
    - take a sequence (called **word)** of symbols $A_0\, A_1\, ..A_n$ belonging to the alphabet
    - we define run π as the

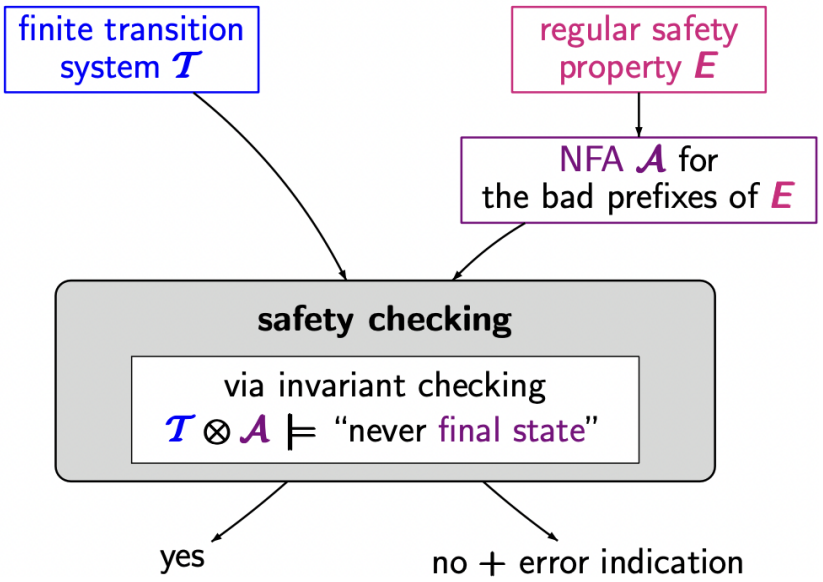
      state sequence $\pi = q_0\, q_1 \ldots q_n$ where $q_0 \in Q_0$ and $q_{i+1} \in \delta(q_i, A_i)$ for $0 \leq i < n$
    - a run π is called **accepting** if the final state q_n is a final state (i.e. 'belongs to F)
  - the language L (A) ⊆ Σ^* accepted by the automaton is defined as the set of all finite words ⊆ Σ which have an accepting run

  - **Regular Properties**
    - a property E is a subset of runs (E ⊆ L (A))
    - a property is said to be regular if its **bad prefixes** (i.e. its subtracks which are not accepting) are recognized from another automaton B
    - **EXAMPLE:** MUTEX
      - we want to guarantee the mutual exclusion

        - of the automaton it recieves he knows the traces that DO NOT guarantee mutual exclusion and the following (automaton B)



        - the automaton accepts the word if and only if two critical operations are done together
            - we practically express the security policy that we want to verify with an automaton that verifies the denial of that policy
- **verifying regular properties through model checking**
    - is given a system model T (expressed as a set of runs)
        - the system model expresses the behaviors of the system
    - and a regular property E is given (expressed by an NFA for bad prefixes) the question and ' **T satisfies E?** ($T \models E$?)



    - T are the behaviors of the system
    - A and the automaton that ends if the security property is violated
        - I "execute" the automaton with all the runs
        - if I never "finish" it means that the property is never violated
    - as actually doing the derivation we don't see it, but it is possible to do it relatively efficiently

- model checkers fall into the category of formal

    - methods methods to eliminate flaws (design or implementation) so as to
        - make the system safe
        - make sure that, even if safe , the system does (only) what we expect

- a threat model that we will use often instead is the following
    - **Dolev Yao Threat Model:** attackers
        - can eavesdrop all messages on the network and can break messages into parts
        - can save what they eavesdrop
        - can remove messages from the network
        - can send messages (new or overheard) to anyone on the network
        - can not compromise in any way everything that involves encryption without having the right

- classes of defences
    - **isolation:** the execution of a program is somehow prevented from "touching" the execution of other programs
    - **monitoring:** program that monitors the execution and operations performed at runtime of another program, possibly blocking the execution
    - **obfuscation :** the code or data is transmitted in such a way that it is understandable only to those who know a secret, which is hidden from the attacker

- **MEMORY CORRUPTION**
    - problem known for decades but still one of the main causes of security problems
    - introduced in the implementation phase

- we must ask ourselves
    - how do memory corruption flaws work?
    - what can their impact be?
    - **how can we spot these flaws?**
        - static and dynamic analysis
    - **what can the underlying platform do?**
        - the language toolchain should help
    - devs what can the programmer do about it?

- example

- let's consider the following code C

```
char buffer [4];
buffer [4] = 'a';
```

  - when the second line is executed **anything can happen**
    - if you are lucky segmentation fault
    - otherwise, if the attacker can check the value 'a', you will start a remote code execution
  - for the semantics of C in this case **anything can happen**
    - access to a part of "random" memory, the result is absolutely unpredictable.
    - The compiler in fact assumes that "ok, in this case anything is fine"
      - often the red line is "optimized" with a skip
        - if all goes well everything is certainly fine also do nothing

- programs written in C (or in C ++) they are vulnerable to memory corruption because they do not provide any tools to avoid the problem
  - the responsibility is totally left to the developer
    - => memory corruption is the main cause of
- classic attacks cases of memory corruption
  - access out of bounds to arrays or buffers
  - stuff with the 'pointer arithmetic
  - dereferencing null
  - pointers dangling pointers
  - memory allocation without verifying success
    - the malloc returns null in case of
  - memory leak error due to lack of free
    - garbage collector, the memory is left dirty

```
1000  …
1001  void f (){
1002    char* buf,*buf1;
1003    buf = malloc(100);
1004    buf[0] = 'a';
        ...
2001    free(buf1);
2002    buf[0] = 'b';
        ...
3001    free(buf);
3002    buf[0] = 'c';
3003    buf1 = malloc(100);
3004    buf[0] = 'd'
3005  }
```

possible null dereference (if malloc failed)

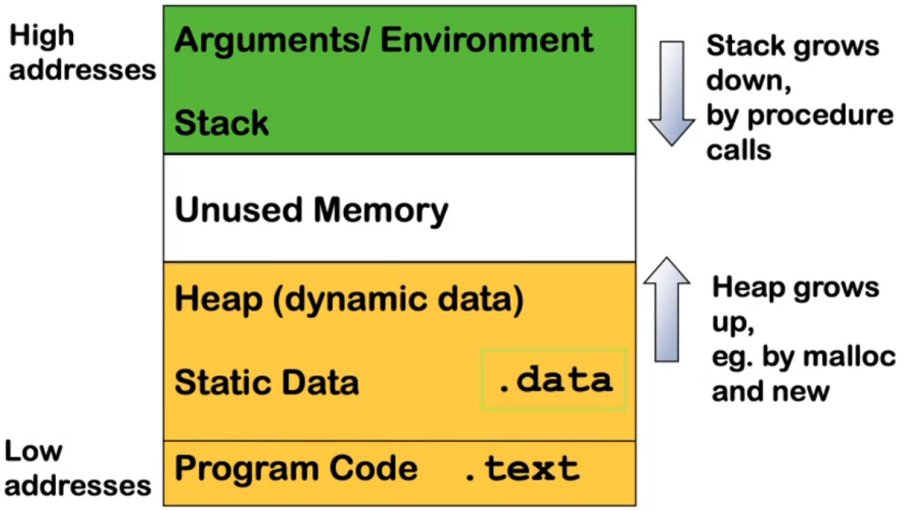
potential use-after-free if buf & buf1 are aliased

use-after-free; buf[0] points to de-allocated memory

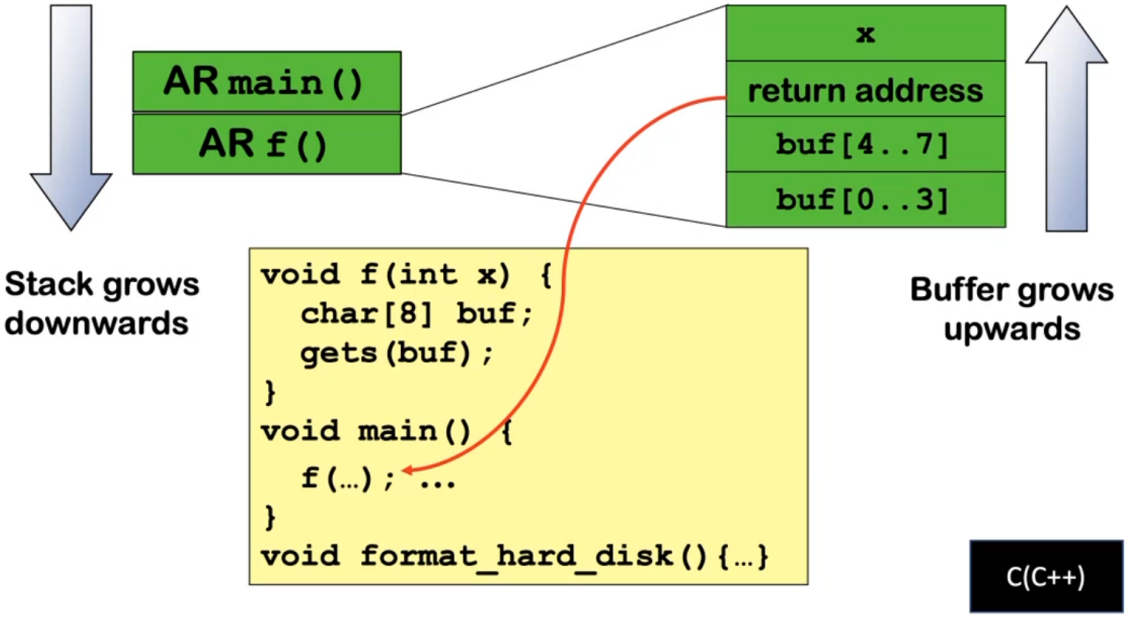
memory leak; pointer buf1 to this memory is lost & memory is never freed

use-after-free, but now buf[0] might point to memory that has now been re-allocated

- let's see the runtime of a compiled language (the C)
  - **memory layout**



  - **stack**



- the compiler determines the layout of the activation records at compile-time i generates the code that, when executed at runtime, will access the right locations
  - ditto for the heap
- **epilogue & prologue:** code generated by the compiler that is executed before the call function (push & pop AR)
- **the data layout and data access are designed together with the compiler**

- **example:** let's now look at a C library function **gets (char * str)**
  - reads from stdin and saves in the string pointed to by str
    - the reading ends *only* when a newline or an EOL is read
  - looks at the stack figure: if the function f () invokes a gets the attacker could overflow and overwrite the return address of the activation record, and then **execute anything**
    - **code reuse attack**: the attacker overwrites the return address making it point to already existing code (format_hard_disk)
    - **code injection**: the attacker inserts his code into a buffer and points the return address to that buffer
    - note that depending on how the compiler allocates on the stack there are things that may not be overwritten (an int before buff [0..3] would always be safe)

- **example:** library function **(char * src, char * dst)**
  - same thing: the strcpy assumes that dst is long enough and that src is correctly terminated
  - **nb:** neache **strncpy (char * src, char * dst, int n)** is safe

        - I tell it to copy in dst **less** than the real size of src (a character and less, just the terminator)
        - in dst an unfinished string ends there, now the attacker can read to the bitter end
            - the good dev inserts the terminator in dst explicitly after the strncpy
                - the good devs are very rare

- example: **format string attack** through **printf**
    - printf (string) where string is passed by the user through stdin
        - the malicious user passes me "% x% x% x% x% x% x% x% x% x"
            - % x reads and printa bytes
            - I delivered to all 'attacker all the portion of the stack resting on the string allocation

- to give the idea of how unsafe C is
    - "in a correct program the addition to a pointer will not produce a pointer out of the allocated memory portion"
        - **the GCC assumes that no pointer overflow can ever happen on any architecture**

```
if (buf + len> = buf_end)
    return -1;
```

            - any such checks are **eliminated** by the compiler at compile time because the guard is always assumed to be bogus
                - *bug caught by the linux kernel*
            - *which optimizations are reasonable and which introduce errors or flaws?*

- **in general**
    - right now we rely on **defensive programming**: the programmer while writing is thinking about all the ways in which an attacker could act and plug the holes in the implementation phase
    - it would be better if the language itself guaranteed that the strings are always terminated, without trusting the programmer who has other things to do

- **safe programming language:**

    - in a safe language the programmer does not have to worry about accesses out of bounds
        - the runtime realizes this and reacts appropriately (exceptions ..)
    - adequate type systems
    - control for the integer overflow
    - the allocated memory is initialized by default
    - the inaccessible memory is freed automatically
    - ...
- **safe vs unsafe**
    - **unsafe:** accepts and executes also non-sensible statements (eg access out of bound)
        - the semantics of such instructions are undefined: in fact *anything can happen*
        - **the responsibility is left to the programmer**: he must be the one to make sure that only instruction the correct (and safe) are then actually executed
            - not only must "program well" but must also (know and) take into account the flaws introduced by the underlying platform (which goes from the compiler to the operating system up to the cpu)
    - **safe:** the language is ensures that a statement is executed if and only if it makes sense and, when it is not, a specific error is reported (such as java exceptions)
        - **the responsibility 'is (as much as possible) of the language: it** is the language to having to somehow identify (statically and / or at runtime) and / or prevent the execution of wrong statements
    - C / C ++ are obviously unsafe
    - Java and C # are designed to be safe
        - but can offer unsafe
            - java native interface features: it allows you to invoke native machine code
            - ...
    - **therefore:** a language is considered safe if and when it offers **memory safety** and **type safety**, and also
        - the **trusted computer base (TCB)** (i.e. the underlying platform for the execution of the code you are writing) and must manage the behaviors of the
            - example language: the jvm that executes the bytecode must maintain the security abstractions of java (duh)

        - must be guaranteed ' **compositionality**
            - in a safe language it is possible to understand the behavior of a program in modular way, and it is possible to ascertain the safety of a code fragment by looking only at that single fragment, without having to understand and check everything
        - **that the memory unsafety breaks compositionality.**
            - else.Notememory unsafe
                - being memory unsafe the execution of P can access all the memory: it could therefore also alter the data used by module Q
                    - that the code of module Q is safe or not is therefore irrelevant: we cannot guarantee the safety of Q because this could be compromised at runtime by the execution of another module
                        - policy = "Q does not allow information leaks": perhaps the source of Q and its ese cuction verify the policy, but this is compromised by P that could make a print attack and leak all the data in memory
        - . In principle, a program written with a safe language **cannot crash due to segmentation fault.**
            - Theoretically we could disable the memory checks made by the OS
                - in reality the type-checkers, the compilers and the whole toolchain are still pieces of code written by humans, so they could in turn have bugs and allow (rare) behaviors not allowed
                    - . memory access control of the operating system is however fundamental

- **PLATFORM LEVEL DEFENCES (RUNTIME & MEMORY ORGANIZATION)**
    - methods of defense against attacks provided by the compiler, the OS, the hw machine and so on *without the dev being aware*
        - some techniques may need support hw
        - others introduce overhead
        - others break compatibility with already compiled codes

- **Defense # 1: STACK CANARIES**
    - a dummy value (called canary or cookie) is inserted at compile time on the stack in front of the return address

        - the overflow corrupts the canary and the attack is detected
            - very little overhead
    - the compiler that implements the canaries produces epilogue and prologue different from the original compiler!
        - prologue and epilogue are richer!
            - **runtime support is extended**

```
void fun (const char * input) {
    char buffer [12];
    strcpy (buffer, input);
}
```

            - the **prologue** must push the canary onto the stack immediately after the control data (the return address) and before allocating the local variables
            - **the canary-enabled epilogue** must check that the canary still contains the original value before resetting the program counter

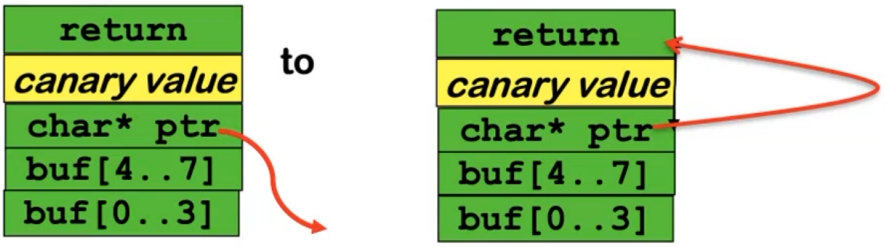
```
extern uintptr_t __stack_chk_guard;
noreturn void __stack_chk_fail(void);

void a_function(const char* input)

uintptr_t canary = __stack_chk_guard;

char buffer[12];
strcpy(buffer, input);

if ((canary = canary ^ __stack_chk_guard) != 0 )
    __stack_chk_fail();
```

            - with
                - __stack_chk_guard = original value of the canary
                - __stack_chk_fall = callback function called in case of detected stack buffer overflow

A careful attacker can still defeat canaries, by

- **overwriting the canary with the correct value**
- **corrupting a pointer to point to the return address to then change the return address without killing the canary**

eg changing



- heap overflow attacks are not minimally managed

- **Defense # 2: IBM ProPolice**
  - overflow is dependent on the order in which the stuff is allocated
    - random rearrangement of the stack to reduce damage
      - even better: allocate all buffers on top, on top of all the other elements

- **Defense # 3: Shadow Stack**
  - a hidden stack (and that must not be compromised) keeps a copy of the return address
  - overhead higher but higher security

- **Defense # 4: Non Executable Memory (NX Memory)**
  - we structure the (virtual) machine of execution of the language so that there are portions of "executable" and other "non-executable"
    - **memory executable memory:** contains the code
    - **non-executable memory:** for data storing
    - the cpu or virtual machine (eg. jvm) refuses to execute code contained in non-executable memory
      - the attacker does not it can no longer jump to its code as all inputs (and therefore any malicious code that the attacker passes me) are saved in non executable memory
  - intel calls it eXecute-Disable (XD)
  - AMD calls it enhanced virus protection
  - **limitations**
    - upfar we have only considered cases of ahead-compiling
      - I have the code and I compile it all (and I get an executable target code (from the cpu or from the virtual machine it doesn't matter)

        - NX Memory fails in case d ella **Just In Time Compilation**
            - I read a line from the source
            - the co   mpile
            - I
            - read the next line
    - the case of **code injection** is covered but the **code reuse attack** is still possible
        - *so you have to be very careful with the*
            - **libc** for example it offers a lot of functionality potentially usable in a malicious way (exec (), system (), ..)
    - the compiler translates the function calls of the source code into `call <address>` of the target code (for example the assembly) where <address> is the address of the function code
        - in C for the invocation of the function f () it may be possible to know the static address (or the correct offset) of the code a compile (staitc) time
        - if the compiler manages to hard-wire the addresses in the binary and NX memory is used it is very difficult for the attacker to do damage
            - when it fails (for example with virtual methods in C ++: `or -> m ();` the address of m () is determined at runtime by looking at the virtual method table (vtable)
                - in these cases it is possible to do damage

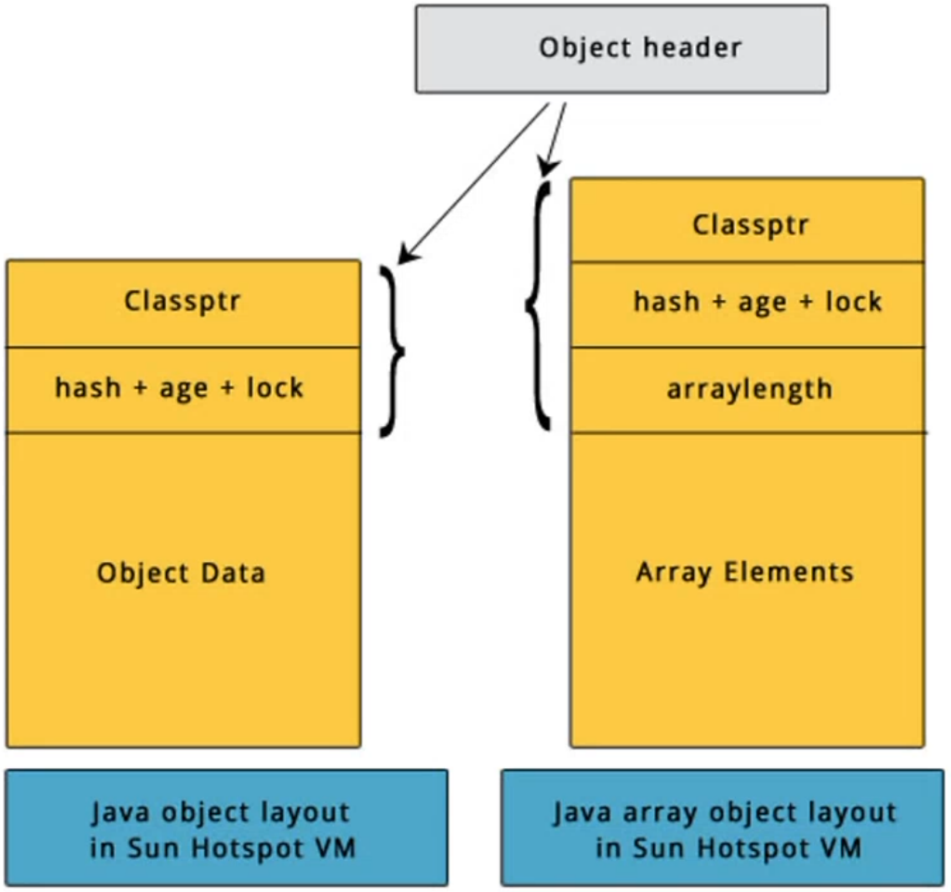
- **Defense # 5: Fat Pointers**
    - we have seen that many of the problems related to **memory safety** are due to the scarcity of information regarding the memory (in C a pointer tells you ***only*** where the memory area it points to begins and the type of data it contains)
    - intuition: add information to the pointer regarding the size of the *memory chunk* pointing
        - the compiler
            - records the information relating to the dim ension

              ```
              char * p = malloc (5 * sizeof (char))
              ```

            - pointer: p -> **[s]**[t] [u] [f] [f]
            - **fat pointer** [p | 5] -> **[s]**[t] [u] [f] [f]

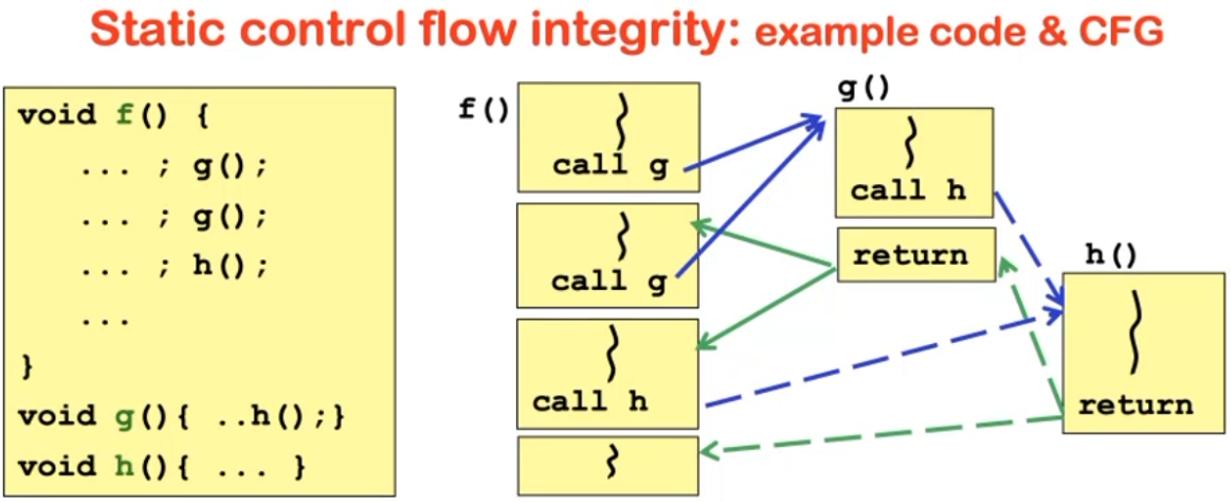
- has two major disadvantages
    - execution overhead notable
    - non-binary compatible: all existing C code must be recompiled with the new compiler that adopts fat pointers

- **Defense # 6: "Typed Data Descriptor"**
    - idea on the same strand of fat pointers: adding information about the size
        - here instead of adding info to the pointers we add them directly to the memory chunks
    - and 'a solution adopted by JAVA, for example
        - JVM: the *data* contain information regarding the data
        - controls at runtime, increased overhead



- **CONTROL FLOW INTEGRITY (CFI)**
    - extra information and controls to identify unexpected control flows
        - the code has a flow (REPL, with related function calls)
        - an attacker can compromise the flow in various ways and for various purposes
            - code injection, code reuse attack, ...

- so far we have seen **dynamic return integrity** (d ynamic, that is' runtime techniques that require "only" to change the language toolchain)
  - stack canaries and shadow stacks
    - allow checks and verifications to avoid the manipulation of the return points
- **static return integrity**
  - the idea is to perform static analyzes on the code and to determine the **control flow graph (cfg)** and then to monitor that at runtime no jumps that are not allowed or foreseen are made
    - if in the code of the function f () the call to the function g () never appears then if at runtime g () is called while executing f () there is something suspicious (as well as the return from g () to f () is suspect)
  - **example:**



Before and/or after every control transfer (function call or return) we could check if it is legal – ie. allowed by the cfg

Some weird returns would still be allowed

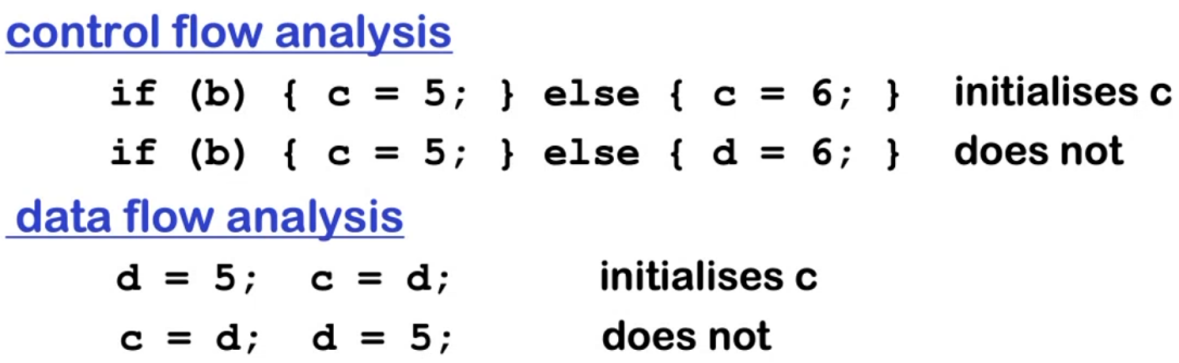
- **eg if we call `h()` from `g()`, and the return is to `f()`, this would be allowed by the static cfg**
- **Additional *dynamic* return integrity check can narrow this down to actual call site – using recorded call site on shadow stack**

- **STATIC ANALYSIS (FOR SECURITY)**
  - until now we have focused on the runtime
  - now we see security techniques used in the backend of the compilers

        - we always assume to get the abstract syntax tree (AST) from the frontend: the programs are syntactically correct
    - automated analysis at compile time for the detection of bugs
    - various techniques
        - simple static checks (grep on patterns)
        - type checking (int + bool not allowed)
        - advanced analyzes that take into account the semantics
            - dataflow analysis
            - control flow analysis
            - symbolic eval
            - model checking
            - ...
    - in the sw development process the static analysis is in fact called the **code review**
        - I wrote the code, I designed and wrote the tests: before testing I do code review
    - the static analysis is fundamental
        - the code is too and too complex to do the check manual
        - test cases can leave out particular and unexpected cases (strange inputs built ad hoc)
    - static analysis tries to answer the following questions
        - the prog ramma ends with every input?
        - how big can the heap get?
        - can there be leaks of confidential information?
        - ...
        - what will be in this variable x?
        - can the pointer p be null?
        - p and q point to two different structures in the heap?
        - ...
    - in fact to do static analysis we use **a program that analyzes programs**
        - program analyzer: a program that takes as input another program P and tells us if P is correct or if P fails for certain inputs (etc.)
            - we obviously would like the analyzer perfect
                - **soundness**: **all** errors
                - **completeness:** no false alarms are raised
                - **termination:** the analyzer always terminates
            - **rice theorem:** all non-trivial properties concerning the behavior of programs written with turing-equivalent languages are undecidable

        - we should therefore resort approximation, that is, we would like
            - an acceptable number of false positives / negatives
            - not too many warnings
            - a good error reporting
                - that the bugs are easily fixed
            - the possibility of teaching the analyzer which false positives are to be ignored
    - **NB:** the program analyzer tends to be "included ”In the compiler

- we see short examples of static analysis
    - warnings relating to unused, uninitialized or al dead code (post return) are all examples of static analysis
        - prohibiting uninitialized variables could be not so much the static analysis but the definition of the language itself (eg java)



        - control flow analysis
            - the second case gives an error because cod I'm sure not initialized

- there are obviously limits to what the analyzer can do

```
if (b) {
    c = 5
}
```

    - the variable c is initialized? depends on the value of b
        - the analyzer as mentioned can either answer "boh" or give false positives / negatives

- **correctness** vs **security**
  - the static analysis performs (among other things) some optimizations at compile time
  - **example:** consider the following C code

```
void foo()
{
   int x;

   x = read_password();
   use(x);
   x = 0; // clear password
   untrusted();
   return;
}
```

    - the value 0 assigned to x is never used
      - **dead store optimization**: assignments whose values are not used are deleted it
        - optimization **correct** in terms of efficiency but it is terrible in terms of **security**
          - `x = 0` it is statically replaced (without notifying the dev) with `skip` and the program (which was initially safe) is transformed into an unsafe program by the compiler
  - the **challenge** is therefore balancing the optimizations
    - the performances are crucial, but now safety is just as important the

- static analysis terms, given a program,are identified **program points**
  - a **program point** is a point in the code structure which will then correspond to a value in the program counter (PC)
    - any point in the program = any value of the

- **invariant PC:** a property that holds at every execution point of the program
  - a property is invariant **in a program point** if for each possible input the property is verified when it "arrives" at that point

- **Safe Programming Languages Design**

- to design a programming language (and to write programs) it is necessary to decide / understand how the language works
  - the semantics of the operational semantics language must be defined
    - : definition of rules for step-by-step evaluation until you get a
    - **semadenotational technique:** defines mathematical functions for the interpretation of language constructs and evaluates the program by combining the
    - **axiomatic semantic functions:** provides rules for reasoning about programs
      - (example: hoare triples)
  - here we see the **denotational semantics** and we will useto define small functional programming languages that adopt specific techniques for safety
    - the interpreter of the languages we will write will be implemented in OCaml
      - it is very convenient to represent the data and simulate the REPL
      - **NB:** OCaml is an eager language (not lazy) e quindi la valutazione dei parametri viene fatta prima di passarli!

- **MicroFUN**: il nostro linguaggio di programmazione funzionale
- **(OCaml) AST per MicroFUN:** rappresentazione della sintassi in termini di sintassi astratta

```
type expr =
  | CstI of int
  | CstB of bool
  | Var of string
  | Let of string * expr * expr
  | Prim of string * expr * expr
  | If of expr * expr * expr
  | Fun of string * expr   (* Lambda , param Body *)
  | Call of expr * expr
```

- **NB:**
  - `Prim of string * expr * expr`

        - espressione aritmetica primitiva
            - `string` rappresenta il sibolo dell'operazione (+,...)
            - le altre due `expr` sono gli operandi
    - `Let of string * expr * expr` ~~> `let ... = ... in ...`
    - `Fun of string * expr`
        - funzioni anonime con un parametro e un corpo
            - `string` e' l'identificativo del parametro formale
            - `expr` e' il corpo della funzione
        - posso dare un nome ad una funzione usando Let che associa un identificativo ad una expr (che può essere la Fun)
    - `Call of expr * expr`
        - invocazione di funzione: prende la funzione stessa e la valutazione del parametro attuale
    - in MicroFUN non e' prevista la ricorsione
        - mancando la ricorsione e il costrutto per il ciclo vediamo che il linguaggio non e' turing equivalente (pace)

- **eseguire** un programma scritto in MicroFun (cioè valutarlo con il nostro interprete) significa quindi ricevere un AST e valutarlo in accordo alla precedenza degli operatori
    - di fatto facciamo una visita dell'AST fin tanto che non e' prodotto un valore numerico o booleano

- **MicroFun RunTime Structure (Stack)**

An environment is a map from identifier to a value (what the identifier is bound to). We represent the environment as an association list, i.e., a list of pair (identifier, data).

```
type 'v env = (string * 'v) list
```

Given an environment {env} and an identifier {x} it returns the data {x} is bound to. If there is no binding, it raises an exception.

```
let rec lookup env x =
   match env with
   | []      -> failwith (x ^ " not found")
   | (y, v)::r -> if x=y then v else lookup r x
```

- **MicroFun Runtime Values**
  - **MicroFun ha lo scoping statico**: la visibilità delle variabili dipende dalla struttura statica del sorgente (dalla posizione delle variabili nel codice)
    - come lo C
  - quando dichiaro una funzione questa avrà associato un ambiente (env) di visibilità che corrisponde allo scoping della funzione nel momento in cui questa e' stata dichiarata
    - in questo modo la funzione "vede" sempre e solo ciò che gli e' visibile staticamente
      - al contrario nello scoping dinamico la visibilità dipende dall'esecuzione a runtime del programma
  - rappresentiamo i booleani come interi
    - 0 e' falso, 1 e' vero
  - **i tipi a runtime sono i seguenti**

```
type value =
  | Int of int
  | Closure of string * expr * value env
```

    - string e' il nome del parametro formale
    - expr e' il corpo della funzione
    - value env e' l'ambiente statico (sintattico)
    - **attenzione:** MicroFun e' funzionale e come tipico **le funzioni sono valori tanto quanto gli interi**

- **MicroFun Interpreter**
  - per definire l'interprete dobbiamo dare la semantica (denotazionale, espressa con regole di inferenza)
    - nb: la parte gialla e' "l'implementazione" della regola di inferenza

  - **valutazione identificativo**

$$\frac{env(x) = v}{env \triangleright Var\ \mathrm{x} \Rightarrow v}$$

eval (Var x) env -> lookup env x

- **operazione aritmetica primitiva**

$$\frac{env \triangleright e1 \Rightarrow v1 \quad env \triangleright e2 \Rightarrow v2}{env \triangleright \text{Prim}(+,e1,e2) \Rightarrow v1 + v2}$$

```
eval Prim("+", e1, e2) env ->
                    eval e1 env + eval e2 env
```

- **let**

$$\frac{env \triangleright erhs \Rightarrow xval \quad env[xval / x] \triangleright ebody \Rightarrow v}{env \triangleright \text{Let x = erhs in ebody} \Rightarrow v}$$

```
eval (Let(x, erhs, ebody)) env ->
      let xval = eval erhs env in
        let env1 = (x, xval) :: env in
          eval ebody env1
```

  - ovviamente il **let** esegue operazioni sull'ambiente
    - ricorda che il let ha la forma del let ... = ... in ...
      - valutiamo erhs (il valore da assegnare ad x) nell'ambiente corrente e lo mettiamo in xval
      - creiamo un nuovo ambiente env1 con la nuova associazione (x, xval)
      - valutiamo il body del let nel nuovo ambiente env1
        - stiamo di fatto simulando il push e il pop dei record di attivazione sullo stack
          - (push ~~> valutazione body ~~> pop)

- **Lambda**
  - cosa succede quando nell'ambiente corrente trovo una lambda?
    - bisogna costruire la chiusura (così' da poter accorpare l'ambiente statico)

$$env \triangleright Fun(x,e) \Rightarrow Closure(\text{“}x\text{”}, e, env)$$

      - x: nome del parametro formale
      - e: corpo (codice) della funzione

        - env: puntatore all'ambiente **corrente**, cioè quello in cui la funzione viene dichiarata
    - cosa succede se va valutato nell'ambiente un identificativo che corrisponde ad una chiusura?

$$\frac{env \rhd Var("f") \Longrightarrow Closure("x", body, fDecEnv) \quad env \rhd \arg \Longrightarrow va \quad fDecEnv[^{va}/_{x}] \rhd body \Longrightarrow v}{env \rhd Call(\mathbf{eval}("f"), arg) \Longrightarrow v}$$

- fDecEnv: ambiente di dichiarazione della funzione
- va: valutazione dell'argomento
- step1: valuto il parametro (estendendo l'ambiente corrente ma solo perche' i let hanno sempre **in**
- step2: estendo l'ammbiente fDecEnv con l'associazione relativa al parametro (associo ad x (parametro formale) il valore del parametro (va)
- step3: valuto il body nel nuovo ambiente con
    - se questo mi ritorna il valore v allora la valutazione della chiamata funzione e' v
- anche in questo caso e' chiara la simulazione del push e del pop nello stack del record di attivazione relativo alla funzione chiamata

- **MicroFun Interpreter Code**

```
let rec eval (e : expr) (env : value env) : value =
    match e with
    | CstI i -> Int i
    | CstB b -> Int (if b then 1 else 0)
    | Var x  -> lookup env x
    | Prim(ope, e1, e2) ->
     let v1 = eval e1 env in
     let v2 = eval e2 env in
     begin
     match (ope, v1, v2) with
     | ("*", Int i1, Int i2) -> Int (i1 * i2)
     | ("+", Int i1, Int i2) -> Int (i1 + i2)
     | ("-", Int i1, Int i2) -> Int (i1 - i2)
     | ("=", Int i1, Int i2) -> Int (if i1 = i2 then 1 else 0)
     | ("<", Int i1, Int i2) -> Int (if i1 < i2 then 1 else 0)
     |  _ -> failwith "unknown primitive or wrong type"
     end
```

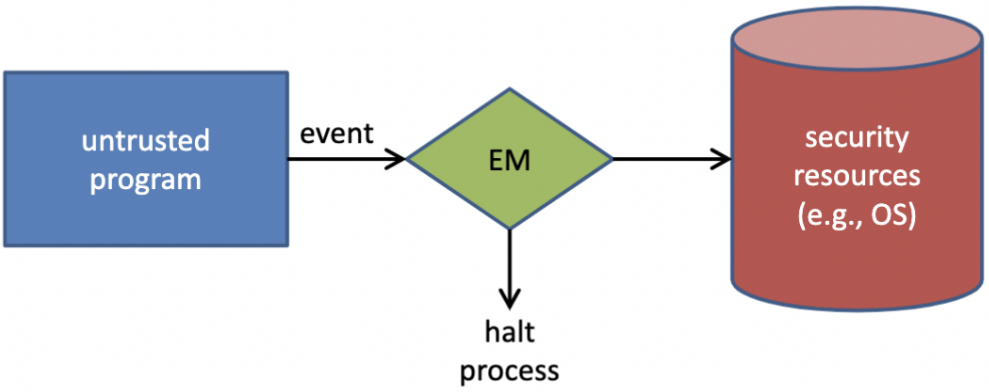
```
| Let(x, eRhs, letBody) ->
    let xVal = eval eRhs env in
    let letEnv = (x, xVal) :: env in
    eval letBody letEnv
  | If(e1, e2, e3) ->
    begin
    match eval e1 env with
    | Int 0 -> eval e3 env
    | Int _ -> eval e2 env
    | _     -> failwith "eval If"
    end
```

```
| Fun(x,fBody) -> Closure(x, fBody, env)
```

```
| Call(eFun, eArg) ->
    let fClosure = eval eFun env in
    begin
    match fClosure with
    | Closure (x, fBody, fDeclEnv) ->
     let xVal = eval eArg env in
     let fBodyEnv = (x, xVal) :: fDeclEnv
     in eval fBody fBodyEnv
    | _ -> failwith "eval Call: not a function"
    end
```

- **SECURITY POLICIES**
  - abbiamo, in linguaggio di programmazione a scelta, un meccanismo M utilizzabile dal programmatore
  - il programmatore si definisce una sua policy di sicurezza P (in base a ciò che il linguaggio permette)
  - **possiamo provare che il meccanismo M enforce la policy P?**
    - quali sono le buone policy?
    - possiamo esprimere tutte le policy?
    - ...
    - come funziona M?

- **Execution Monitor (EM)**
  - un meccanismo (un programma) che controlla (monitora) l'esecuzione a runtime di un programma untrusted
    - gli eventi sono mediati dall'EM (ad esempio le system call)
  - esempio: file system access control
    - l'EM e' (dentro) l'OS
    - controlla la validità' delle policy usando le l'access control list



- gli EM sono moduli che vengono eseguiti in parallelo con un'applicazione untrusted
  - lo scopo e' di prevenire, individuare e rimediare agli errori (o le operazioni volutamente malevoli) che l'applicazione esegue a runtime
- le decisioni dell'EM possono essere prese basandosi sulla execution history
  - cioè valuta che la policy di sicurezza sia rispettata prima dell'operazione corrente e che sia rispettata dopo il termine dell'operazione
  - in caso contrario l'EM non permette (o annulla) l'operazione

- **Programs and Policies**
  - An *execution (or trace)* ***s*** is a sequence of security-relevant program *events* **e** (also called actions)
  - Sequence may be **finite** or **(countably) infinite**
    - $s = e_1; e_2; \ldots; e_k; e_{halt}$
    - $s = e_1; e_2; \ldots; e_k; \ldots$
    - The empty sequence ε is an execution
    - If s is the execution $e_1; e_2; \ldots; e_i; \ldots e_l; \ldots$ then **s[i]** is the execution $e_1; e_2; \ldots; e_i;$
  - We simplify the formalism.
    - We model program termination as an infinite repetition of e_{halt} event.
    - Result: now all executions are infinite length sequences

- vediamo un programma S come un insieme di tracce (S1, S2, ...)
- una policy P e' una proprieta' del programma (o dei programmi)
  - una policy divide lo spazio dei programmi in due insiemi
    - accettabili (permessi)
    - non accettabili (non permessi)
      - questi programmi sono in qualche modo “censurati”
        - terminati nelle run che violano P

    - esempio di policy
        - **access control policy:** relativa all'utilizzo di certe risorse (tipo file) o chiamate di sistema
        - **availability policy**: se un programma acquisisce una risorsa deve anche rilasciarla (prima o poi)
        - **bounded availability policy:** se un programma acquisisce una risorsa deve rilasciarla entro un momento prefissato (eg: entro l'esecuzione di questo program point)
- la proprietà' P denota l'insieme di tracce (un linguaggio di tracce) L(P)
    - la traccia **s** soddisfa la proprietà P sse **s** appartiene a L(P)

- **Quali sono le politiche che possono essere verificate (enforced) da un Execution Monitor?**

    1. EM policies are universally quantified predicates over executions
        - $\forall s. P(s)$
        - **Policy P is called the detector.**
    2. The detector must must be prefix-closed
        - **P(ε) holds**
        - **P(s;e) holds then P(s) holds**
    3. If the detector is not satisfied by a sequence (the detector recjects the sequence) then it must do so in finite time
        - $\neg P(s) \implies \exists i \neg Ps[i]$

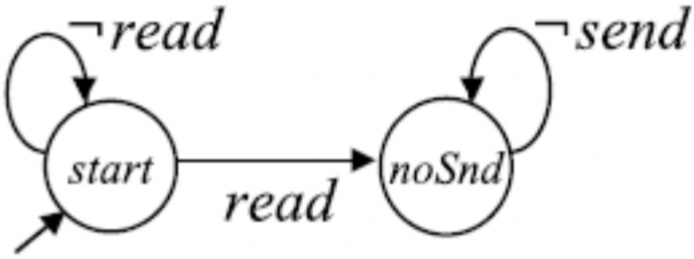
    - **politica di safety:** una policy che soddisfa (1), (2) e (3)
        - “a bad thing never happens”
            - ad ogni passo di esecuzione del sistema il sistema e' safe, appena smette di essere safe blocco tutto
    - **gli execution monitor permettono di soddisfare le politiche di safety**
    - Safety property can be enforced **using only traces** of program
        - If P(t) does not hold, then all extensions of t are also bad
    - Amenable to **run-time enforcement**: don’t need to know future
    - **Examples**:
        - **access control** (e.g. checking file permissions on file open)
        - **memory safety** (process does not read/write outside its own
        - memory space)
        - **type safety** (data accessed in accordance with type)
    - **proprietà di liveness:** “something good eventually happens”

    - nb: “liveness” significa “vitalità”
    - ad esempio la **nontermination:** “il server mail non smetterà di girare”
    - queste politiche non possono essere puramente enforced a runtime
- le proprietà di **safety** e di **liveness** sono cruciali in teoria
    - risultato teorico: ogni proprietà e' l'intersezione di safety property e liveness property

- **EM Behaviour**
    - gli EM possono solo vedere il passato, non hanno idea di cio' che succedera in futuro
        - ecco perché possono forzare proprietà di safety ma non di liveness
    - **si assume che**
        - il monitor ha accesso all'intero stato della computazione
        - il monitor può' avere uno stato arbitrariamente grande
        - le proprietà di safety enforced sono modulo la potenza di calcolo a disposizione
        - il predicato che il monitor usa per determinare se deve terminare l'esecuzione e' calcolabile

- **EM Enforcement**
    - analizza la singola esecuzione (corrente)
        - $p(\pi) : (\forall \sigma \in \pi : p(\sigma))$
    - l'esecuzione deve essere terminata non appena il prefisso invalida la policy
        - $\neg p(\tau) \Rightarrow (\forall \sigma : \neg p(\tau\sigma))$
            - τ e' una traccia finita
            - σ e' una traccia (non necessariamente finita)
            - la giustapposizione concatena due tracce
    - deve individuare le violazioni in tempo finito
        - $\neg p(\sigma) \Rightarrow (\exists i : \neg p(\sigma[..i]))$
            - $[..i]$ e' il postfix operator il prefisso di una data traccia lunga i step
    - **enforceable policy => safety property**
        - una **politica** enforceable rende rispettata una **proprietà**

- **Security Automata**
  - automi a stati finiti
    - riconoscono (accettano) il linguaggio delle esecuzioni permesse
    - come alfabeto hanno l'insieme degli eventi (definiti sopra)
    - sugli archi hanno l'evento corrente
    - tutti gli stati sono accettanti (prefix closed: ogni istruzione potrebbe essere l'ultima, fino a quando non si viola la policy tutto ok)
  - esempio facile: no sends after reads
    - la policy specifica che dopo aver eseguito una read (su un file) non si devono inviare dati (su una socket)
    - l'automa accetta tutte le esecuzioni che rispettano la policy



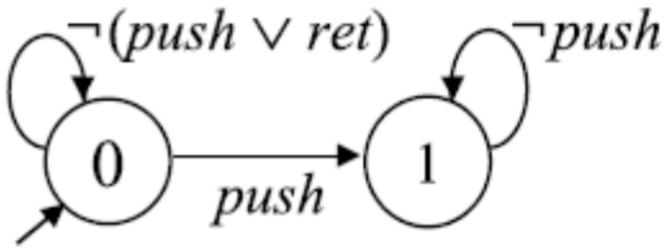
- Gli **Execution Monitor** *tradizionali* hanno diversi problemi
  - **inefficienti**: context-switch a nastro tra EM e programma monitorato
  - **TCB troppo grande:** l'EM estende l'OS
  - **deboli:** l'EM non può vedere le operazioni interne al programma ma solo le interazioni che il programma vorrebbe fare con l'OS
  - **poco modulari:** cambiare la policy richiede la modifica dell'OS

- **In-Lined Referenced Monitor (ILRM)**
  - idea di base: implementare l'execution monitor *by in-lining its logic into the untrusted code*
    - vogliamo hard-cablare la logica dell'EM nel codice eseguibile del programma untrusted
      - la procedura di inlining deve pero' essere automatizzata
      - ci sono diversi problemi da gestire
        - come generare automaticamente il codice dell'EM?
        - come preservare la logica del programma untrusted
        - come impedire ai programmi di corrompere l'EM?
  - **In-Lining Algorithm**
    - esegui l'inlining dell'automa prima di **ogni** evento (generato dal binario del programma untrusted)

        - valuta parzialmente (specializza) gli archi dell'automa prima dell'evento che controlla (alcuni archi spariranno del tutto)
        - genera il **guard code** per ciò che rimane degli automi

    - ***In-Lining Algorithm Actual Steps***
        - **insert security automata:** inserisci una copia dell'automa prima di ogni istruzione target nel binario del programma untrusted
        - **evaluate transitions:** valuta i predicati delle transizioni (predicato “l'istruzione target può causare la transizione?”) per ogni istruzione monitorata
        - **simplify automata:** elimina tutte le transizioni che hanno il transition predicate falso
        - **compile automata:** traduci l'automa rimanente di ogni istruzione in codice target.
            - se il codice aggiunto rileva una violazione della policy (cioè' l'auto rifiuta il suo input (esecuzione)) viene invocato un fail

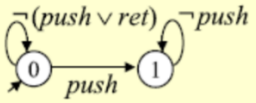
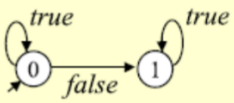
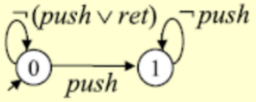
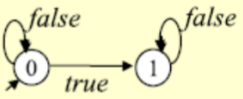
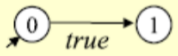
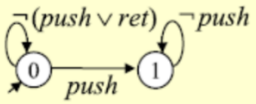
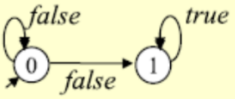
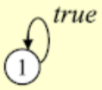
- **ILRM Case Study**
    - abbiamo la policy “push exactly once before return”



    - e vogliamo fare l'inline della policy nel seguente codice binario

```
mul r1, r0, r0
push r1
ret
```

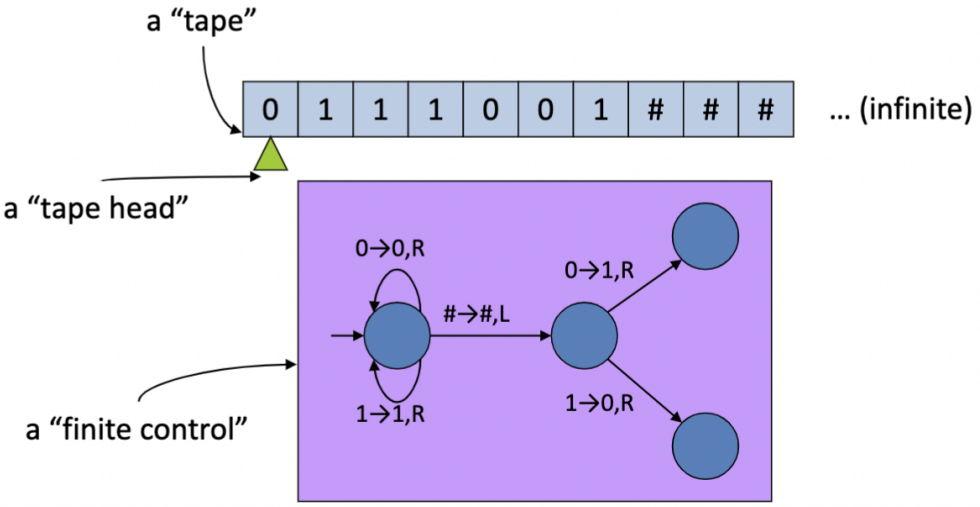
    - simulazione dell'algoritmo e' la seguente

Insert security automata	Evaluate transitions	Simplify automata	Compile automata
			
`mul r1,r0,r0`	`mul r1,r0,r0`	`mul r1,r0,r0`	`mul r1,r0,r0`
			`if state==0` `then state:=1` `else ABORT`
`push r1`	`push r1`	`push r1`	`push r1`
			`if state==0` `then ABORT`
`ret`	`ret`	`ret`	`ret`

- Teoria della Calcolabilità (duh)

Turing Machines and Computability

- Turing Machine
  - Alan Turing (1936)
  - simple mathematical model of a computer
  - consists of:



TM Expressive power

- Can do simple arithmetic
- TMs don't necessarily terminate
- Can evaluate a C program encoded in binary
- Can simulate arbitrary TMs (given as input) on arbitrary
- inputs (given as input): the "**universal TM**"
- Fact: **Can do anything a real computer can do** (**but very, very slowly**)
- TMs can't solve **undecidable problems** (e.g., **halting problem**)

- The **halting problem** is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever.
- Turing proved a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.
  - **the halting problem is undecidable over Turing machines**
- Turing result is significant to practical computing efforts, **defining a class of applications which no programming invention can possibly perform perfectly.**

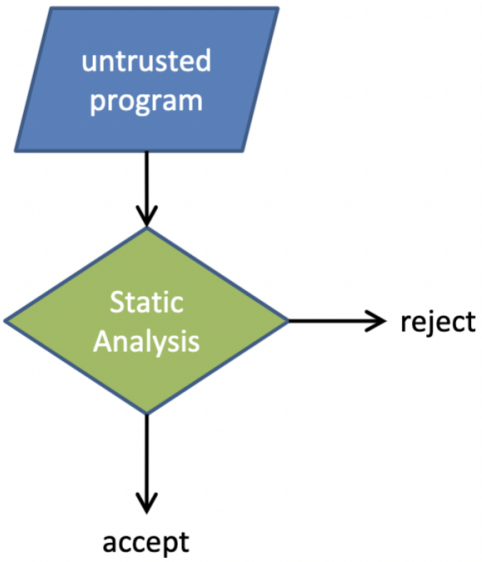
Computation theory

- RE (recursively enumerable) is the class of decision problems for which a 'yes' answer can be verified by a Turing machine in a finite amount of time.
  - if the answer to a problem instance is 'yes', then there is some procedure which takes finite time to determine this, and this procedure never falsely reports 'yes' when the true answer is 'no'. However, when the true answer is 'no', the procedure is not required to halt
- co-RE is the set of all languages that are complements of a language in RE.
  - co-RE contains languages of which membership can be disproved in a finite amount of time, but proving membership might take forever.

_

- **(Security Policy) Enforcement Strategies**
  - abbiamo quindi visto:

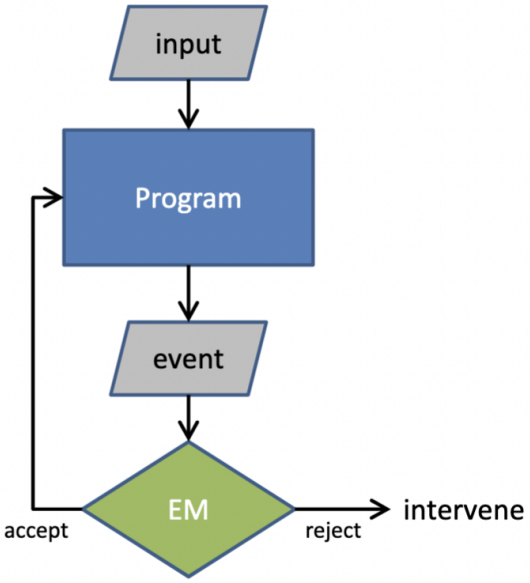
Take 1: static analysis



- Approach:
  - analyze untrusted code BEFORE it runs
  - return “accept” or “reject” in finite time
- Pros:
  - immediate answer
  - code runs at full speed
- Cons:
  - high load overhead
  - weak in power…?

Recursively Decidable Policies

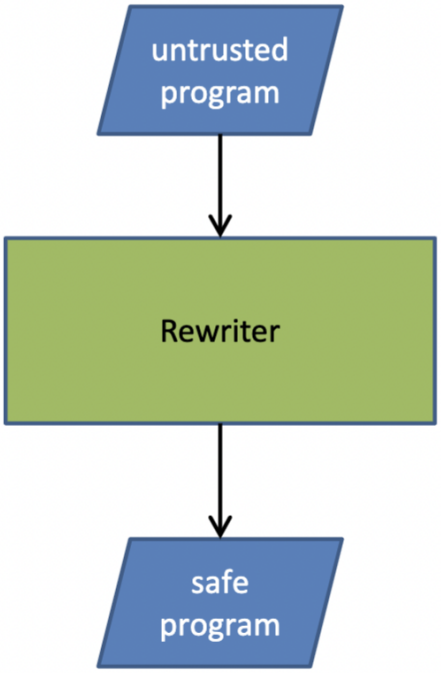
Take 2: EMs



- Approach:
  - EM monitors events
  - intervenes to prevent violations
  - implemented outside program
- Cons:
  - no answer until execution
  - runtime slow-down (context-switches)
- Pros:
  - lower load-time overhead than static analysis
  - more powerful…?

co-Recursively Enumerable Policies

IRM Strategy



- Approach:
  - transform untrusted code
  - must return new program in finite time
  - transformed code must satisfy policy
  - behavior of safe code must be preserved
- Pros:
  - lowest runtime overhead
  - load-time overhead is once-only
  - sometimes no answer until execution

- **Execution Monitor come primitiva del linguaggio di programmazione**
  - progettiamo un linguaggio di programmazione (basato su MicroFUN) che ha come primitiva linguistica gli execution monitor

        - portiamo quindi un meccanismo pensato per la sicurezza dentro al linguaggio di programmazione
    - **e poi ne sviluppiamo l'interprete** (estendendo quello di MicroFUN)

- Requisiti dell'EM *as language primitive*
    - deve poter accedere ai runtime data riguardanti gli *eventi* rilevanti per la sicurezza
        - *qual e' la prossima istruzione che sta per essere eseguita?*
    - deve poter terminare l'esecuzione (in caso di attacco individuato)
        - o comunque bloccare tutto e portare il programma in un buon stato (tipo una richiesta di conferma da parte dell'utente)
    - deve proteggere lo stato del monitor e il codice da eventuali manomissioni
        - il compilato deve entrare a far parte della TCB del runtime support del linguaggio (NX Memory?..)
    - deve avere poco overhead

- **FunEM (**micro linguaggio funzionale con EM come primitiva)
    - come sempre a noi arriva direttamente **l'AST**

```
type ide = string
type exp = Eint of int
  | Den of ide
  | Sum of exp*exp
  | Times of exp * exp
  | Minus of exp * exp;;
```

        - **runtime structure**

```
let emptyenv = []
(* the empty environment *)

let rec lookup env x =
  match env with
     | []         -> failwith ("not found")
     | (y, v)::r -> if x = y then v else lookup r x

let bind env (x:string) (v:int) = (x,v)::env;;
```

- **Controllo degli Accessi (embedded nel linguaggio!)**
  - l'operazione (aritmetica) che voglio fare e' permessa?

```
type acl = Empty
         | AC of string * acl
(* Access control lists *)

let rec emCheck alist op =
  match alist with
    | Empty     -> false
    | AC(aop, als)-> if op = aop then true else emCheck als op;
```

  - emCheck
    - prende come parametri
      - alist: lista del controllo degli accessi
      - op: operazione che si vorrebbe eseguire
    - scorre ricorsivamente la lista fino a trovare l'autorizzazione per l'operazione passata
      - se non la trova ritorna false e l'operazione e' dichiarata non ammissibile
  - la ACL non deve essere corrotta dagli attaccanti, deve cioè' far parte della TCB del linguaggio
    - dove la salvo a runtime? in una memoria non eseguibile

- **Definiamo l'interprete**
  - **valutazione di una variabile**
    - semantica

$$\frac{env(x) = v}{env\ acl \rhd Den\ x \Rightarrow v}$$

    - implementazione

```
eval Den x, env, al -> lookup x env
```

  - **valutazione di una operazione (aritmetica: in questo caso +)**
    - semantica

$$\frac{Sum \in al\ env \rhd e_1 \Rightarrow v_1, env \rhd e_2 \Rightarrow v_2}{env\ acl \rhd Sum(e_1, e_2) \Rightarrow v_1 + v_2}$$

    - implementazione

```
eval Sum(e1, e2) env al ->
    if (emCheck alist "add")
       then (eval e1 env al) + (eval e2 env al)
       else failwith("Sum not allowed")
```

      - *qui si vede quanto il nostro linguaggio sia eager*

  - il codice complessivo e' di fatto derivato dai tre casi sopra

```
let rec eval exp env (alist:acl) = match exp with
  | Eint(n) -> n
  | Den x -> lookup env x
  | Sum(e1,e2) ->  if (emCheck alist "add")
       then (eval e1 env alist) + (eval e2 env alist)
       else failwith("Sum not allowed")
  | Times(e1,e2) -> if (emCheck alist "prod")
       then  (eval e1 env alist) * (eval e2 env alist)
       else failwith("times not allowed")
  | Minus(e1,e2) -> if (emCheck alist  "sub")
       then (eval e1 env alist) - (eval e2 env alist)
       else failwith("Minus not allowed");;
```

- a questo punto facciamo un passo ulteriore
  - **introduciamo l'execution monitor!**
    - non che va a monitorare tutto il programma ma un particolare frammento di codice
    - possiamo quindi definire e forzare politiche di sicurezza su frammenti di codice
    - **in modo composizionale possiamo usare frammenti di codici che consideriamo essere sicuri in quanto ognuno avrà il suo custom EM che forza la policy adeguata**

  - `letEM x = e1 with al in e2`
    - introduciamo (con questa notazione astratta alla ocaml) un execution monitor che va a inserire una politica di sicurezza (definita dalla access list al) che va rispettata nella valutazione di e1 ed e2
    - e' una estensione del let: ora gestisce la politica di sicurezza
      - **stiamo aggiungendo una politica *locale* per la valutazione di e2 (body del let)**

- **ora però dobbiamo estendere il linguaggio delle espressioni aritmetiche con l'execution monitor locale**

```
type ide = string
type iexp =
  | Eint of int
  | Den of ide
  | Sum of iexp * iexp
  | Times of iexp * iexp
  | Minus of iexp * iexp
  | LetEM of ide * iexp * acl * iexp;;
```

- iexp sta per “inlined expression” a sottolineare la presenza dell'em embedded
  - `letEM x = e1 with al in e2`
    - x: ide
    - e1: iex
    - al: acl
    - e2: iexp
      - il codice che vogliamo monitorare con la nuova politica

- **ora introduciamo politiche locali con il letEM**
  - dobbiamo quindi estendere anche l'access control list e gestire questa dualità tra locale e “globale”

```
let rec extend al1 al2 =
  match al2 with
    | Empty -> al1
    | AC(aop, als) -> AC(aop, (extend al1 als));;
```

- al2 e' la lista locale
- concateno al2 in testa
  - potrei quindi “overrideare” dei permessi (fico!)

- **estendiamo ora l'interprete**
  - tutto come prima tranne la **gestione dell'EM**
    - **semantica**

$$\frac{env\ acl[al] \rhd e1 \Rightarrow v1 \quad anv[x = v1], acl \rhd e2 \Rightarrow v}{env, acl \rhd LetEM\ x = e1\ with\ al\ in\ e2 \Rightarrow v}$$

- dove **acl** e' la nuova lista degli accessi estesa (al_acl)

- **implementazione**

```
LetEM(i,e1,al,e2) ->
   let newal = (extend alist al) in
       let v = (eval e1 env newal) in
              (eval e2 (bind env i v) newal)
```

- **interprete esteso**

```
let rec eval iexp env (alist:acl) = match iexp with
  | Eint(n) -> n
  | Den x -> lookup env x
  | Sum(e1,e2) ->  if (emCheck alist "add")
      then (eval e1 env alist) + (eval e2 env alist)
      else failwith("Sum not allowed")
  | Times(e1,e2) -> if (emCheck alist "prod")
     then  (eval e1 env alist) * (eval e2 env alist)
     else failwith("times not allowed")
  | Minus(e1,e2) -> if (emCheck alist  "sub")
     then (eval e1 env alist) - (eval e2 env alist)
     else failwith("Minus not allowed")
  | LetEM(i,e1,al,e2) -> let newal = (extend alist al) in
                            let v = (eval e1 env newal)
                              in (eval e2 (bind env i v) newal)
```

- **NB:** simle ACL (come quello che abbiamo appena fatto) sono rappresentabili come automi (banali)

- **gli EM sono molto più complessi di così**
  - eseguono computazioni arbitrare per decidere se permettere o bloccare un dato evento
  - possono avere side effects
  - possono cambiare il control flow del programma

- **NB: step per la progettazione dell'interprete (post frontend)**
  - **definizione della struttura sintattica (AST)**
    - **type expr = ...**
    - **definizione dei valori "a runtime"**
      - **INT, Closure, ...**
  - **definizione del runtime**
    - **env**
    - **acl**
    - **...**
  - **implementazione eval (post regole di inferenza (semantica))**

- estendere il linguaggio visto sopra in modo tale da permettere la definizione e la chiamata di funzione
  - con acl: una funzione può essere ammessa o meno tramite permessi (acl) locali (letEM) o "globali"

_

Parentesi su IRM

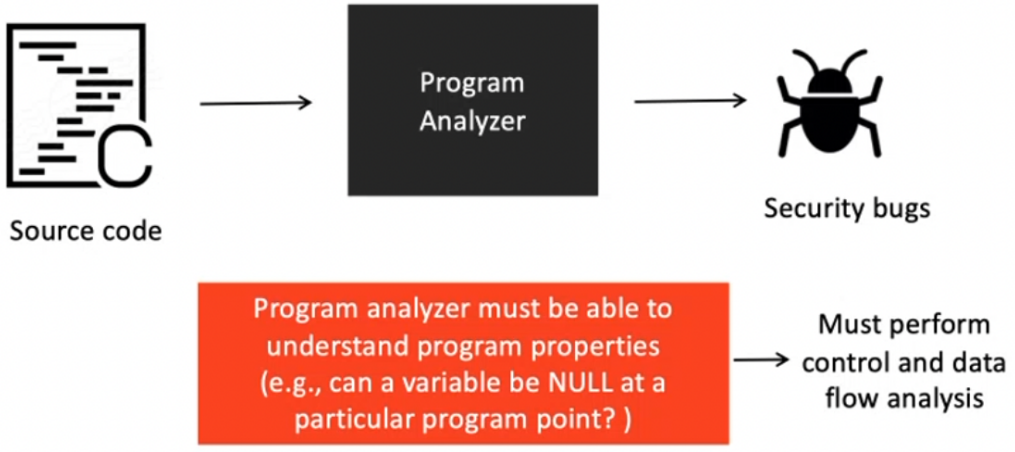
- abbiamo visto come instrumentare del codice (relativo ad una politica di sicurezza) nel binario del programma untrusted, così da renderlo trusted
  - questo e' l'esempio di codice instrumentato
    - il dev che scrive il programma untrusted usa un linguaggio che poi viene compilato in un linguaggio target
      - in realtà il runtime usa un altro programma (cioè' esegue un altro codice rispetto a quello target), dovuto all inlining degli automi
    - noi qui vediamo un esempio analogo
      - mi arriva un codice da interpretare (post frontend)
        - il nostro "binario"

            - eseguo un passo di compilazione intermedia in cui faccio l inlining degli automi per l'enforcement della politica di sicurezza
                - praticamente l'eval di expr costruisce iexpr (intermediate expr), che e' arricchito con le informazioni necessarie all enforcement delle politiche di sicurezza
                    - “genero quindi un altro binario”
            - interpreto il nuovo codice iexpr con ieval
    - l'esempio e' [qui]

_

- gli EM sono ottimi
    - **cosa facciamo dopo?**

- **Behaviour Detection**
    - idea: osservare il comportamento del programma untrusted
        - sta facendo quello che dovrebbe?
    - problemi
        - come definire “il comportamento atteso”?
        - come individuare (efficientemente)le deviazioni dal comportamento atteso
        - come evitare compromissioni del detector?
    - approccio



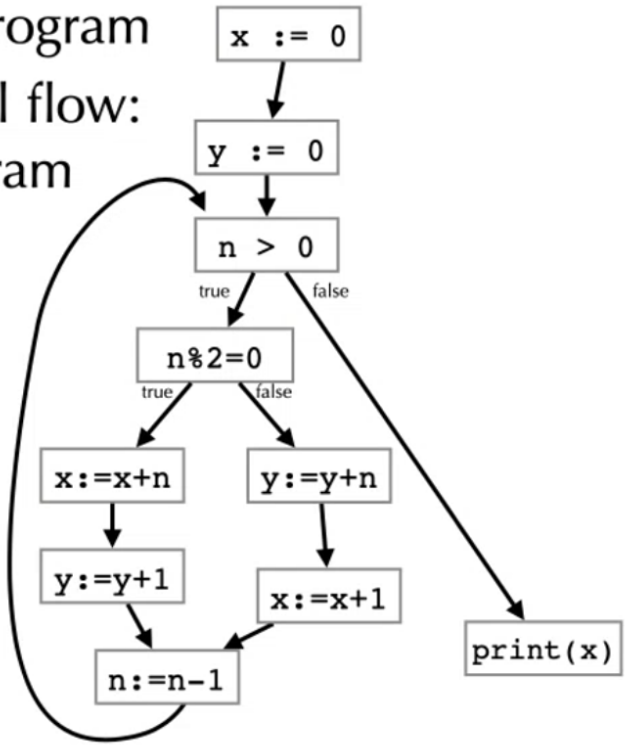
    - **backend del compiler**
        - includono anche la behaviour detection
            - LLVM: compilatore state of the art che fa un sacco di analisi statiche fiche

- **Control Flow Integrity** (behaviour detection & static analysis)
  - **idea**: cerchiamo di costruire il **cfg** (control flow graph) tramite
    - analisi statica sul sorgente
    - analisi statica sul binario
      - si fa quindi un execution profiling
        - profilazione delle possibili esecuzioni del programma
        - se l'esecuzione a runtime diverge ci sono problemi
  - definiamo delle politiche di sicurezza
    - se le esecuzioni (che comunque rispettano il profiling) violano le politiche ci sono dei problemi

- **Control Flow Graph**

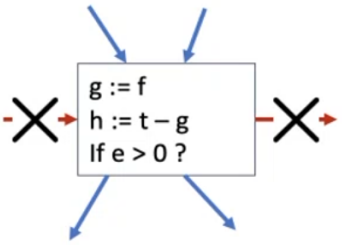
- Graphical representation of a program
- Edges in graph represent control flow: how execution traverses a program
- Nodes represent statements

```
x := 0;
y := 0;
while (n > 0) {
  if (n % 2 = 0) {
    x := x + n;
    y := y + 1;
  }
  else {
    y := y + n;
    x := x + 1;
 }
  n := n - 1;
}
print(x);
```



  - rappresentare **ogni** istruzione con un nodo rende il tutto troppo pesante e c'e' un'esplosione del numero dei nodi
    - **i nodi del cfg sono i basic block**
      - sequenza di istruzioni con un ingresso ed un'uscita specifica
        - non puoi saltare fuori dal blocco per vie traverse

- A **basic block** is a sequence of straight line code that can be entered only at the beginning and exited only at the end



Building Basic Blocks

identify leader:

The first instruction in a procedure, or
The target of any branch, or
An instruction immediately following a branch (implicit target)

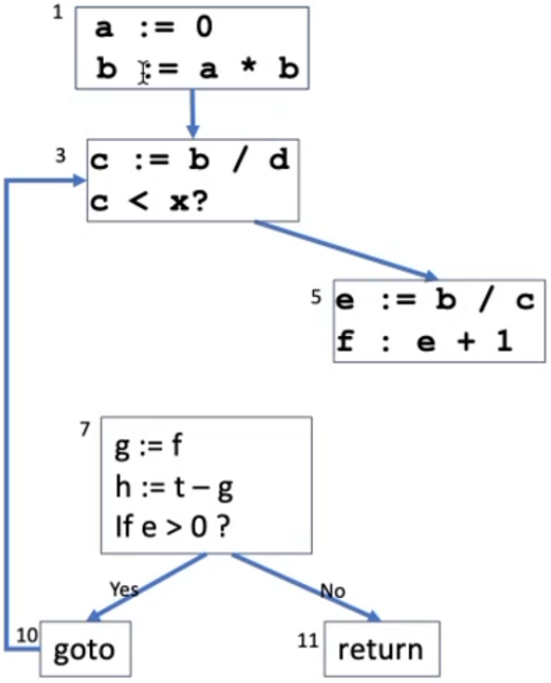
– Group all subsequent instructions until the next leader

- un leader può essere
  - la prima istruzione del blocco
  - un branch di un condizionale (il blocco inizia dal ramo then di un if)
  - ...

Building a CFG

Construction

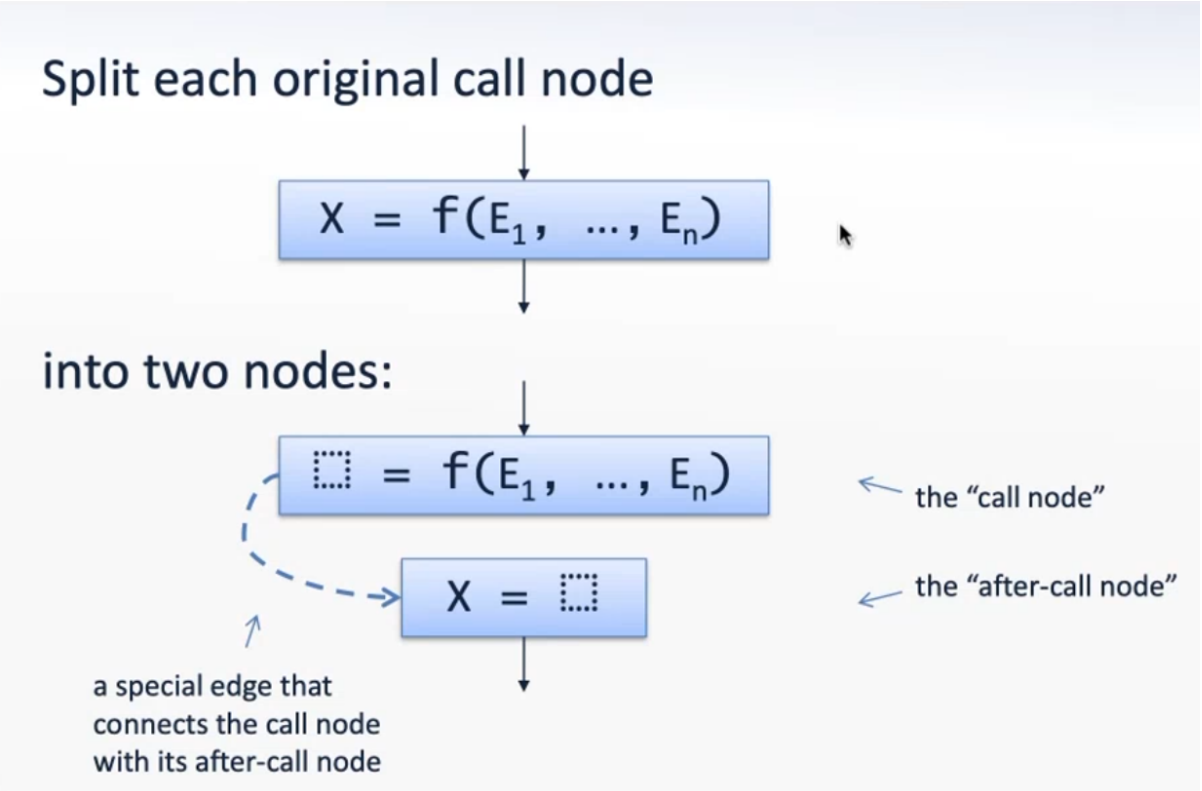
–Each CFG node represents a basic block
–There is an edge from node i to j if
  –Last statement of block i branches to the first statement of j

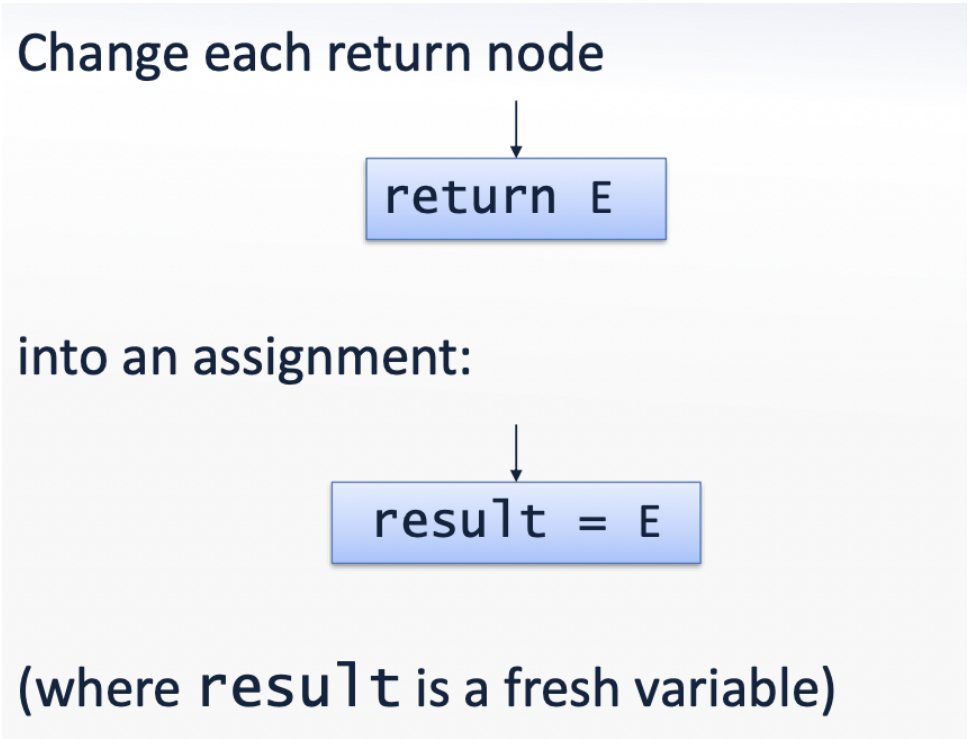


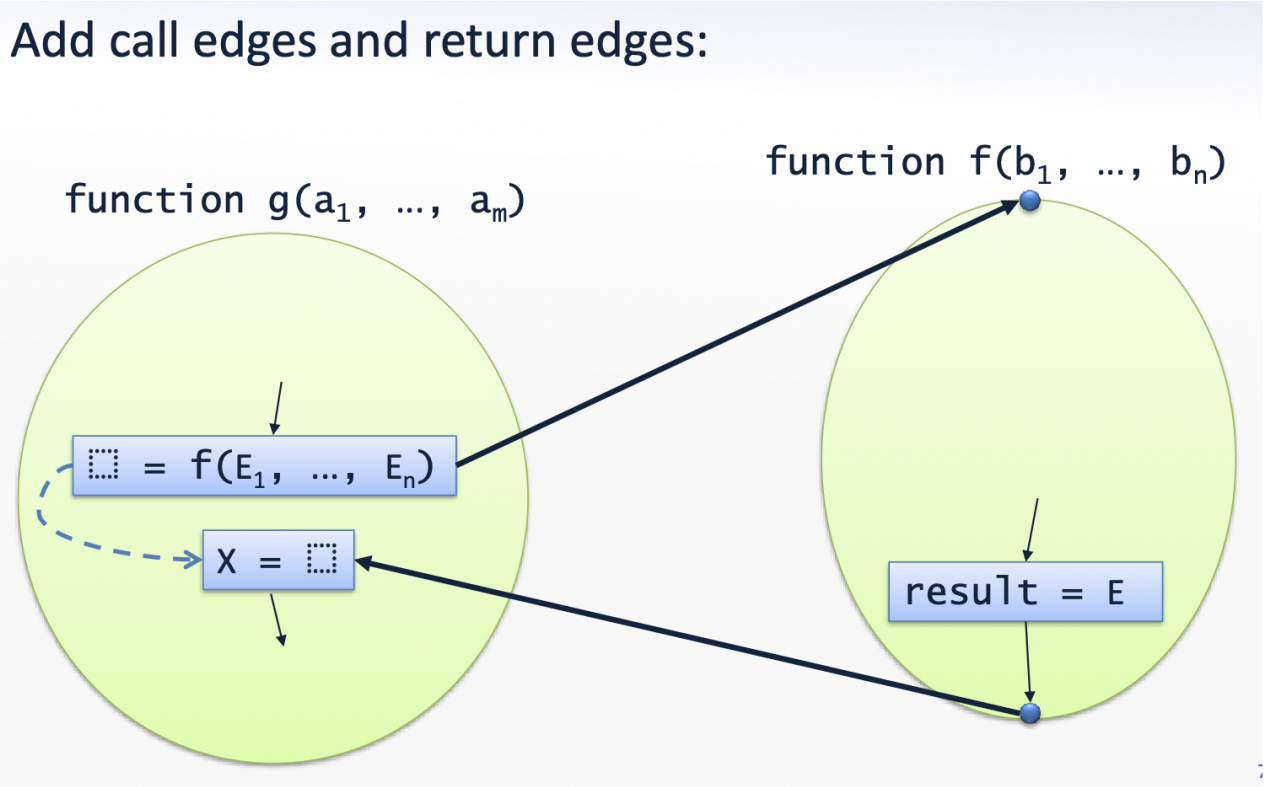
- **due tipi di analisi**
  - **intraprocedural analysis** (come da esempio sopra)
    - analizzare la singola funzione

- **interprocedural analysis**
  - analizzare l'intero programma e tenere conto delle chiamate di funzione

- **Intra-Procedural Analysis**
  - costruisci un cfg per ogni funzione
  - “assemblare” i cfg per capire le chiamate inter-funzioni ei loro retur







- **Control Flow Integrity (CFI)**
  - restringe il flusso di controllo di un programma all'insieme delle tracce valide (permesse)
    - questo viene fatto confrontando le possibili tracce permesse (cfg) dedotte a compile-time con l'esecuzione reale a runtime
  - **come viene fatto realmente l'enforcement tramite CFI?**
    - per ogni trasferimento di controllo (eg chiamata di funzione) determina staticamente le possibili destinazioni
    - inserisci a runtime una bit label unica per ogni destinazione
    - inserisci codice binario a runtime che controllerà se la bit label (pattern) dell'istruzione matcha il pattern delle possibili destinazioni
  - riassumendo: **ingredienti della CFI**
    - definire il comportamento del programma untrusted
      - build del cfg
    - individuare deviazioni dai comportamenti permessi in modo efficiente
      - IRM (inline reference monitor)
    - evitare la manomissione del detector
      - randomness nella generazione delle bit label ad ogni esecuzione
        - l'attaccante non può prefabbricare label che permettono il jump verso codice malevolo

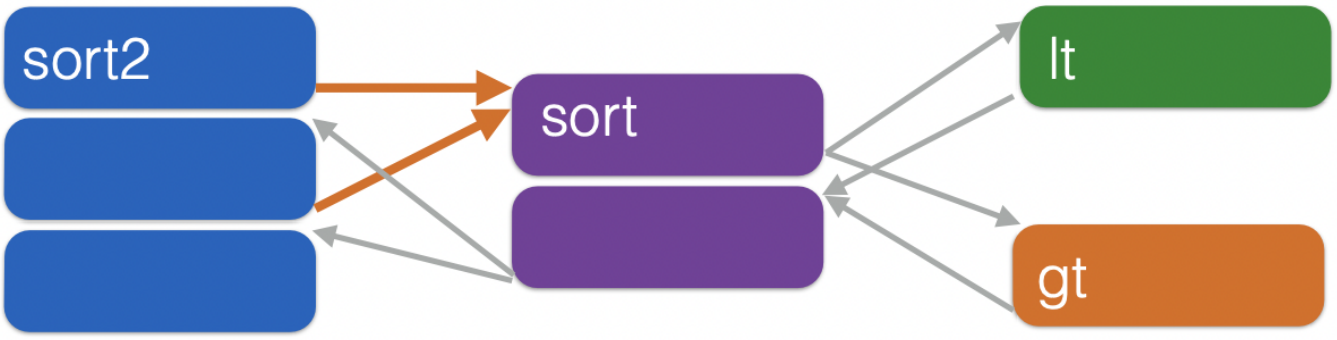
CFI: The steps of the algorithm

Hypothesis:
a) the code is immutable,
b) the target address cannot be changed

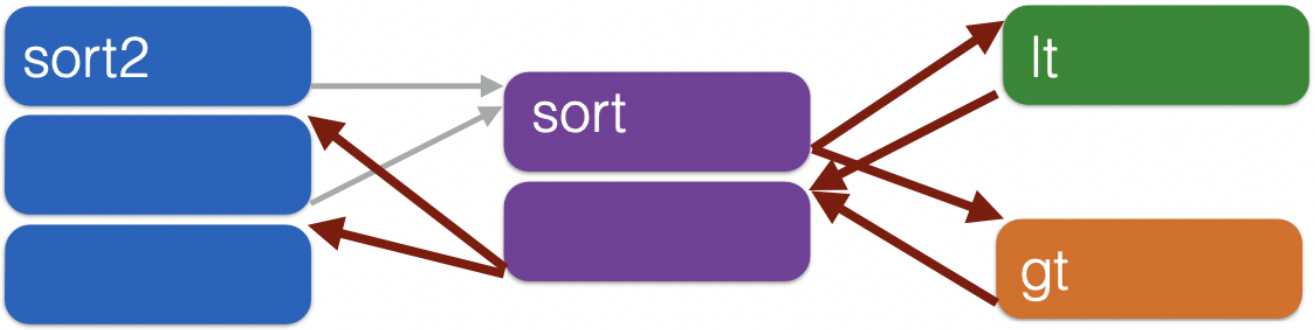
1. **Compute the CFG** in advance
   - During compilation (ahead of time), or from the binary
2. **Monitor the control flow** of the program to ensure that it only follows paths allowed by the CFG
   - **Direct calls** need **not** be **monitored (why?)**
3. **Monitor only indirect calls**
   - **jmp, call, ret with non-constant targets**

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(a, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```



Direct calls *(always the same target)*



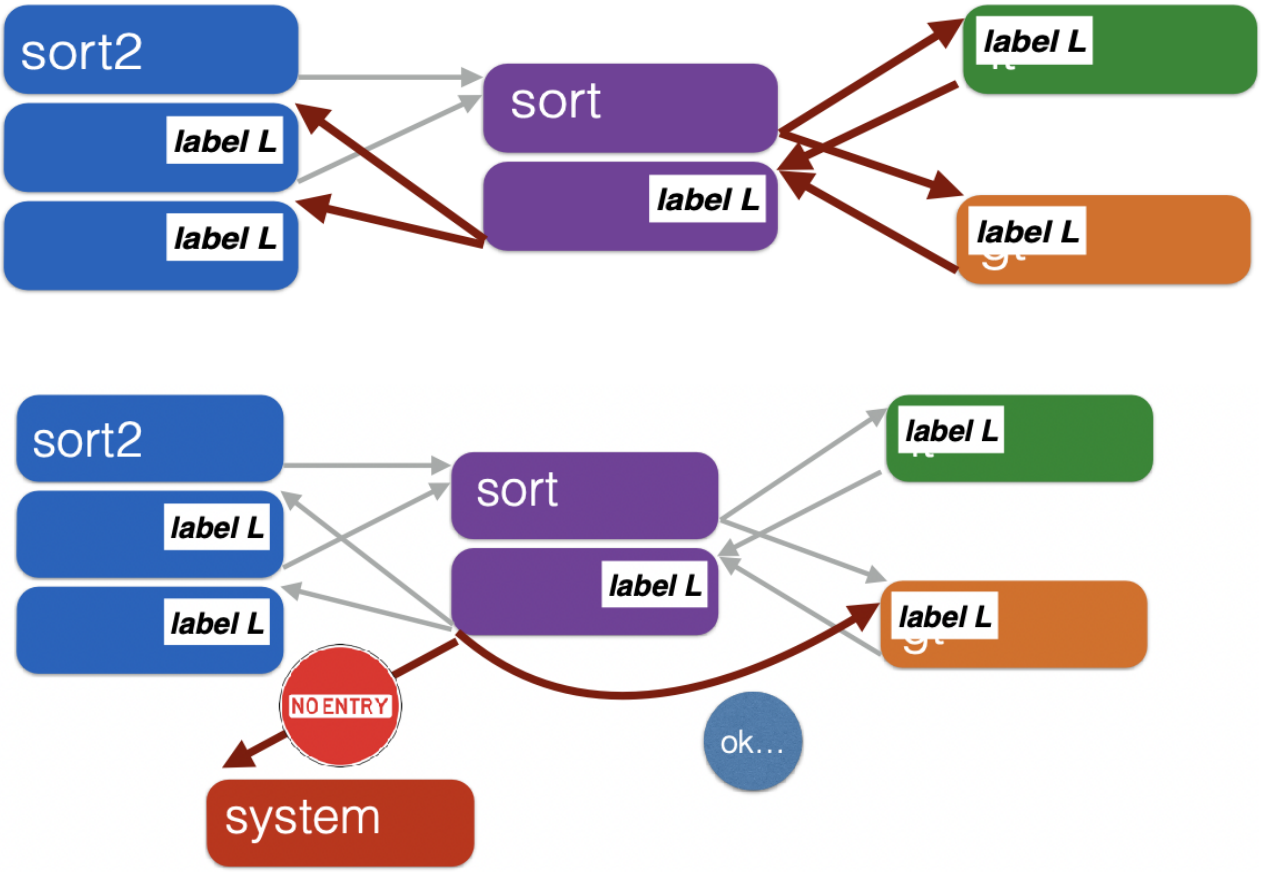
Indirect transfer (`call` *via register, or* `ret`)

- le direct call sono chiamate esplicite il cui registro può essere calcolato dal compilatore in modo statico
  - le chiamate dirette hanno sempre lo stesso target
  - **il compilatore fa parte della TCB** quegli indirizzi inaccessibili da parte dell'attaccante e quindi non vanno controllati
    - **gli indirizzi sono immutabili**
- gli indirect transfer sono trasferimenti la cui chiamata e il cui ritorno dipendono dai parametri
  - return x<y (potrebbe non tornare, dipende da x)

- **come forzare la sicurezza controllando solo gli indirect transfer**

IRM(Program Trasformation)	1. Insert a label just before the target address of an indirect transfer 2. Insert code to check the label of the target at each indirect transfer 3. Abort if the label does not match Labels are determined by the CFG

- **Usare sempre le stesse (bit) lable non e' sicuro**

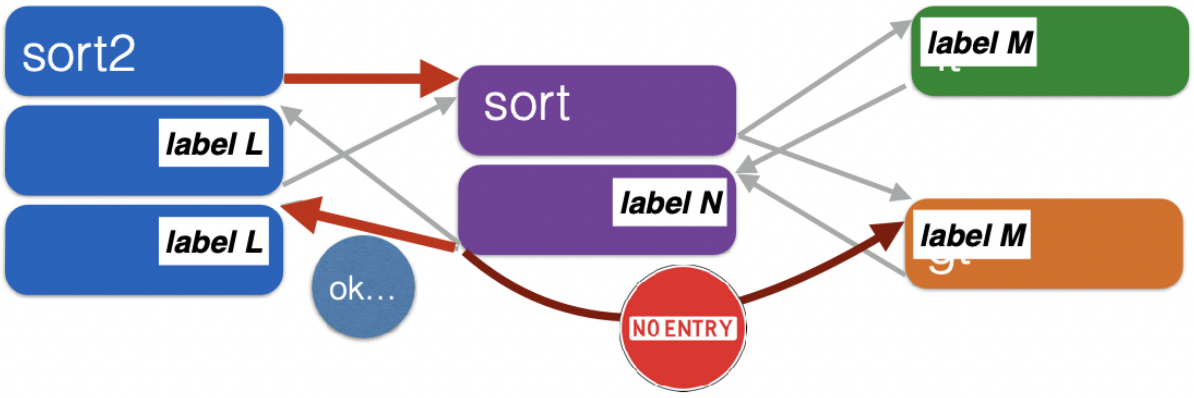


Use the same label at all targets

Blocks return to the start of direct-only call targets but not incorrect ones

- `mimcry attack!`

- `etichette distinte in base alla chiamata`



Constraints:

- return sites from calls to **`sort`** must share a label (*L*)
- call targets **`gt`** and **`lt`** must share a label (*M*)
- remaining label unconstrained (*N*)

- **QUINDI con CFI Analysis**
  - Unique IDs
    - Bit patterns chosen as destination IDs must not appear anywhere else in the code memory except ID checks
  - Non-writable code
    - Program should not modify code memory at runtime
  - Non-executable data
    - Program should not execute data as if it were code
  - Enforcement: **hardware support + static analysis + prohibit system calls that change protection state**

- **bad news**
  - e' comunque possibile effettuare dei control flow attack facendo chiamate che il cfg permette ma che hanno effetti dannosi
    - mimcry attack
      - **(soluzione: shadow call stack)**
  - data-only-attacks non coperti
  - argomenti malevoli (eg: system call)
    - filename rewrite

–

- **Safety vs Security**
  - **safety:** proteggere il sistema da fallimenti accidentali

- **security:** proteggere il sistema da attacchi attivi
  - tendenzialmente cio' che e' buono per la safety e' buono anche per la security

- **Safe Programming Language (again)**
  - **Safety (#1) for us**

  - A programming language can be considered safe if one can **trust** the **abstractions** provided by the programming language
  - The programming language **enforces** these abstractions and **guarantees** that they **cannot be broken**
    - boolean is either true or false, and never 23 or null
    - Programmer doesn't have to care if true is represented as 0x00 and false as 0xFF or vice versa

- There are many dimensions of safety
  - memory-safety, type-safety, thread-safety, arithmetic safety; guarantees about (non)nullness, about immutability, about the absence of aliasing,...
- For some dimensions, there can be many levels of safety

  - **eg: livelli di safety per il buffer overflow**
    - con "safe" qui intendiamo che non ci sono effetti negativi per la sicurezza
      - no info leak
      - no codice malevolo
      - ..

1. let an attacker inject arbitrary code
2. *possibly* crash the program (or else corrupt some data)

'unsafe'; some undefined semactics

3. *definitely* crash the program
4. throw an exception, which the program can catch to handle the issue gracefully
5. be ruled out at compile-time

'safe'

- **Nozione dei Tipi & Type Systems**

- Types assert **invariant properties** of program element
  - Variable **x** will always hold an **integer**
  - Method **m** will always return an object of class **A** (or one of its subclasses)
  - The array **arg** will never store more than 10 items
- **Type is a property of program constructs**

- **Type Systems**
  - un type system e' una collezione di regole che assegnano tipi ai costrutti del linguaggio
    - aggiungono constraints per la validità' (semantica) di un programma
      - violarli causa errori **che possono essere prevenuti e identificati a static time (type checker nel compiler)**
  - definisce quali operazioni sono possibili per ogni tipo definito
  - fornisce una formalizzazione delle regole (semantica)
  - le type rule sono definite appositamente sulla struttura delle espressioni (expr)
    - le regole sono specifiche da linguaggio a linguaggio

- **Type Checking vs Type Inference**
  - **type checking:** processo di verifica del rispetto dei tipi in programmi full-typed
    - può essere fatta a static rime, a runtime o in maniera mista
  - **type inference:** processo di derivazione dei tipi di un determinato "oggetti"
    - tipo ocaml

- **type soundness (aka type safety or strong typing)**
  - le asserzioni (i predicati che si basano sulla correttezza di tipo) in fase di implementazione sono mantenuti anche a runtime

- **controllo dei tipi in fase di analisi statica**
  - definire un type checker e' di fatto come definire un interprete di un linguaggio dove i valori che vengono calcolati dall'interprete sono i tipi

- **Tipi Statici vs Tipi Dinamici**

- The dynamic type of an object is the class C that is used in the "new C" expression that creates the object
  - A run-time notion
  - Even languages that are not statically typed have the notion of dynamic type
- The static type of an expression is a notation that captures all possible dynamic types the expression could take
  - A compile-time notion

- **Type Safety**
  - i linguaggi type-safe garantiscono che un programma che passa la fase di type-checking manipolano dati in maniera corretta in relazione al loro tipo
    - eg: non fanno somme tra bool
  - un linguaggio type safe esclude a priori un sacco di errori (e vulnerabilità) a runtime

- **Programming languages can be**
  - **memory-safe, typed, and type sound**:
    - **Java, C#, Rust, Go**
      - **though some of these have loopholes to allow unsafety**
    - **Functional languages such as Haskell, ML, OCaml, F#**
  - **memory-safe and untyped**
    - **LISP, Prolog, many interpreted languages**
  - **memory-unsafe, typed, and type-unsafe**
    - **C,C++**
      - **Not type sound: using pointer arithmetic in C, you can break any guarantees the type system could possibly make**

- **NB:** e' grigia la definizione dei modificatori di accesso (private, public, ...) che "vengono trattati" come "estensione del tipo"
  - se modifichi (in C++) un private int stai rompendo la type soundness

- **Avoiding Buffer OverFlows in Java**

- Language-based solution: Buffer overflow is ruled out at **language-level,** by combination
  - compile-time typechecking (static checks)
  - at load-time, by bytecode verifier (bcv)
  - runtime checks (dynamic checks)

    - **JAVA example**

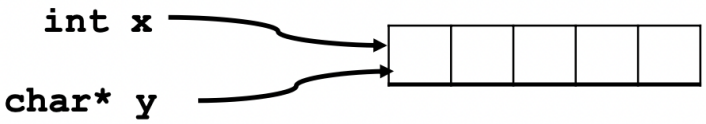
```
public class A extends Super{
  protected int[] d;
  private A next;

  public A() { d = new int[3]; }
  public void  m(int j) { d[0] = j; }
  public setNext(Object s)
        next = (A)s;
  }
}
```

runtime checks for 1) non-nullness of d, and 2) array bound

runtime check for type (down)cast

- **NB:** la type safety e' una proprieta' molto delicata, basta pochissimo per comprometterla
  - **esempio: type confusion attack**

    Data values and objects are just blobs of memory. If we can create type confusion, by having two references with different types pointing the same blob of memory, then *all* type guarantees are gone.



    - **type confusion in JAVA**

```
public class A{
    public Object x;
     ...
}
```

```
public class A{
  public int x;
   ...
}
public class B{
 void setX(A a) {
  a.x = 12;
 }
}
```

What if we could compile B against A but we run it against A?

We can do pointer arithmetic again!

If Java Virtual Machine would allow such so-called *binary incompatible* classes to be loaded, the whole type system would break.

- richer data types
  - distingui il dato che "può' essere null" da quelli "sicuro non null"

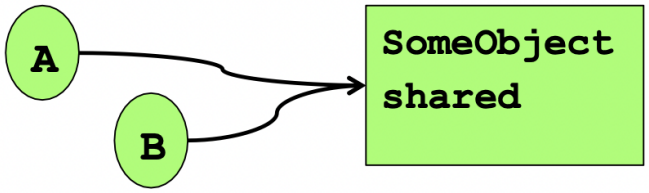
- Language based Guarantees
  - **visibility**: public, private, …
    - private fields not accessible from outside a class
  - **immutability**
    - In Java: final int i = 5;
    - in C(++) : const int BUF_SIZE = 128;
    - In Java String objects are immutable
  - Scala, Rust provides a more systematic distinction between mutable and immutable data to promote the use of immutable data structures

- THREAD SAFETY & ALIASING (&RACE)

- A program contains a **data race** if **two execution threads simultaneously access the same variable and at least one of these accesses is a write**
  - data races are highly non-deterministic, and a pain to debug!
- **thread-safety** = **the behaviour of a program consisting of several threads can be understood as an interleaving of those threads**
- In Java, the semantics of a program with data races is effectively undefined:
- **Moral of the story: Even "safe" programming languages can have very weird behaviour in presence of concurrency**
  - **thread-safety:** il comportamento del programma rimane corretto (sicuro) anche in presenza di più thread che vengono eseguiti in parallelo

- **ALIASING**

Aliasing: two threads or objects A and B both have a reference to the same object `shared`



- **Aliasing ISSUES**
  - Concurrent (multi-threaded) context: data races
    - Locking objects (synchronized methods in Java) can help… expensive & risk of deadlock & lock-free data structure
  - Single-threaded context: dangling pointers
  - Single-threaded context: broken assumptions
    - If A changes the shared object, this may break B's code, because B's assumptions about shared are broken

- **Conclusioni (Temporanee)**

- Some important programming language features:
  - type safety: the actual (runtime) type **matches** with the expected one
  - memory operations are compatible with the source-level abstraction (may forbid the use of un-initialized variables)
- memory safety: no **unintended/invalid** memory access
- thread safety: no **unintended** operations between threads
  - no *race conditions*, safe synchronization facilities, etc.
- no undefined behaviors (~ "time bombs")
  - no need for the compiler to detect or mitigate them !
  - aggressive optimizations, able to suppress security checks !

- **Build secure compilation chains**
  - **Compiler (static analyses)**
    - control-flow integrity: preserves intended control-flow method call/returns (e.g, Java), valid paths in the control-flow graph
  - **Runtime**
    - Secure interoperability with lower-level code
    - abstract machine with **build-in security components**

- **APPLICATION LEVEL SANDBOXING**
  - **compartimentalizzazione**
    - **isolation:** combinazione di **confidentiality** e **integrity** di dati e codice
      - proprietà di sicurezza fondamentale inter-processo
  - **problemi**
    - qual e' la trusted computer base?
      - compartimentalizzare funzionalità critiche dentro un processo trusted riduce le funzionalità della TCB ma estende la stessa con il meccanismo di compartimentalizzazione
    - la compartimentalizzazione può essere controllata da security policy
      - quanto sono complesse e costose queste policy?
    - cosa riguardo input e output
      - vogliamo le interfacce di interazione (tipo java nio) altrettanto semplici ed efficienti (anche se compartimentalizzate)

Compartments may provide access control mechanisms that can be configured.

This involves:

1. **Rights/permissions**
2. **Parties** (eg. users, processes, components)
3. **Policies** that give rights to parties
   - specifying who is allowed to do what
4. **Runtime monitoring** to enforce policies,
   - which becomes part of the TCB

- **Secure Software Engineering**
  1. **Divide systems into chunks** – aka compartments
     - Different compartments for different tasks
  2. **Give minimal access rights to each compartment**
     - principle of least privilege
  3. **Have strong encapsulation between compartments**
     - flaw in one compartment cannot corrupt others
  4. **Have clear and simple interfaces between compartments**
     - expose minimal functionality

- **Safe Programming Languages & Access Control**
  - in un linguaggio di programmazione sicuro l'access control può essere fornito a language-level, dato che le interazioni tra componenti possono essere limitate e controllate dalle astrazioni del linguaggio stesso

AC at the language level makes it possible to have security guarantees in the presence of untrusted code (which could be malicious or just buggy)

1. Without **memory-safety**, this is impossible. Why?
   - Because B can access any memory used by A
2. Without **type-safety**, it is hard. Why?
   - Because B can pass ill-typed arguments to A's interface

- **CODE BASED ACCESS CONTROL**
  - tramite l'access control (fatto come ti pare, anche language based) riesci ad ottenere la compartimentalizzazione
    - limiti le azioni che il programma può fare
      - tendenzialmente con il language based riesci a farlo senza i problemi della compartimentalizzazione visti prima
  - parti diverse del programma sono trattate differentemente
    - in base all access control di quella porzione
  - access control (code based) lavora on top dell user-based access control dell'OS
  - **ingredienti per l'access control (code based e non)**
    - **permessi**

1. **Permissions are Java Objects**
   1. **permissions** are an instance of the **AllPermission class**,
2. **Permissions represent a right to perform some actions.**

Examples:

- FilePermission(name, mode)
- NetworkPermission
- WindowPermission
- Permissions have a **set semantics**, so one permission can be a superset of another one.
  - FilePermission("*", "read") includes FilePermission("some_file.txt", "read")
- Developers can program and define new custom permissions.
  - **i permessi** hanno un ruolo nel **management della method invocation**

There are different possibilities to manage at run-time method invocation.

- Take #1: allow action if top frame (aka activation record) on the stack has permission for that action
- Take #2: only allow action if all frames (activation records) on the stack have permission

- Take #1: The solution is very dangerous: a class may accidentally expose dangerous functionality
- Take #2: The solution is very restrictive: a class may want to, and need to, expose some dangerous functionality, but in a controlled way
- We need a ore flexible solution: **stack walking** also known as **stack inspection**

- `protection domains`
  1. **Where did it come from?**
     - **where on the local file system (hard disk) or where on the internet**
  2. **Was it digitally signed and if so by who?**
     - **using a standard PKI**
- `policies`

- **STACK INSPECTION**

Every resource access or sensitive operation is protected by aguard call (**demandPermission(P))** for an appropriate permission P

Constraint: No access without permission!

The algorithm for granting permission is based on stack inspection (also called stack walking)

- **la stack inspection ha come ipotesi la primitiva linguistica che permette l'overriding dei permessi da parte del programmatore, che si prende la responsabilità "di dire al runtime" di eseguire operazioni privilegiate**
  - h chiama g che chiama f
    - in f posso voler fare operazioni per le quali heg non sono autorizzate

        - io programmatore posso dire al runtime che f può fare la roba e che me ne prendo la responsabilità
        - altrimenti se heg non hanno i permessi si fallisce

- **Programming Stack Inspection**
    - gli stack frames sono annotati con i loro protection domains e con i permessi a loro garantiti
    - durante l'ispezione gli stack frame sono cercati a partire dal più al meno recente (ultimo chiamato verso i chiamanti)
        - **fail** se un frame ispezionato non ha i permessi adeguati per l'operazione *corrente*
        - **succeed**
            - se tutti gli stack frame hanno il permesso di eseguire l'operazione
            - se il programmatore si e' preso la responsabilita' di eseguire operazioni privilegiate

Enable_permission(P)

means: don't check my callers for this permission, I take full responsibility

This is essential to allow controlled access to resources for less trusted code

Disable_permission(P)

means: don't grant me this permission, I don't need it

This allows applying the principle of real privilege (ie. only giie or ask the privileges really needed, and only when they are really needed)

- **algoritmo per la stack inspection**

DemandPermission(P) algorithm:

1. for each caller on the stack, from top to bottom: if the caller
    a) lacks Permission P: throw exception
    b) has disabled Permission P: throw exception
    c) has enabled Permission P: return
2. check inherited permissions

- Java Security Guarantees

1. **memory safety**
2. **strong typing**
3. **visibility restrictions**
4. **immutable fields**
5. **unextendable classes**
6. **immutable objects, eg String, Boolean, Integer, URL**
7. **sandboxing based on stackwalking**

This allows security guarantees to be made even if part of the code is untrusted – or simply buggy

- `TCB for JAVA`
  - **Byte Code Verifier** (BCV) typechecks the byte code
  - **Java Virtual Machine** (JVM) executes the byte code (with some type-checking at run time)
  - **SecurityManager** (SM) does the runtime access control by stack inspection
  - **ClassLoader** (CL) downloads additional code, invoking BCV & updating policies for the SecurityManager

  Some Java features are still the root of security problems:

  - **Large TCB with large & complex attack surface**, growing over time
    - Many classes in the core Java API are in the TCB and can be accessed by malicious code
    - Security-critical components are implemented in Java & runs on the same VM, incl. ClassLoader and SecurityManager

- `E riguardo i linguaggi di programmazione unsafe?`
  - `sandboxing non a language level`

- Unsafe languages cannot provide sandboxing at the language level … but
- An application written in an unsafe language could still use OS sandboxing by splitting the code across different processes (this is what Chrome does)
- An alternative approach:
  use sandboxing support provided by underlying hardware, to impose memory access restrictions inside a process

- **SECURE ENCLAVE**

Example: security-sensitive code in larger program

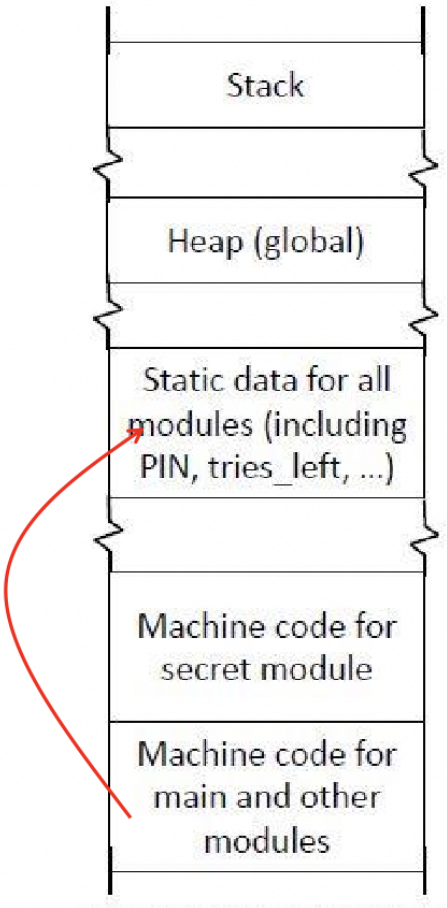
secret.c

```
static int tries_left = 3;
 static int PIN = 1234;
 static int secret = 666;

 int get_secret (int pin_guess) {
   if (tries_left > 0) {
    if ( PIN == pin_guess) {
      tries_left = 3; return secret; }
    else {
      tries_left--; return 0 ;}
 } }
```

main.c

```
# include "secret.h"
... // other modules
void main () {
...
}
```

Bugs or malicious code *anywhere* in the program could access the high-security data



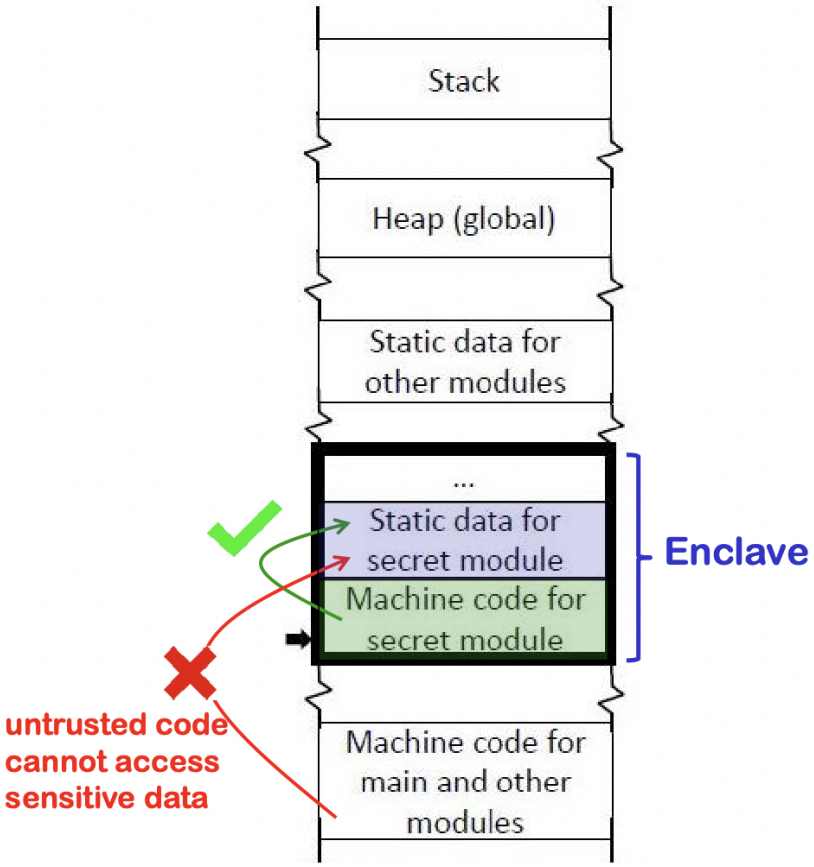
Isolating security-sensitive code with secure enclaves

secret.c

```
static int tries_left = 3;
 static int PIN = 1234;
 static int secret = 666;

 int get_secret (int pin_guess) {
  if (tries_left > 0) {
   if ( PIN == pin_guess) {
     tries_left = 3; return secret; }
   else {
     tries_left--; return 0 ;}
 } }
```

main.c

```
# include "secret.h"
... // other modules
void main () {
...
}
```



1. Enclaves isolates part of the code together with its data
   - Code outside the enclave cannot access the enclave's data
   - Code outside the enclave can only jump to valid entry points for code inside the enclave
2. Less flexible than stack walking:
   - Code in the enclave cannot inspect the stack as the basis for security decisions
   - Not such a rich collection of permissions, and programmer cannot define his own permissions
3. More secure, because
   - OS & Java VM (Virtual Machine) are not in the TCB
   - Also some protection against physical attacks is possible
     - `l'enclave richiede supporto hw (intel sgx)`

- `Perche' guardiamo ai secure enclave (e all'hw support)`

- a noi interessa la sicurezza language-based
  - A recent wave of attacks have shown that hardware isolation mechanisms can be attacked via *software-exploitable side-channels*.
    - Over the past few years, many major isolation mechanisms have been successfully broken using side-channels
    - Foreshadow broke confidentiality of enclaved executions on Intel processors.

Recap: different forms of compartmentalisation

- **Conventional OS acccess control** } **access control *of* applications and *between* applications**

- **Language-level sandboxing in safe languages**
  - **eg Java sandboxing using stackwalking**
  - **Java VM & OS in the TCB**
- **Hardware-supported enclaves in unsafe languages**
  - **eg Intel SGX enclaves**
  - **underlying OS possibly not in the TCB**

} **access control *within* an application**

- **Language-based sandboxing is a way to do access control within a application: different access right for different parts of code**
  - This reduces the TCB for some functionality
  - This may allows us to limit code review to small part of the code
  - This allows us to run code from many sources on the same VM and don't trust all of them equally
- **Hardware-based sandboxing can also achieve this also for unsafe programming languages**
  - Much smaller TCB: OS and VM are no longer in the TCB
  - But less expressive & less flexible
  - No stackwalking or rich set of permissions

- **INFORMATION FLOW**
  - dove sono i punti in cui l'informazione fluisce?
    - **channel:** il mezzo su cui l'informazione transita
      - **legitamete channel:** canale intended per la comunicazione tra due moduli
      - **covert channel:** canale non pensato per il trasferimento di informazioni ma che può essere exploitato
      - **storage: scritto** da un modulo e **letto** da un altro
    - noi vogliamo restringere l'accesso alle informazioni a solo coloro che hanno il diritto di accedervi
      - la proprietà di **isolation** e' fondamentale
        - tutt'ora ci sono casi di attacchi molto dannosi (spectre, meltdown, ..)
        - **speculation** su processori moderni

```
try {
  i = secret;
  V = M[i]; //speculative (loaded in cache)
} catch (IllegalReadException) {
  // attacker use the handler to leak info
  for j in 0 to Max {
    read M[j];
  }
}
```

          - esempio di **covert channel:** usato un canale non previsto per la trasmissione di informazioni
            - **la cache e' il cover channel**
  - **Tracking Data Flow**
    - due metodi
      - **tainted analysis**
      - **security types for information flow**

- **TAINTED ANALYSIS**
  - tracciare il flusso di informazioni di un programma tramite delle analisi statiche
    - identificare la sorgente di taint info (informazione “infetta”)
      - tainted data
        - format string
        - SQL query malevole (sql injection)
        - ...

        - identificare la propagazione del taint
            - tracciare “le istruzioni” che manipolano i dati in modo da capire se il risultato dell'istruzione e' tainted o no
        - **se l'analisi trova che l'informazione tainted potrebbe venire usata dove ci si aspettano dati untainted viene notificata la potenziale vulnerabilità**

- **Tainted Analysis in Action**
    - sorgenti di informazioni
        - **untrusted - tainted** (public)
            - tutto cio' che c'e' di pubblico nel codice
        - **trusted - untainted** (secret)
            - tutto quello che e' segreto/inaccessible all'attaccante

- **Intuitive Idea**
    - **flusso dell'assegnamento**
        - X := Y

The flow from Y to X is legale whenever the (type of) value of variable Y (t(Y) must be compatible with the (type of) value of variablle x (t(X))

$$\frac{t(Y) < t(X)}{X := Y \text{ is ok}}$$

- con (type of) intende **tainted / untainted**

Legal Flow	Illegal Flow

`void f(tainted int);`
`untainted int a = …;`
`f(a);`
	`void g(untainted int);`
	`tainted int b = …;`
	`g(b);`

untainted < tainted	tainted ! < untainted

Allowed flow as a **lattice** untainted < tainted

        - g si aspetta un untainted: se viene rilevato che il parametro attuale di g e' infetto (tainted) allora si blocca tutto

- l'analisi e' di fatto una inferenza di tipo
    - non vuoi capire se x e' int o bool ma se e' tainted o no
    - No data qualifier is available for program variables: we must infer it.
    - The ***type inference*** algorithm takes the following steps.
        - for each type that does not have a qualifier, generate a **fresh name** α for it.
        - for each statement in the program, **generate constraints** of the form $\alpha < \beta$
        - The solution is simply a substitution of qualifiers α, β … (associating tainted or untainted tags to qualifiers) such hat **all of the constraints are satisfied**.
        - If no solution exists, than we may have spotted an **illegal flow**

- **Tainted Analysis Algorithm Example**
    - **codice da analizzare**

```
int printfun(untainted  string) { … // trusted sink};
tainted   string getsFromNetwork(….); //untrusted source


string name getsFromNetwork(….)
string x = name;
printfun(x)
```

- **step1: annotare con fresh qualifier (alfa e beta)**

```
int printfun(untainted  string) { … };
tainted   string getsFromNetwork(….);


α string name getsFromNetwork(….)
β string x = name;
printfun(x)
```

- step2: genera i constraint per ogni staintment

```
int printfun(untainted  string) { … };
tainted   string getsfromNetwork(….);
```

```
α string name getsFromNetwork(….)
β string x = name;
printfun(x)
```

tainted <= α

α <= β

β <= untainted

  - nb: x <= y si legge come x va in y
    - la freccia e' praticamente al contrario
- step3: controlla i constraints e vedi se c'e' una soluzione

```
int printfun(untainted  string) { … };
tainted   string getsfromNetwork(….);
```

```
α string name getsFromNetwork(….)
β string x = name;
printfun(x)
```

tainted = α

α <= β

β = untainted

tainted <= untainted

we have identified a potentially illegal flow, because there is no possible solution for alpha and beta.

- L'information flow si interseca con il control flow

```
int printfun(untainted  string) { … };
tainted   string getsFromNetwork(….);

α string name getsFromNetwork(….)
β  string x;
If ()       x = name
    else  x = “ciao”;
printfun(x)
```

THEN BRANCH

tainted <= α
α <= β
β <= untainted

tainted <= untainted

Constraints are unsolvable: illegal flow

```
int printfun(untainted  string) { … };
tainted   string getsFromNetwork(….);

α string name getsFromNetwork(….)
β  string x;
If ()       x = name
    else  x = “ciao”;
printfun(x)
```

ELSE BRANCH

tainted <= α
untainted <= β
β <= untainted

tainted = α
untainted = β

Constraints solvable: legal flow

- se faccio un drop del condizionale ottengo

- x = name (**tainted**)
- x = “ciao”
  - in questo caso ottengo **un falso allarme**
    - **il sistema mi dice che x e' tainted anche se in realta' x e' stata sovrascritta**

int printfun(untainted string) { … };
tainted string getsFromNetwork(….);

α string name getsFromNetwork(….)
β string x;
x = name
x = “ciao”;
printfun(x)

variable x is overridden

tainted <= α
α <= β
untainted <= β
β <= untainted

FALSE ALARM
Same constraints
Different Meaning

Constraints are unsolvable: illegal flow

- **Flow Sensitivity**
  - The analysis we developed is **Flow Insentive**
    - The qualifier of each variable ***abstracts the taintness of all values*** it ever contains
  - A flow sensitive analysis accounts for variables whose values may change
    - Each assignment has a different qualifier
      - The two assignment at x in our example would have two different qualifiers
    - Idea: static single assignment
- **STATIC SINGLE ASSIGNMENT**

- per ogni assegnamento della stessa variabile crei una variabile nuova (un versioning della singola variabile)

```
int printfun(untainted  string) { … };
tainted   string getsFromNetwork(….);

α string name getsFromNetwork(….)
β  string x1;  γ string x2
x1 = name
x2 = “ciao”;
printfun(x2)
```

tainted <= α — NO ALARM

α <= β

untainted <= γ

γ <= untainted

Constraints are solvable: γ = untainted α = β = tainted

- **Path Sensitivity**
  - The problem is that the constraints we generates do not correspond to **feasible paths** (i.e. **feasible executions**)

  - **Solution**: *We develop an analysis which considers the feasibility of paths when generating constraints*

The analysis considers execution path sensitity

```
void f (int x) {
    string y;
    (1) If (x) (2) y = ”ciao”
    else  (3) y = getsFromNetwork()
    (4) if (x) =  (5) printfun(x)
(6)}
```

Execution paths
1-2-4-5-6 when x !=0
1-3-4-6 when x = 0

Path
1-3-4-5-6 is infeasible

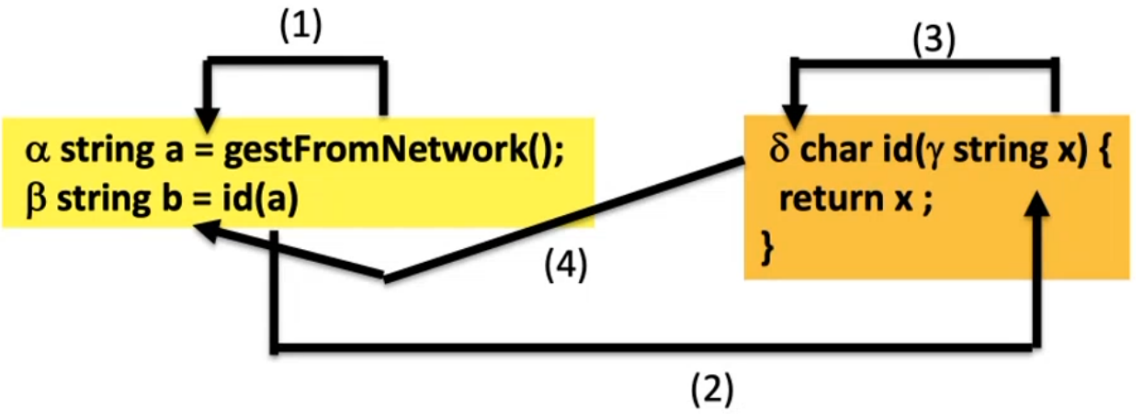
Path senstitive analysis constrains with a path condition the constraints

x!=0 => untainted <= α path segment 1-2
x = 0 => tainted <= α path segment 1-3
x!=0 => α <= untainted path segment 4-5

  - **good news:** path sensitivity rende le analisi molto più' precise
  - **bad news:** path sensitivity rende le analisi molto più complicate
    - molti più constraints
  - classico trade off **precision vs scalability**

- **Next Step: come gestire flussi tainted verso le chiamate di funzioni**

- handling function calls

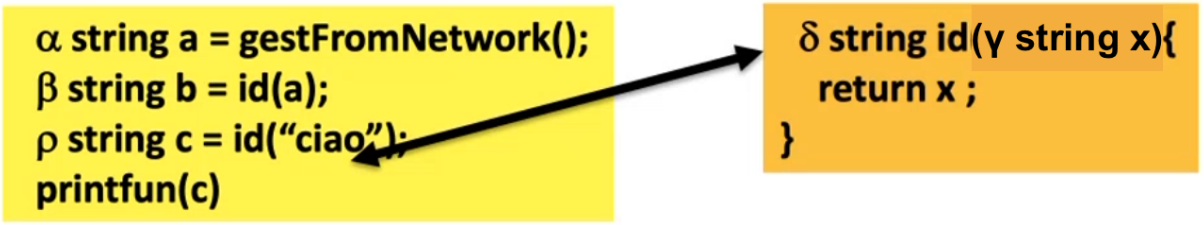


(1) tainted <= α
(2) α <= γ
(3) γ <= δ
(4) δ <= β

MIMIKING THE
CONTROL FLOW GRAPH!!!

- la variabile b e' tainted

- Problema: due chiamate alla stessa funzione id()?



tainted <= α
α <= γ
γ <= δ
δ <= β

---

untainted <= γ
δ <= ρ
ρ <= untainted

tainted <= α <= γ <= δ <= ρ <= untainted

FALSE ALARM

No solution but yet no true tainted flow!!

The analysis is imprecise:, consider a call into which a tainted value is passed, and the return into which we pass the untainted value

- il problema e' detto **context insensitivity**
  - le due chiamate alla stessa funzione sono confuse nel grafo (cfg)
  - ci serve una **context sensitive** analisi che distingue le chiamate

- **handling context sensitivity**
  - We associate a **different label to each call** (e.g. correlate the label with the line number in the program at which the call occurs).
  - We match up **calls with corresponding returns**, only when the labels on flow edges match.
  - We add **polarities** to distinguish **calls from returns**.
    - minus for argument passing, and plus for return values.
  - Context sensitivity is a tradeoff, favoring precision over scalability.
    - the context insensitive algorithm takes roughly time O(n), where n is the size of the program,
    - the context sensitive algorithm will take time $O(n^3)$
  - The added precision actually helps performance. By eliminating infeasible paths it can reduce the size of the constraint graph by a constant factor.
  - The general trend is that **greater precision means lower scalability**

- **Implicit Flow**
  - metti che voglio copiare l'array di char **src** (**tainted**) dentro **dst** (**untainted**)
    - se controllo che ogni elemento di **src** sia corrispondente ad uno dei possibili char (come intero “statico”, li provo tutti) rendo il **dato src** untainted
      - ma l'informazione e' **tainted**

```
void copy (tainted char[ ] src, untainted char[ ] dst), int len) {
  int untainted i;
  int untainted j;
  for (i=0; i<len; i++) {
    for (j=0; j < sizeof(char)*256, j++) {
      if src[i} = (char) j
          dst[i] = (char) j  // Is legal?
      }    untained    untained
    }
}
```

MISSED FLOW

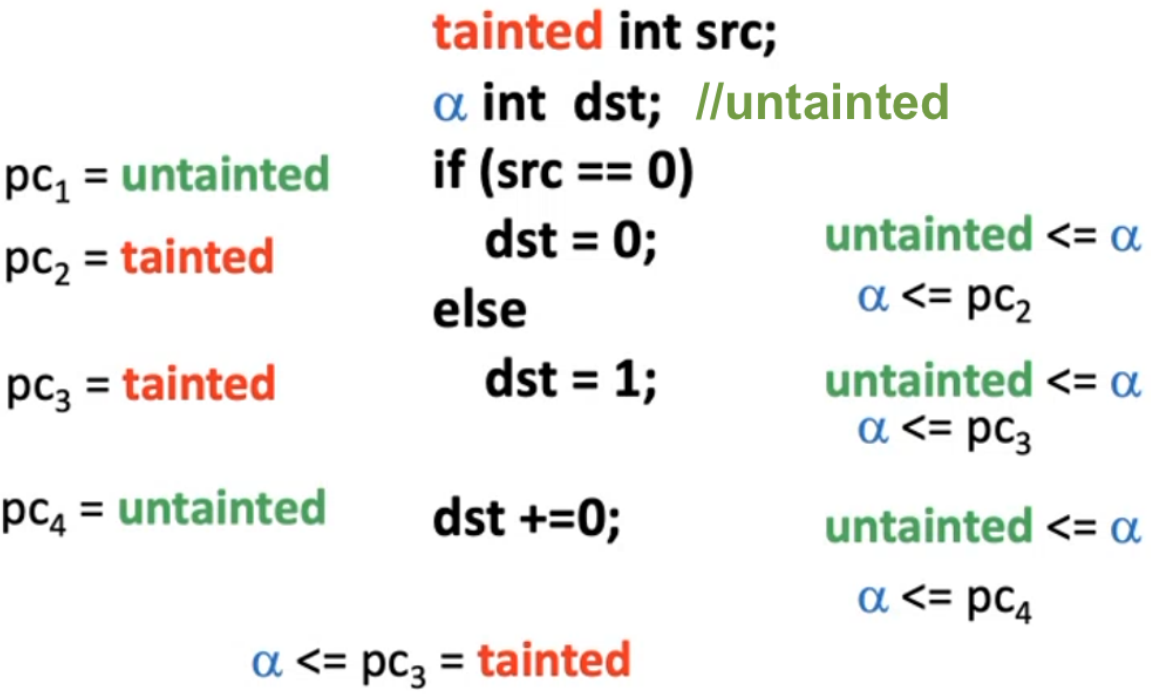
the character was not directly assigned from src but the same value was. The contents of src is certainly copied to dst: the information is leaked.

Data did not flow, but the information did

- `flow implicito`

- `Information Flow (gestione dei flussi impliciti)`
  - The analysis needs to be **more precise**:
    - We add a **taint constraint** affecting the **current flow position** abstracted by the **pc**
  - Idea the assignment x = y (the flow from y to x) now produces two constraints
    1. as expected the contraint between the taint labels of y and x
    2. the pc flow label bounds the label of x
       1. the flow from y to x does not leak through the assignment to x.

- `ESEMPIO`



- **senza l'utilizzo dei constraint dati dal program counter hai un flow implicito!**

  if the source (**src**) is zero, then **dst** will contain the same value as the source, otherwise it will contain one

- **E con i Pointers?**

α char *a =“ciao”;
(β char *) *p = &a;
(γ char *) *q = p;
ρ char *b = getsFromNetwork(…);
*q = b;
printfun(*p)

SOLUTION
α = β = untainted
ρ = γ = tainted

The analysis: constraints
untainted <= α
α <= β
β <= γ
tainted <= ρ
ρ <= γ
β <= untainted

MISSED ILLEGAL FLOW
p and **q** are aliases
writing a tainted data to **q**
makes **p** contents tainted also

- SOLUZIONE

  Pointer assignment **flows in both ways**

  α ***p** = …:
  β ***q** = …;
  :
  p = q

  α <= β
  β <= α

- questo assicura che anche gli aliasing constraints sono considerati

- SUMMARY

- **Model**: *Model of programming language behaviour*
- **Threat**: The attacker controls some data and attempts to taint programa data in oder to create security vulnerabilities (buffer overflow, string attacks, injections)
- **Countermeasures**: Analysis to track flow of data

- **NB**
  - **integrity:** gli input sono pericolosi per l'integrità dei dati
  - **confidentiality:** gli output sono pericolosi per la confidenzialità' dei dati

- **More on COVERT CHANNEL**
  - tutti quei canali di informazione non previsti che possono dare all'attaccante informazioni su informazioni riservate (o comunque aiutarlo a fare speculazioni)

More subtle forms of indirect information flows can arise via hidden channel aka covert channels aka side channels

(non)termination

```
while (hi>99) do {....};
if (hi=99) then {“loop”} else {“terminate”}
```

execution time

```
for (i=0; i<hi; i++) {...};
if (hi=1234) then {...} else {...}
```

exceptions

```
a[i] = 23
```

may reveal length of a (if i is known), or leak info about `i` (if length of a is known), or reveal if a is null..

  - altri covert channel possono essere
    - rumore della macchina (significa alto carico sulla cpu)
    - power consumption

    - ...

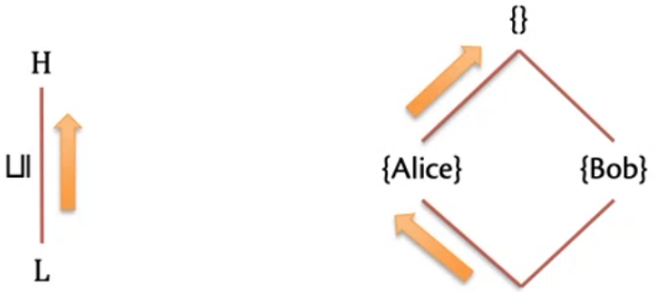
- **Information Flow POLICY**
    - una politica su IF specifica delle restrizioni sui dati
    - **esempio di policy per la confidenzialità**
        - il valore ve tutti i dati da v derivati devono poter essere letti solo dall'utente A
            - **v is allowed to flow only to A**

They form a *lattice* $\langle L, \sqsubseteq \rangle$ with join operation $\sqcup$.

For $\ell, \ell' \in L$, if $\ell \sqsubseteq \ell'$, then:

ℓ' is *at least as restrictive as* ℓ, and thus,

information flow from ℓ to ℓ' is allowed.



- For each ℓ and ℓ', there should exist label $\ell \sqcup \ell'$, such that:
  - $\ell \sqsubseteq \ell \sqcup \ell'$, $\ell' \sqsubseteq \ell \sqcup \ell'$, and
  - if $\ell \sqsubseteq \ell''$ and $\ell' \sqsubseteq \ell''$, then $\ell \sqcup \ell' \sqsubseteq \ell''$.
- $\ell \sqcup \ell'$ is called the **join** of ℓ and ℓ'.
- Examples: $L \sqcup L = L$, $H \sqcup H = H$, $L \sqcup H = H$

Is a flow allowed?

Given

- a lattice $\langle \{L, H\}, \sqsubseteq \rangle$ of labels,
- a program C, and
- labels on program inputs and outputs,

are all the flows from inputs to outputs that are caused by executing C allowed?

- **NON INTERFERENCE POLICY**
  - Under a non-interference provision, a computer is seen as a machine having inputs and outputs. These are categorized in terms of their sensitivity as low (not classified information, or having a low sensitivity) or high (sensitive, and not to be viewed by individuals or resources without the necessary clearance).
    - According to the conditions laid down by the model, any sequence of low sensitivity inputs will produce outputs which are correspondingly low, regardless of any high level inputs that may also exist.
  - **se un programma C soddisfa NI tutti i flussi da input ad output sono permessi**
- altri covert channel possono essere

- **State Equivalence**

  Le C be a program

  $C(M_i)$ are the observations produced by executing C to termination on initial state M_i:

  - **final outputs, or**
  - **intermediate and final outputs.**

  Then, observations tagged with L should be the same:

  - $C(M_1) =_L C(M_2)$.

  $\forall M_1, M_2$: **if** $M_1 =_L M_2$, **then** $C(M_1) =_L C(M_2)$

    - **dove per un programma C e relativi mapping variable to label {L, H}**
    - **e il per ogni M1, M2 e' la policy di non interferenza**
    - **C(M1) =LC(M2)**
      - **il programma C, prese due memorie M1, M2, produce sempre le stesse "osservazioni" (observable by the attacker) Low**

- **ATTENZIONE**
  - per come abbiamo definito il lattice e le policy si hanno problemi relativi alla combinazione
    - si prende sempre la restrizione maggiore
      - metti di avere due dati Low
        - nome studenti
        - voto esami
        - la combinazione di queste dovrebbe essere H ma in realta' verra' definita con L
      - idem con dai High
        - la combinazione di dati high produrra' sempre high quando in realta' potrebbe dover produrre dati Low
          - messaggio segreto, chiave simmetrica (HIGH)
            - messaggio cifrato: dovrebbe essere LOW!
    - serve definire un processo di declassificazione
      - definire chi come dove e perche' declassa la sicurezza

- TYPE SYSTEM FOR INFORMATION FLOW
  - Static
  - Fixed environment Γ
  - Security levels (labels) as types
    - Security level Γ(**x**) is the type of **x**.
  - **Goal: type-correctness ⇒ noninterference**
- Typing Expressions
  - Judgement $\Gamma \vdash \mathbf{e} : l$
  - Intuitive meaning: According to environment Γ, expression **e** has type (i.e., label) l.

$$\frac{}{\Gamma \vdash n: \bot}$$

⊥ Is the least restrictive type
The minimum of the lattice

$$\frac{\Gamma(x) = \ell}{\Gamma \vdash x: \ell}$$

$$\frac{\Gamma \vdash e1: \ell 1, \Gamma \vdash e2: \ell 2}{\Gamma \vdash e1 + e2: \ell 1 \sqcup \ell 2}$$

- non va dimenticato il contesto (if then else)

- typing conditionals

if y>0 then x:=1 else x:=2

#1 The security level of x is not sufficent to type IF-branches
The security context (*cxt*) of both branches is that of Γ(y)

#2 In or example: Check if $ctx \sqsubseteq$ Γ(x)
where ctx = Γ(y).

Judgements of the form

$$\Gamma, ctx \vdash c$$

Γ is the **environment** mapping variables to security labels

cxt is the **context** label (implicit flow)

Intuitive meaning: According to environment Γ, and context label ctx, statement **c** is type correct.

- gamma e' detto ambiente dei tipi

- QUINDI

Static type system

Assignment-Rule:
$$\frac{\Gamma \vdash \mathbf{e} : \ell \qquad \ell \sqcup ctx \sqsubseteq \Gamma(\mathbf{x})}{\Gamma, ctx \vdash \mathbf{x := e}}$$

If-Rule:
$$\frac{\Gamma \vdash \mathbf{e} : \ell \qquad \Gamma, \ell \sqcup ctx \vdash \mathbf{c1} \qquad \Gamma, \ell \sqcup ctx \vdash \mathbf{c2}}{\Gamma, ctx \vdash \mathbf{if\ e\ then\ c1\ else\ c2}}$$

While-Rule:
$$\frac{\Gamma \vdash \mathbf{e} : \ell \qquad \Gamma, \ell \sqcup ctx \vdash \mathbf{c}}{\Gamma, ctx \vdash \mathbf{while\ e\ do\ c}}$$

Sequence-Rule:
$$\frac{\Gamma, ctx \vdash \mathbf{c1} \qquad \Gamma, ctx \vdash \mathbf{c2}}{\Gamma, ctx \vdash \mathbf{c1;c2}}$$

- **NB:**

If a program does not satisfy NI, then type checking fails.

But, if type checking fails, then the program might or might nor satisfy NI.

- **il type system e' conservativo: ammette falsi positivi**
  - cioè' che soddisfano il type checker ma che non la non interfirence

- **Can we build a complete system?**
  - Is there an enforcement mechanism for information flow control that has no false negatives?
    - A mechanism that rejects only programs that do not satisfy noninterference?
  - No! [Sabelfeld and Myers, 2003]
    - "The general problem of confidentiality for programs is undecidable."
    - The halting problem can be reduced to the information flow control problem.
    - Example:

      `if h>1 then c; l:=2 else skip`
      - If we could precisely decide whether this program is secure, we could decide whether **`c`** terminates!
- **tutta la baracca del type system non gestisce minimamente il covert channel costituito dalla non termination**

To prevent covert channels due to infinite loops, we can strengthen the typing rule for while-statement, to allow only low guard expression:

$$\frac{\Gamma \vdash \mathbf{e} : \bot \qquad \Gamma, ctx \vdash \mathbf{c}}{\Gamma, ctx \vdash \mathbf{while\ e\ do\ c}}$$

type correctness implies termination sensitive NI

- **ma in questo modo si hanno problemi di tipo!**

  while h > 0 **do** h= h+1; **is not well typed**!!!

- **Portiamo tutto a RUNTIME (type checker) e vediamo se riusciamo a ridurre il numero di falsi positivi relativi alla NI**

Dynamic mechanisms use run time information to decrease false negatives.

A run-time monitor checks/deduces labels along the execution:

- When an assignment **x:=e** is executed
  - either check whether $\Gamma(\mathbf{e}) \sqcup ctx \sqsubseteq \Gamma(\mathbf{x})$ holds (fixed Γ)
    - *The execution of a program is halted when a check fails.*
  - or deduce $\Gamma(\mathbf{x})$ such that $\Gamma(\mathbf{e}) \sqcup ctx \sqsubseteq \Gamma(\mathbf{x})$ holds (flow-sensitive Γ).
- Monitor maintains a context label ctx.
  - When execution enters a conditional command, the monitor augments ctx with the label of the guard.
  - When execution exits a conditional command, ctx is restored.

    - **attenzione**
      - un meccanismo dinamico potrebbe leakare informazioni quando decide di terminare un esecuzione per via di un check fallito

- Assume fixed Γ: $\Gamma(\mathbf{h})$=H and $\Gamma(\mathbf{l})$=L.
- Consider program:

```
p:=0;
if h>0 then l:=1 else h:=1;
l:=2
```

- If **h>0** is *true*, then execution is halted.
  - No public output.
- If **h>0** is *false*, then execution terminates normally.
- Problem: **h>0** is leaked to public outputs.
  - lo stop e' dovuto al contesto del then!

Flow sensitive labels

Assume **Γ(h) =H and Γ(l) =L.** Consider the program

x:=h; x:=0; l:=x

Is this program safe?

- If Γ(x) is fixed to H, then the program is rejected, because the type analysis of l:=x fails.

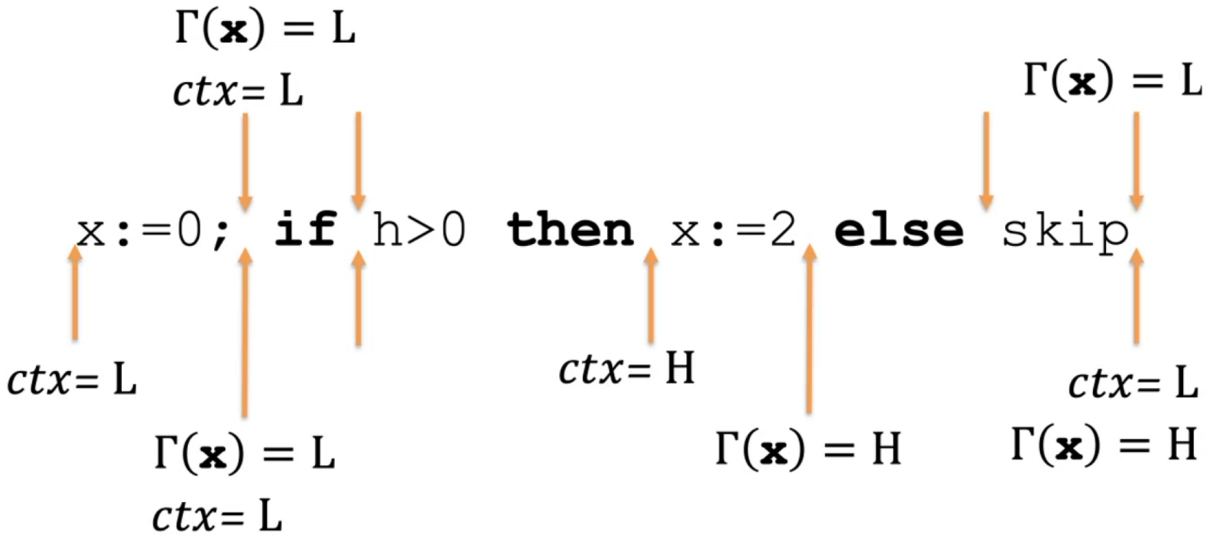
If Γ(x) is flow-sensitive, then

- Γ(x) becomes H **after x:=h**,
- Γ(x)= becomes L **after x:=0**,
- the analysis of l:=x succeeds.

Flow-sensitive labels can enhance permissiveness even further.

- **MEGA ESEMPIO**

Assume fixed $\Gamma(\mathbf{h}) = \mathrm{H}$ and flow-sensitive $\Gamma(\mathbf{x})$.



- **sopra la riga ramo else, sotto la riga ramo then**

- problema

```
x:=0; if h>0 then x:=2 else skip
```

- If **h>0** holds, then after **x:=2**, Γ(**x**) becomes H.
- If **h>0** does not hold, then Γ(**x**) remains L.

This label is not sound!

Problem: Even though **h** flows to **x**,

1. **x** is tagged with H only when **h>0**;
2. **x** is tagged with L otherwise

  - non e' sound perche' sui due rami ha tipo diverso
    - pensa a ocaml
  - soluzione 1
    - Make purely dynamic flow-sensitive mechanism more conservative:
    - Block execution before entering conditional commands with high guards.

    Back to our example:

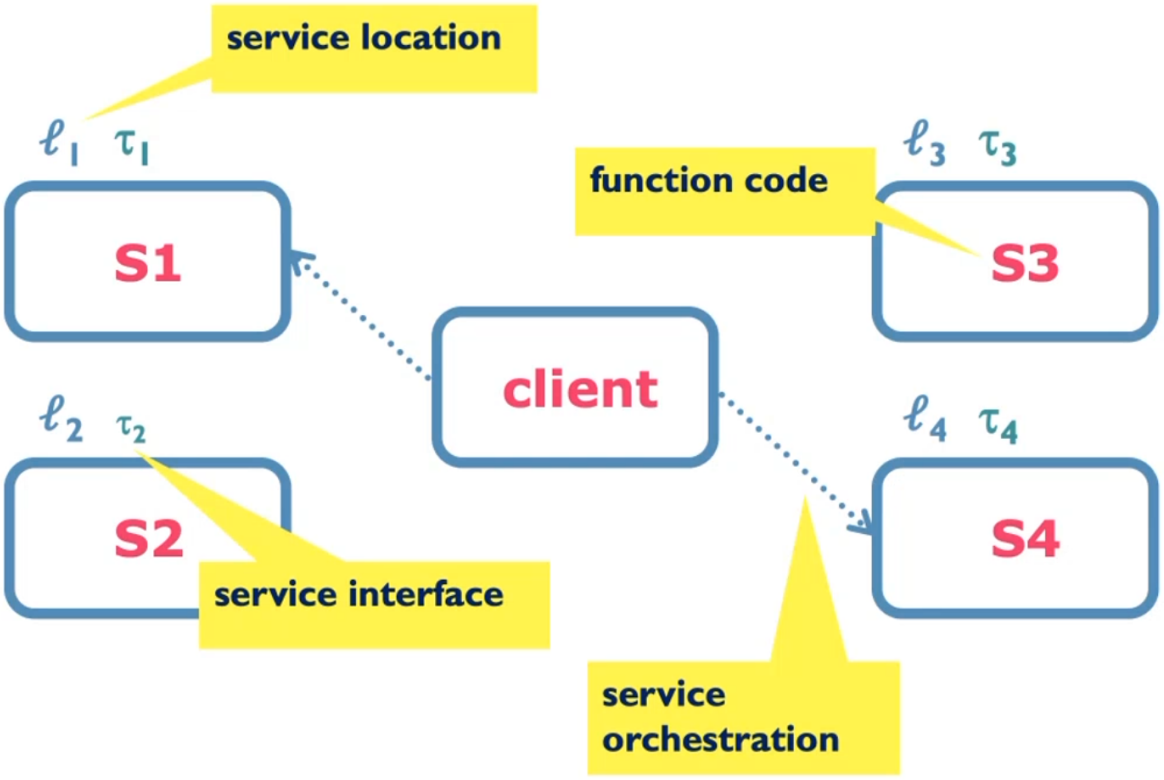
    x:=0; **if** h>0 **then** x:=2 **else** skip

    All execution would stop after x:=0.

  - soluzione 2
    - **Exploit on-the-fly static analysis to update the labels of target variables in untaken branches to capture implicit flow.**
    - **The mechanism is no longer purely dynamic.**
      - faccio analisi statica al volo prima di eseguire la guardia (a runtime non controlli entrambi i rami, staticamente si)
        - così' ho risolto

- **quindi: inferenza dei tipi relativi alla non interference**
  - Static:
    - Low run time overhead.
    - No new covert channels.
    - More conservative.
  - Dynamic
    - Increased run time overhead.
    - Possible new covert channels.
    - Less conservative.
  - Ongoing research for both static and dynamic.
    - Different expressiveness of policies, different NI versions, different mechanisms.

- Programming in a World of Stateless Service



- Two kinds of security concerns:
  - secrecy of transmitted data, authentication, etc (**network security**)
  - control on computational resources (**access control, resource usage analysis, information flow control, etc**)
- need for linguistic mechanisms that:
  - **work in a distributed setting**
  - **assume no trust relations among services**
  - **can also cope with *mobile code***

- **In-Line Monitor: Safety Framings**

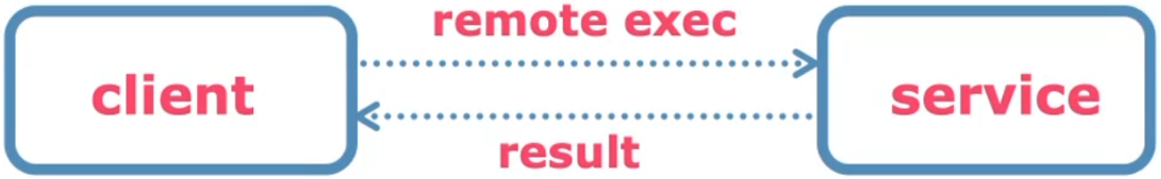
Client: protect from untrusted code

In-line monitor: safety framing

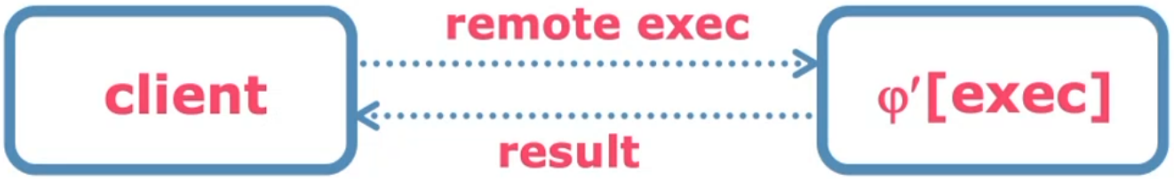
The policy φ is enforced stepwise within its scope

  - bisogna instrumentare il codice mobile in modo che non faccia danni

Services: protect from clients



(now the safety framing belongs to the service)



Scoped policies check the local execution histories

  - il contrario di prima: il cliente non ha potenza di calcolo e quindi fa una remote execution

- **Function as a Service**

- `function selection`

  Problems with "request by name" :

  - what if named service **S2** becomes unavailable ?
  - …and if **S2** is outperformed by **S1** or **S3** ?
  - hard reasoning about non-functional properties of services (e.g. security)
  - security level independent of the execution context (unless hard-wired in the service code)

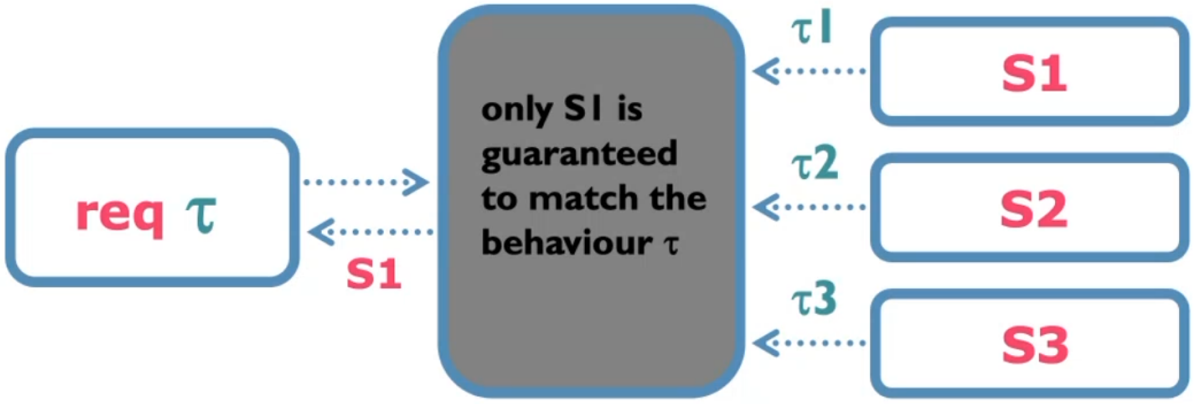
  From syntax-based to semantics-based invocation

  Service names $\ell,\ell',..$ tell me nothing about the behaviour!

- `la soluzione e'`

  Req-by-contract: request a service respecting

  the desired behaviour



  τ imposes both functional and non-functional constraints

  - `tau rappresenta un comportamento`

- **mobile code download and execution**

  Example: download code that obeys the policy φ

  - **voglio un comportamento tau_0 e il servizio mi risponde un codice che va da tau_1 a tau_2 rispettando la politica phi**
    - esempio
      - **tau_0:** "dammi il codice per analizzare questi dati che ho"
      - **servizio:** risponde con del codice che analizza quei dati
- **remote exec**

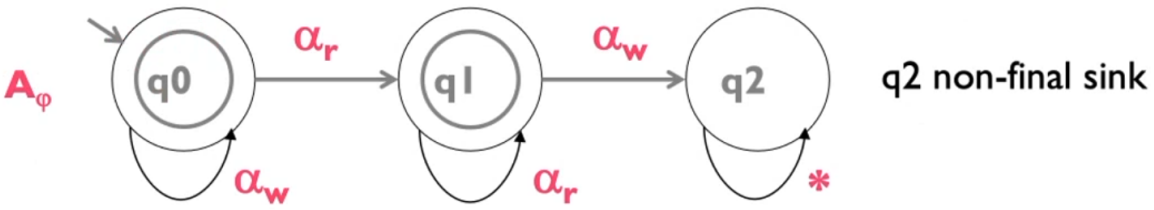
  Example: a remote executer that obeys the policy φ′

- **Observable Behaviour**
  - **access events** are the actions relevant for security (e.g. read/write local files, invoke/be invoked by a given service, etc)
    - mechanically inferred, or inserted by programmer.
    - their meaning is fixed globally.
    - access events cannot be hidden.
  - the (abstract) behaviour observable by the orchestrator over-approximates the **histories**, i.e. sequences of access events, obtainable at run-time.

- **What kind of Policies**
  - History-based security
  - Policies φ are **regular** properties event histories
  - Policies φ, φ' have a **local scope**, possibly **nested** $\varphi[\cdot\cdot\varphi'[\cdot\cdot]\cdot\cdot]$
  - A policy can only control histories of a **single** site (no trust among services)
  - Histories are **local** to **stateless** service sites (stateful easy)
    - **esempio**

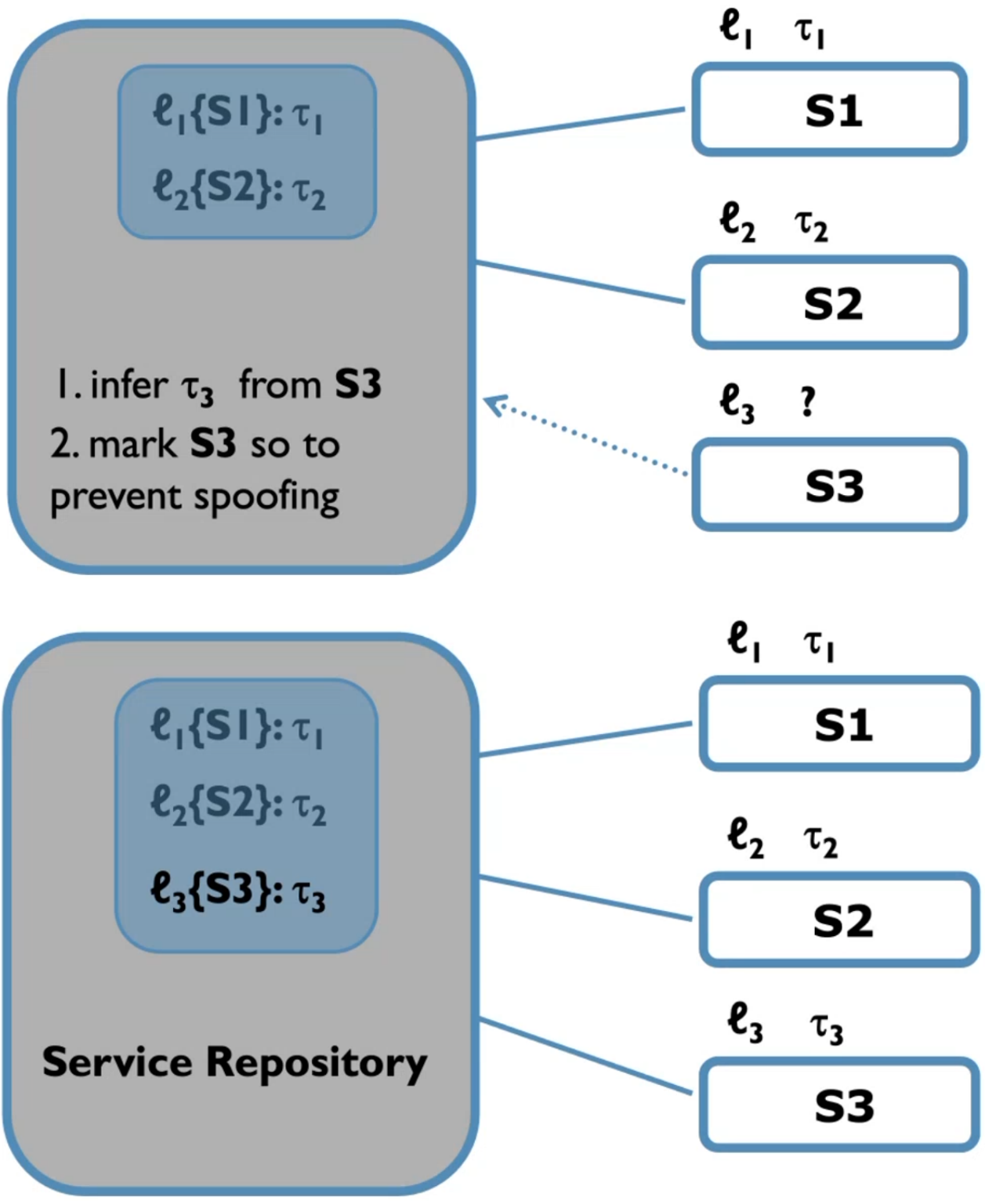
      φ Chinese Wall: cannot write (α_w) after read (α_r)



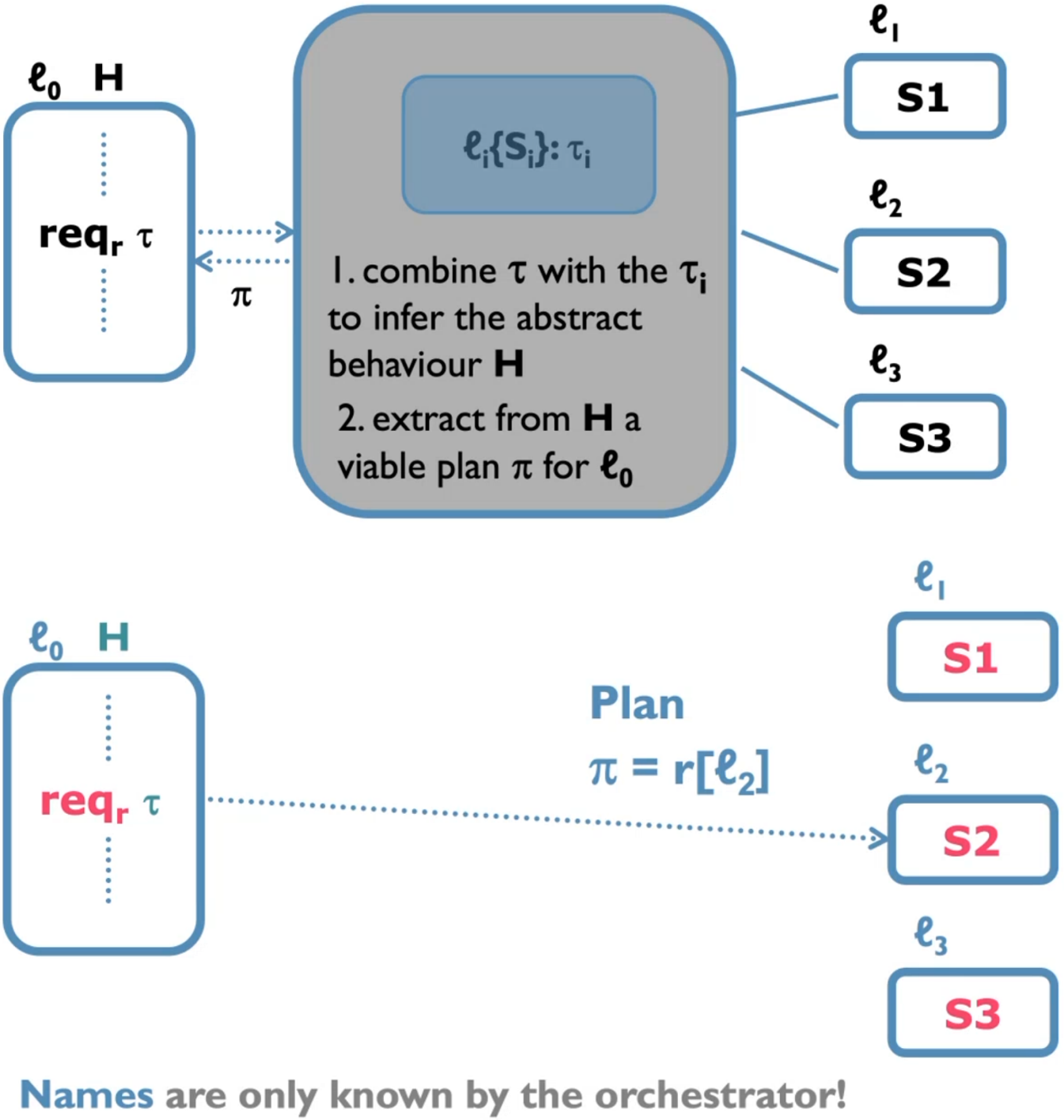
$\alpha_w \alpha_r \alpha_w \not\models \varphi$

- **principio del privilegio minimo**
  - i programmi devono avere l'insieme minimo di permessi, il tanto che basta al loro funzionamento
    - un servizio deve sempre obbedire a tutte le policy attive
    - le policy devono sempre poter vedere tutta la history

- **Service Publication**
  - pubblicazione di un servizio (funzionalità)
    - pensa ai remote methods in java


$\ell_1\{S1\}: \tau_1$
$\ell_2\{S2\}: \tau_2$
1. infer τ_3 from **S3**
2. mark **S3** so to prevent spoofing
ℓ_1 τ_1
S1
ℓ_2 τ_2
S2
ℓ_3 ?
S3
$\ell_1\{S1\}: \tau_1$
$\ell_2\{S2\}: \tau_2$
$\ell_3\{S3\}: \tau_3$
Service Repository
ℓ_1 τ_1
S1
ℓ_2 τ_2
S2
ℓ_3 τ_3
S3

- Service Orchestration



- `what is a plan?`
  - A plan drives the execution of an application, by associating each service request with one (or more) appropriate services
  - With a **viable plan**:
    - executions never violate policies
    - there are no unresolved requests
    - you can then dispose from any execution monitoring!
  - Many kinds of plans:
    - **Simple:** one service for each request
    - **Multi-choice:** more services for each request
    - **Dependent:** one service, and a continuation plan
    - ...
- `who do we trust`

  The orchestrator, that:

  - certifies the behavioural descriptions of services **(types annotated with effects H)**
  - composes the descriptions, and ensures that selected services match the requested types
  - extracts the **viable plans** (through model-checking)

  Also, someone must ensure that services do not change their code on-the-fly

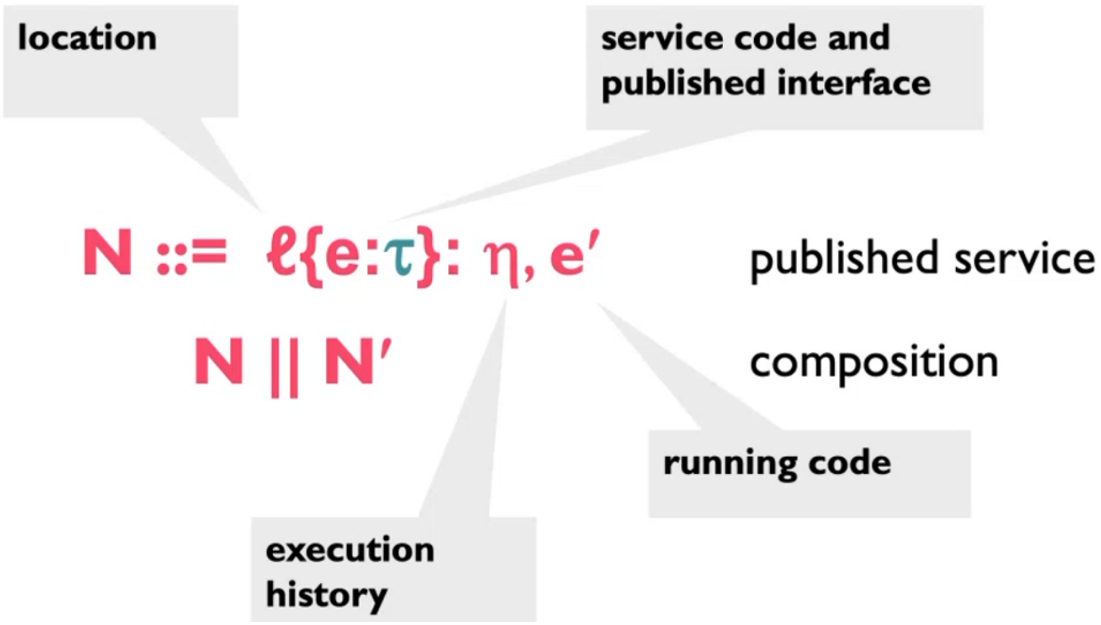
- **SUMMING UP**

a programming model for secure service composition:	static orchestrator:
• **distributed** stateless services • **safety framings** scoped policies on localized execution histories • **req-by-contract** service invocation	• certifies the **behavioural interfaces** of services • provides a client with the **viable plans** driving secure executions

- **SERVICE: FUNCTIONAL + SECURITY + COMMUNICATIONS**

Services	$e ::= x$	variable
	α	access event
	$\text{if } b \text{ then } e \text{ else } e'$	conditional
	$\lambda\, x.e$	Lambda
	$\lambda_z\, x.e$	Recursive lambda
	$e\ e'$	application
	$\varphi[e]$	safety framing
	$\text{req}_r\ \tau$	service request
	$\text{wait}\ \ell$	wait reply

- NETWORK SW ARCHITECTURE



- Plans (mapping requests to service endpoints)

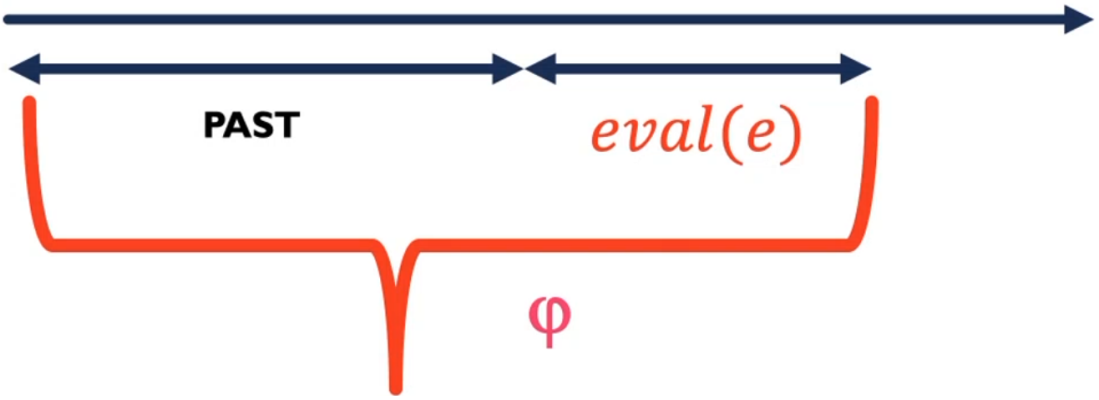
A plan is a function from requests r to services ℓ

π ::=	0	empty
	$r[\ell]$	service choice
	$\pi \mid \pi'$	composition

Plans respect the partial knowledge $\ell < \ell'$ of services about the network ($<$ is a partial ordering)

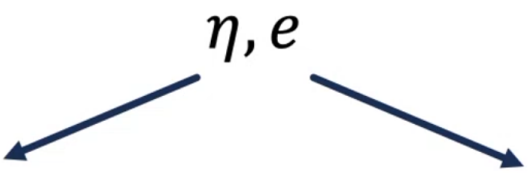
- HISTORY DEPENDANT ACCESS CONTROL

During the evaluation of φ[e] , the whole execution history (i.e. the past history followed by the events generated so far by e) must respect the policy φ.



- Abstract Machine Definition

Configuration of the abstract machine



Execution history **Program to be executed**

Local histories only record the events generated by a service locally on its operational environment

Operational rules

$$\eta\, e \rightarrow \eta'\, e'$$

$$\ell\{e:\tau\}:\ \eta, e1 \rightarrow_{\pi} \eta', e2$$

The evolution of a node is driven by the plan π

- **NB:** l'esecuzione termina con un valore **v**
  - **v:** variabili, astrazioni (funzioni) o richieste

- **Inference Rules for Abstract Machine Executions**

[Event]

$$\eta, \alpha \rightarrow \eta\alpha, ()$$

[Framing In]

$$\frac{\eta, e \rightarrow \eta', e' \qquad \eta' \models \varphi}{\eta, \varphi[e] \rightarrow \eta', \varphi[e']}$$

[Framing Out]

$$\frac{\eta \models \varphi}{\eta, \varphi[v] \rightarrow \eta, v}$$

[If] $\eta, \text{if b then } e_{true} \text{ else } e_{false} \rightarrow \eta, e_{\mathcal{B}(b)}$

[App1]

$$\frac{\eta, e_1 \rightarrow \eta', e_1'}{\eta, e_1\ e_2 \rightarrow \eta', e_1'\ e_2}$$

[App2]

$$\frac{\eta, e_2 \rightarrow \eta', e_2'}{\eta, v\ e_2 \rightarrow \eta', v\ e_2'}$$

[AbsApp1] $\eta, (\lambda x.e)\ v \rightarrow \eta, e\{v/x\}$

[AbsApp2] $\eta, (\lambda_z x.e)\ v \rightarrow \eta, e\{v/x, \lambda_z x.e/z\}$

- **abstract machine network behaviour**

[Inject]

$$\frac{\eta, e \rightarrow \eta', e'}{\ell: \eta, e \rightarrow_\pi \ell: \eta', e'}$$

  - piano passato da l'orchestratore, passo di computazione locale produce un evoluzione del piano

[Par]

$$\frac{N_1 \rightarrow_\pi N_1'}{N_1 \,||\, N_2 \rightarrow_\pi N_1' \,||\, N_2}$$

- esecuzione asincrona: una parte della rete fa un passo (non condiviso) -> tutta la parte della rete ha fatto quel passo

[Request] $\pi = r[\ell'] \mid \pi'$

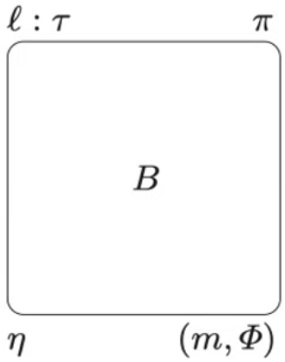
$$\ell: \eta, \mathrm{req}_r\ v \ \|\ \ell'\{e'\}: \varepsilon, \star \ \rightarrow_{\pi}\ \ell: \eta, \mathrm{wait}\ \ell' \ \|\ \ell'\{e'\}: \varepsilon, e'v$$

[Reply]

$$\ell: \eta, \mathrm{wait}\ \ell' \ \|\ \ell'\{e'\}: \eta', v \ \rightarrow_{\pi}\ \ell: \eta, v \ \|\ \ell'\{e'\}: \varepsilon, \star$$

- **request:** **l** richiede **r** con parametro **v** e il piano **pi** specifica che **r** lo fa **l'** (**pi = r[l'] | pi'**) quindi
  - **->**
    - **l** con storia **eta** va in wait per **l'** ed in parallelo **l'** esegue **e'** con storia nulla e parametro **v** passato da **l**
- **reply:** "al rovescio" (e' analogo)

- **UNA FORMULAZIONE ALTERNATIVA**
  - **services: an execution state**



$\ell : \tau$	service location ℓ + interface τ
π	orchestration plan
η	event history
(m, Φ)	monitor flag m + sequence Φ of active policies
B	service code

$$\ell\langle e : \tau \rangle : \pi \triangleright \eta, m, e'$$

Flag m representing the on/off status of the execution monitor on the active security policies.
When the flag is on, the monitor checks that the service history η adheres to the policies at hands.

- **stand alone services**

$$\eta, m, (\lambda_z x.\, e)v \to \eta, m, e\{v/x, \lambda_z x.\, e/z\}$$
$$\eta, m, \alpha \to \eta\alpha, m, *$$
$$\eta, m, \texttt{if}\ b\ \texttt{then}\ e_{tt}\ \texttt{else}\ e_{ff} \to \eta, m, e_{\mathcal{B}(b)}$$
$$\eta, m, \mathcal{C}(e) \to \eta', m', \mathcal{C}(e') \quad \text{if } \eta, m, e \to \eta', m', e' \text{ and } m' = \mathit{off} \vee \eta' \models \Phi(\mathcal{C})$$
$$\eta, m, \mathcal{C}(\varphi[v]) \to \eta, m, \mathcal{C}(v) \quad \text{if } m = \mathit{off} \vee \eta \models \varphi$$

where $\mathcal{C}$ is an *evaluation context*, of the following form:

$$\mathcal{C} \ ::= \ \bullet \mid \mathcal{C}\, e \mid v\, \mathcal{C} \mid \varphi[\mathcal{C}]$$

and $\Phi(\mathcal{C})$ is the set of *active policies* of $\mathcal{C}$, defined as follows:

$$\Phi(\mathcal{C}\, e) = \Phi(v\, \mathcal{C}) = \Phi(\mathcal{C}) \qquad \Phi(\varphi[\mathcal{C}]) = \{\varphi\} \cup \Phi(\mathcal{C})$$

  - **vanno ovviamente ridefinite le regole di inferenza viste sopra, adattandole alla nuova formulazione che prevede il flag m**

- **PLANNING SECURE SERVER COMPOSITION**
  - **The problem we face**
    - **The client program to behave safely (and correctly) requires the composition of services which respect certain security guarantees.**
  - **The proposed methodology**
    - **Identify statically a trusted orchestrator that provides client the viable plans guaranteeing that the invoked services always respect the required properties.**
  - **The orchestrator is the TCB**
- **SECURE ORCHESTRATION STATICALLY**
  - secure orchestrator ?

1. We extract the **abstract behaviour** of the orchestration via a suitable **type system** that ***over-approximates the possible run-time behavior of all the services involved in an orchestration***
2. This abstract behaviour is **compiled** into a suitable form to be **model-checked** in order ***to verify whether or not the security constraints are satisfied***
3. As a result of the model checking we build the **trusted orchestrator** that ***securely coordinates the running services***.

- `Type System for Orchestration`
- `sistema dei tipi a doppio effetto`
  - **types** carry annotations **H** about service abstract behaviour
  - **effects H**, are called ***history expressions***, and over-approximate the actual execution histories
  - the type & effect inferred for a service depends on its **partial knowledge** of the network
- `types`

(basic types are pretty standard)

$$\tau ::= int \mid bool \mid unit \mid \cdots \mid$$

$$\tau \xrightarrow{H} \tau'$$

Functional types

The history expression H describes the latent effect associated with the functional abstraction (the set of possible execution histories)

When the abstraction is applied to a value, its execution will generate a run time behaviours belonging to the execution histories represented by H

- Adding effects to typing rules mean to compute statically something about "***how program executes***" .
- There are many things we may want to conservatively approximate

H ::=

ε	**empty**
h	**variable**
α	**access event**
$H \cdot H'$	**sequence**
$H + H'$	**choice**
$\mu h.H$	**recursion**
$\varphi[H]$	**security block**
$\ell: H$	**localization**
$\{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}$	**planned selection**

Histories are valid when they arise from computations that do not violate any security constraint

$$\eta = \alpha_w \cdot \alpha_r \cdot \varphi[\alpha_w]$$

φ = no write after read

- **Typing Judgements**

$$\Gamma, H \vdash e : \tau$$

the service e evaluates to a value of type τ, and produces a history denoted by the effect H

  - il servizio **e** e' valutato su un tipo **tau** e il suo comportamento (astratto, cioè kinda simulazione statica) e' denotato da **H**
    - ci sono le regole di inferenza da costruire per staticamente desumere staticamente H (che e' l'espressione linguistica che approssima il comportamento del codice **e**)
      - **tau** e' il tipo di dato che calcola **e**
      - **H** e' l'approssimazione del come **e** calcola il dato di tipo **tau** (derivata staticamente con il giusto type system)

- una volta costruito il type checker che costruisce H tu puoi fare model checking su dei programmi le cui politiche di sicurezza devono valere sulla storia
  - tutta sta fatica per rendere sicuro l'orchestratore

- **CONCLUSIONI**

A linguistic framework for secure service composition

- **Goal: apply Programming Language techiques to formally specify the language tool chain**
- **safety framings, policies, req-by-contract**
- **type & effect system**
- **verification of effects, to extract viable plans to contruct the trusted orchestrator**
- **The orchestrator securely composes and runs stateless services**