Secure Programming via Libraries

Implementing Erasure Policies using Taint Analysis

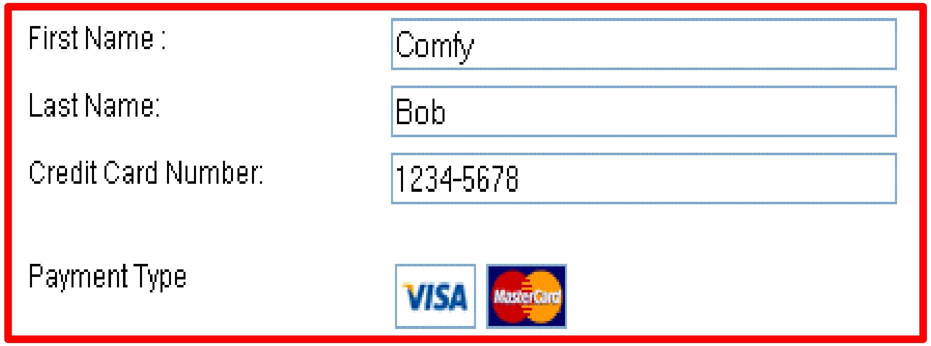
Alejandro Russo (russo@chalmers.se)

Escuela de Ciencias Informáticas (ECI) 2011
UBA, Buenos Aires, Argentina

CHALMERS

What is Erasure?

- A property of **systems** that **require sensitive information** to complete their tasks



First Name :	Comfy
Last Name:	Bob
Credit Card Number:	1234-5678
Payment Type	VISA MasterCard

- Intuitively:
  - A **user** owns some **sensitive data**
  - The system **takes** user's input and **processes** it
  - After the task is completed, **user's input and any derived** data must be **removed** from the system

Language-based Erasure

[Chong, Myers 05]

- Consider programs where
  - No I/O involved
  - Each memory location is equipped with a policy
- Erasure policies:
  - A conditional expression that raises the security level to an higher one
- Erasure: a system is *erasing* if the memory location policies are not violated during execution
- Enforcement: no mechanism is described

Just forget it

[Hunt, Sands 08]

- Programs in a simple I/O imperative language
- Erasure policies are embedded in the language by a dedicated command

  ```
  input x from a in C erasing to b
  ```

- A program is *erasing* if its behavior after the erasure command does not depend on the input received
  - Connection with information-flow
- A type system guarantees a static enforcement, but it works only for that toy language
  - Interesting theoretical result

Ingredients for Erasure

- There are several **design options** to consider
- How to **characterize** an **erasing** system?
  - One way is to define **policies** on its **observable behavior** [Hunt, Sands 08]
- **When,** and under **which conditions**, should erasure take place?
  - Need for an **erasure policy language**
- How to **enforce the erasure policies**?

We propose a Python library attempts to answer these questions

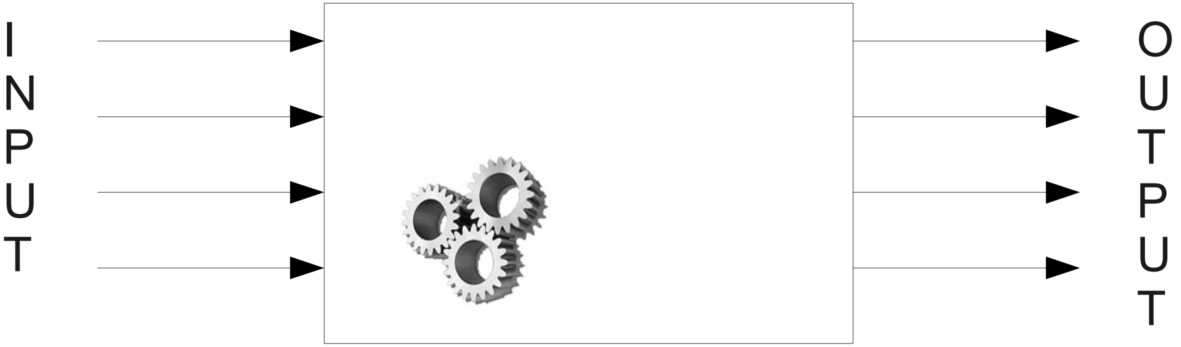
The Erasure Library in a Nutshell

[Del Tedesco, Russo, Sands 10]

- It deals with interactive systems
- It enforces erasure by preventing differences in the observable behavior of the system
- It takes into account complex policies
  - Policies may involve time, or can be triggered by updates in runtime values
  - Python features make it possible to include the library in a program with minor modifications
- It uses taint analysis to track derivate data from data that need to be erased

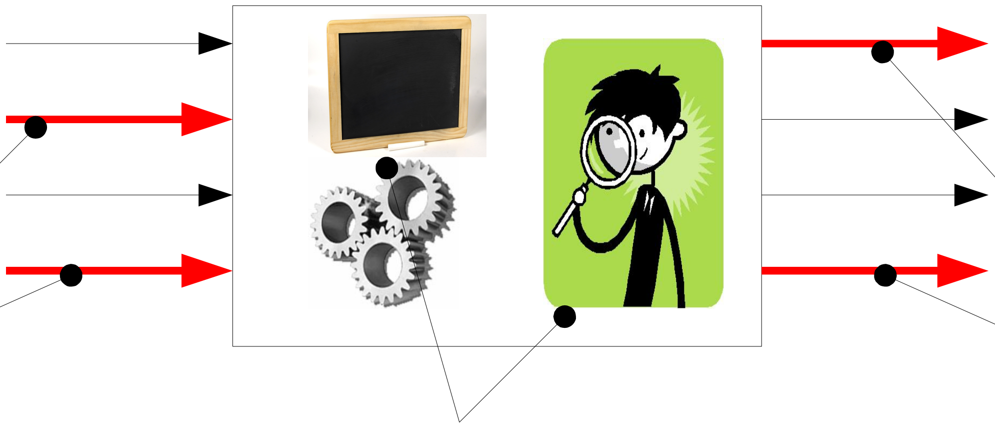
The Erasure Library

- We have a system with I/O.
- What is the purpose of our library?



The Erasure Library

- We have a system with I/O
- The library provides wrappers and internal structures to enforce erasure policies



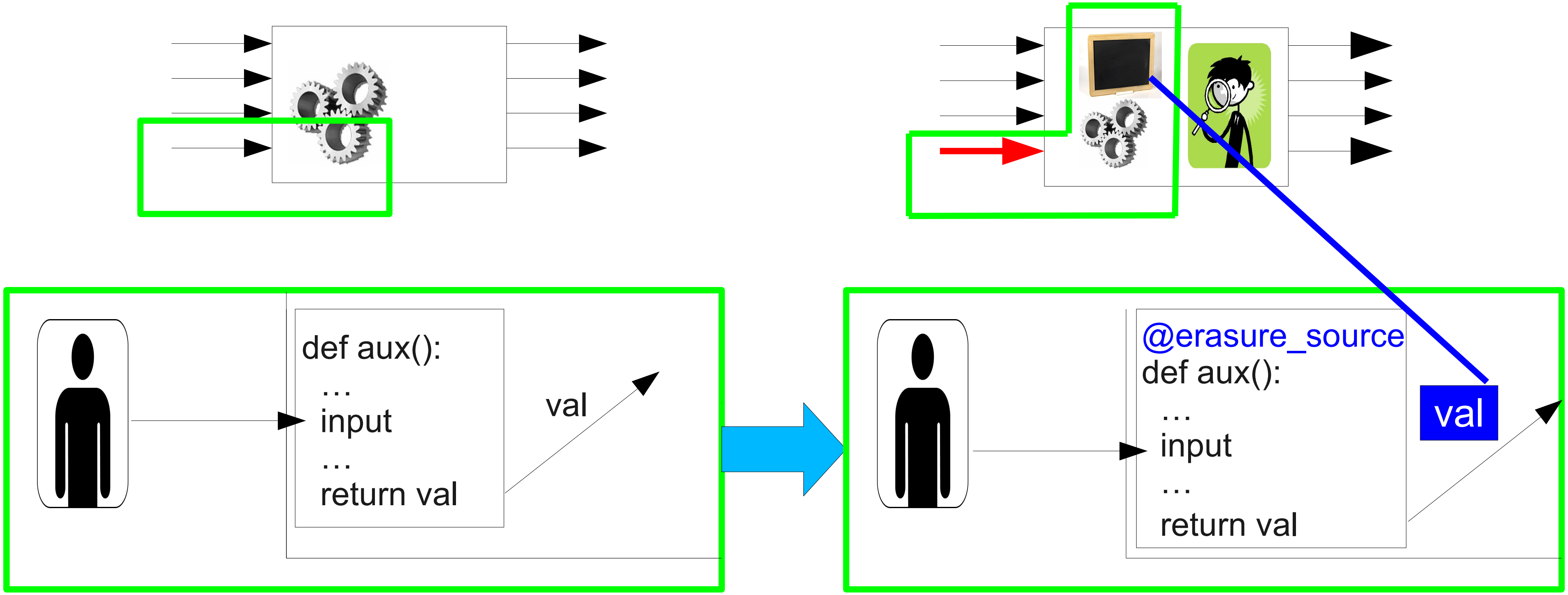
Denote entry points for *erasure-aware* information (sensitive data)

Track the propagation of *erasure-aware* data inside the system.
Implementing the concrete data removal operation

Specify which output actions we need to "observe"

API: Indicating Erasure-aware Data

- Usually systems collect sensitive data from the outside through auxiliary functions
- The library exports `erasure_source` to make such functions erasure-aware

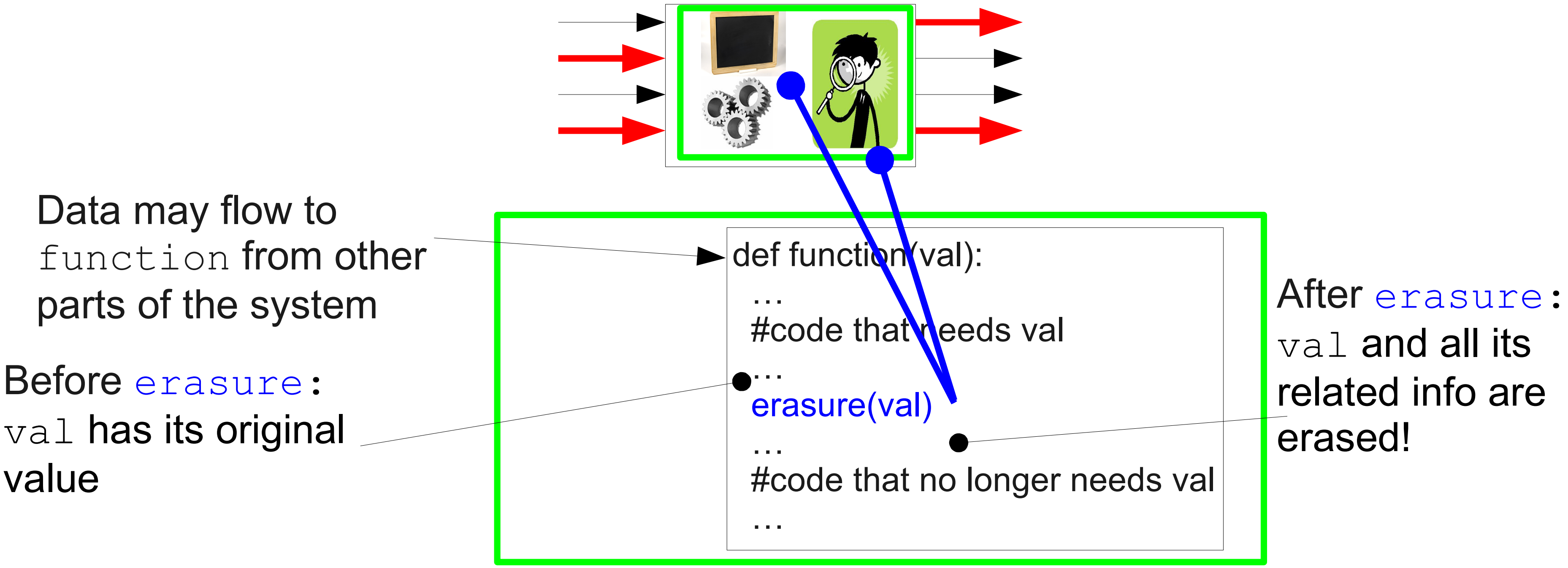


```
def aux():
  …
  input
  …
  return val
```

```
@erasure_source
def aux():
  …
  input
  …
  return val
```

API: Erasing information

- When information is no longer needed, it can be removed
- Derived information has to be removed as well!
  - Taint analysis keeps track of derived information
- The library performs erasure by the `erasure` primitive



Data may flow to `function` from other parts of the system

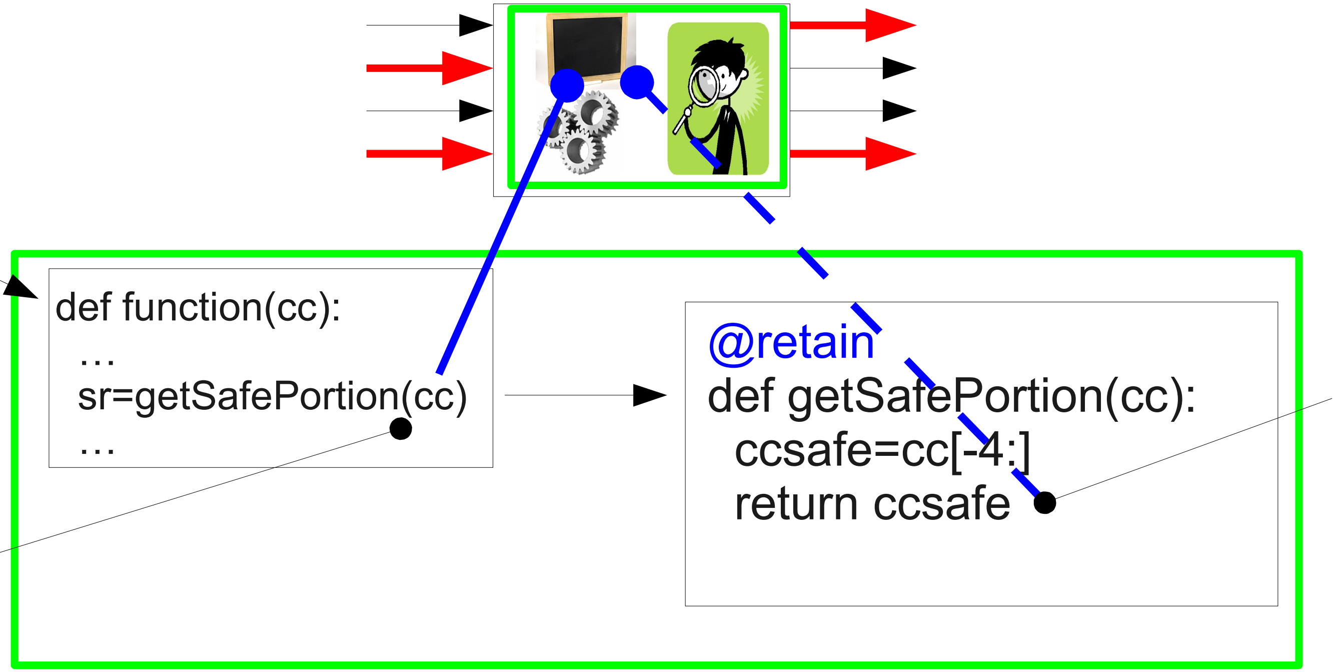
Before `erasure`: `val` has its original value

```
def function(val):
  ...
  #code that needs val
  ...
  erasure(val)
  ...
  #code that no longer needs val
  ...
```

After `erasure`: `val` and all its related info are erased!

API: Retaining Bits of Sensitive Data

- Sometimes it is necessary to retain portions of sensitive data
- Think about last digits of CC numbers in bills
- The library prevents those bits being retain (remembered) by providing primitive `retain`



```
def function(cc):
  …
  sr=getSafePortion(cc)
  …
```

```
@retain
def getSafePortion(cc):
  ccsafe=cc[-4:]
  return ccsafe
```

An erasure-aware value is provided

Regardless of `retain`, `cc` is still erasure-aware

`ccsafe` (therefore `sr`) is no longer controlled by the library

Example

Imports the library

```
from erasure import erasure_source, erasure, retain
@erasure_source
def inputFromUser():
  x=raw_input()
  return x

@retain
def transform(st):
  return st[-4:]
def main():
  print "Please input your credit card number"
  cc=inputFromUser()
  last4=transform(cc)
  print "CC is [", cc,"]","derived info is [", last4, "]"
  print "Calling erasure"
  erasure(cc)
  print "CC is [", cc,"]","derived info is [", last4, "]"
```

Data return by this function is erasure-aware

The last four characters of the input is not erasure-aware anymore

Erase data

Which policies do we support?

- The primitive `erasure` has to be called explicitly by the programmer: it is part of the program!
- It means that policies are as expressive as the programming language!

```
sensitive_val=raw_input()
ans=raw_input("Do you want to erase?")
if ans=="Yes":
 erasure(sensitive_val)
```

Is it everything that we need?

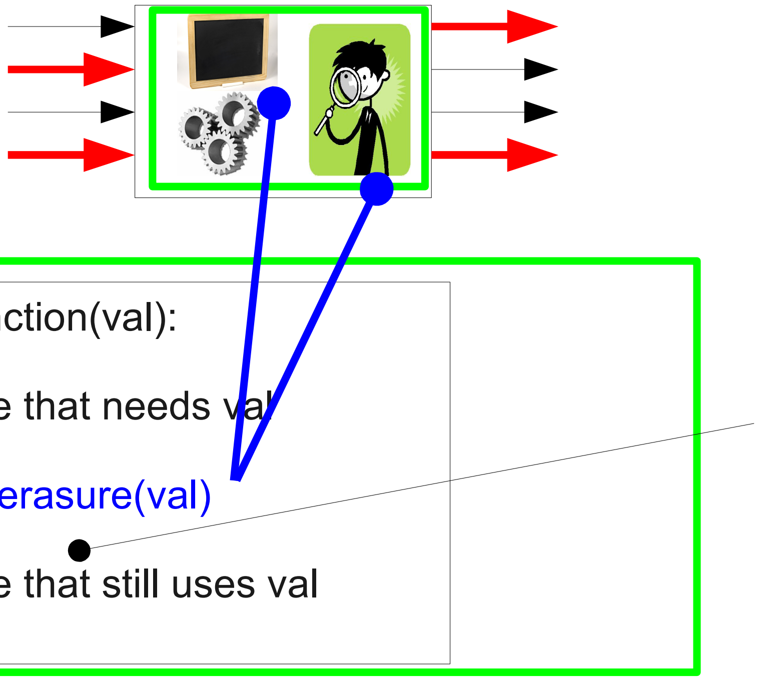
- The policies we can implement with the given API are triggered when `erasure` is executed
- There are other policies that programmers might need and are erasure-specific:
  - "Erase `sensitive_val` in 5 days"
  - "Erase `sensitive_val` if a low privileged user is trying to get the data"
- Previous primitives allow to express those policies, but in an unnatural style. It is better to have an explicit notion for them (**lazy erasure**)

What is lazy erasure about?

- What we want to do is to enforce a “just in time” erasure mechanism
- It is an extension to:
  - Policy language
  - Enforcing technique
- `lazy_erasure` associates objects to policies
- `erasure_escape` annotate functions that may transmit erasure-aware data outside the system in order to check their policies and eventually erase them before it is too late

Lazy API: `lazy_erasure`

- `lazy_erasure` is meant to create an erasure contract that will be used during an “observable action”
- It does not remove the data, but it allows the controlling system to keep track of its propagation



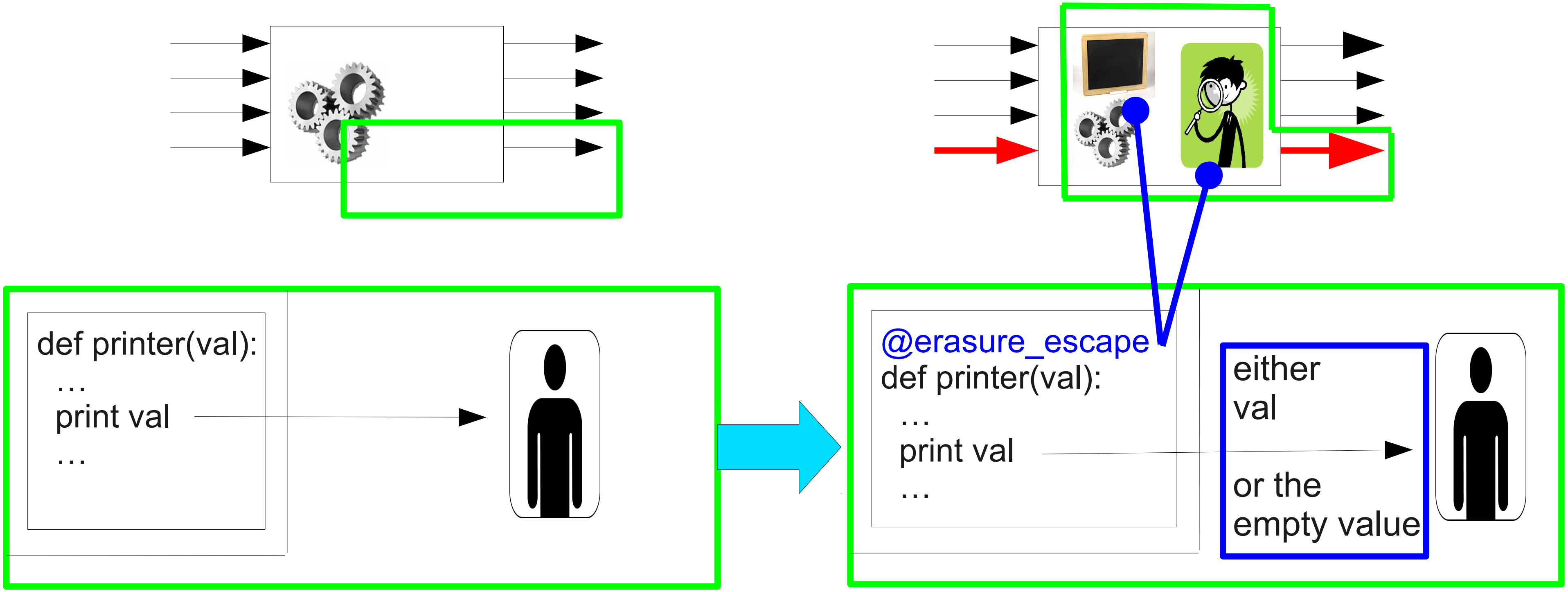
As it happened in the previous example, `val` is an erasure-aware value

```
def function(val):
  …
  #code that needs val
  …
  lazy_erasure(val)
  …
  #code that still uses val
  …
```

Here `val` and all its related info are still available

Lazy API: triggering the policies

- We need to make the system “observationally independent” on the sensitive data
- `erasure_escape` annotates output operations in such a way that erasure-aware data will be erased if their policy evaluates to `true`



```
def printer(val):
  …
  print val
  …
```

```
@erasure_escape
def printer(val):
  …
  print val
  …
```

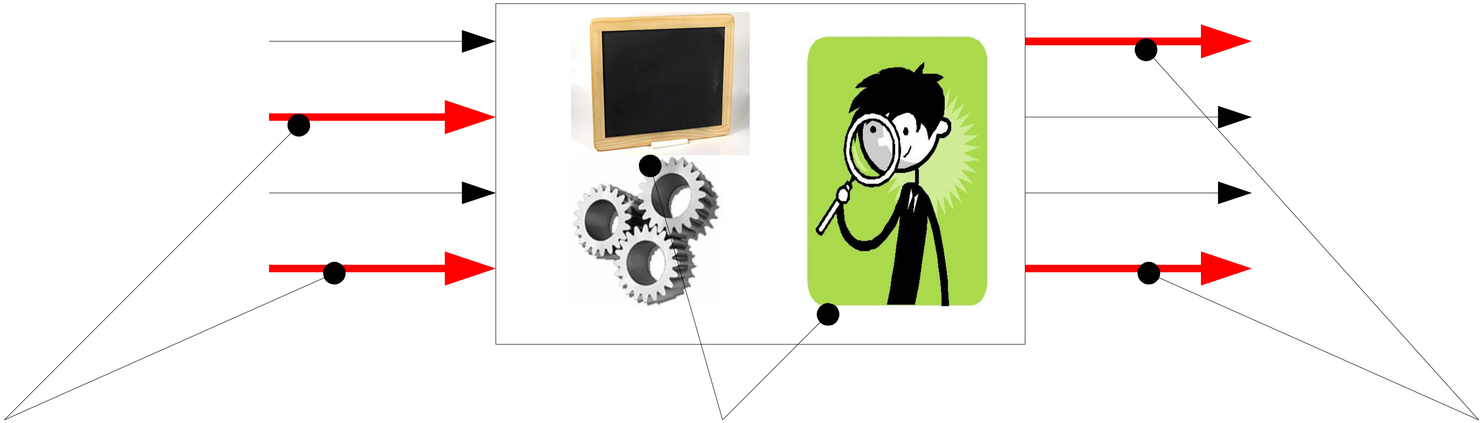
Example

```
from erasure import erasure_source, lazy_era
import time
from datetime import datetime, timedelta

@erasure_source
def inputFromUser():
  x=raw_input()
  return x

def fiveseconds_policy(time):
  return (datetime.today()-time>timedelta(seconds=5))

@erasure_escape
def erasure_channel(a):
  print "The input you provided was [", a, "]"

def main():
  print "Please input your credit card number"
  cc=inputFromUser()
  lazy_erasure(cc,fiveseconds_policy)
  while(1):
    erasure_channel(cc)
    time.sleep(1)
```

The lazy erasure policies are functions on the timestamp of the input data

Observable channel

Recall The Erasure Library



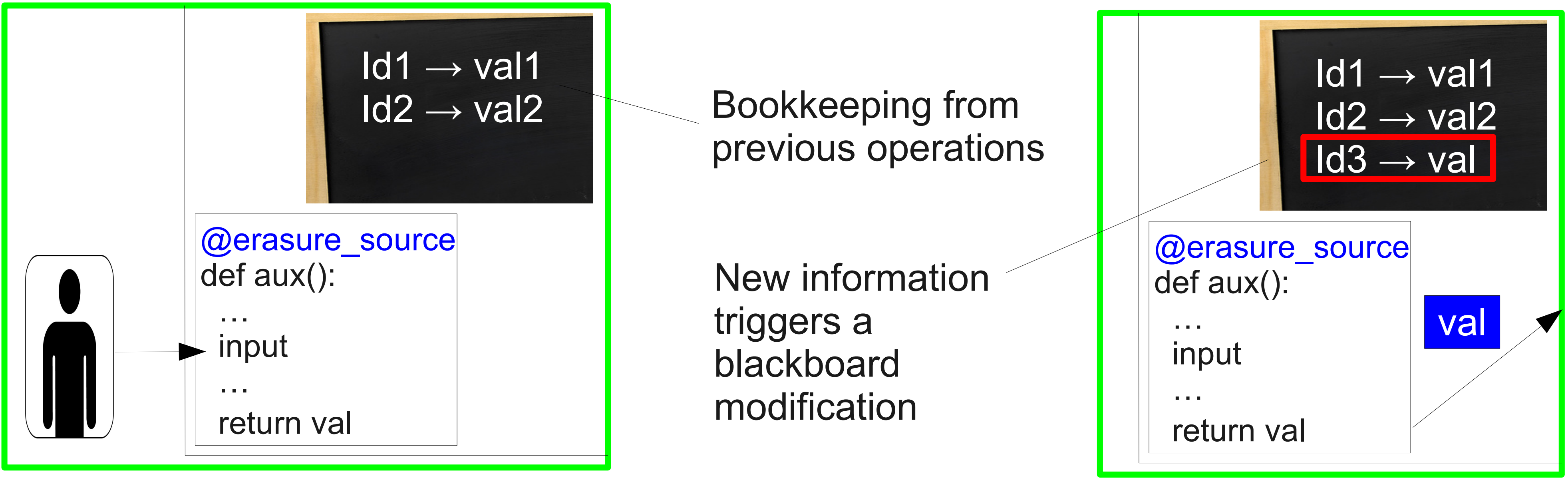
Denote entry points for *erasure-aware* information (sensitive data)

Track the propagation of *erasure-aware* data inside the system.
Implementing the concrete data removal operation

Specify which output actions we need to "observe"

Who is implemented?

- We need to keep track of dependencies among erasure-aware values
- This means we need to identify them uniquely
- The blackboard keeps track of identities



```
@erasure_source
def aux():
  …
  input
  …
  return val
```

Bookkeeping from previous operations

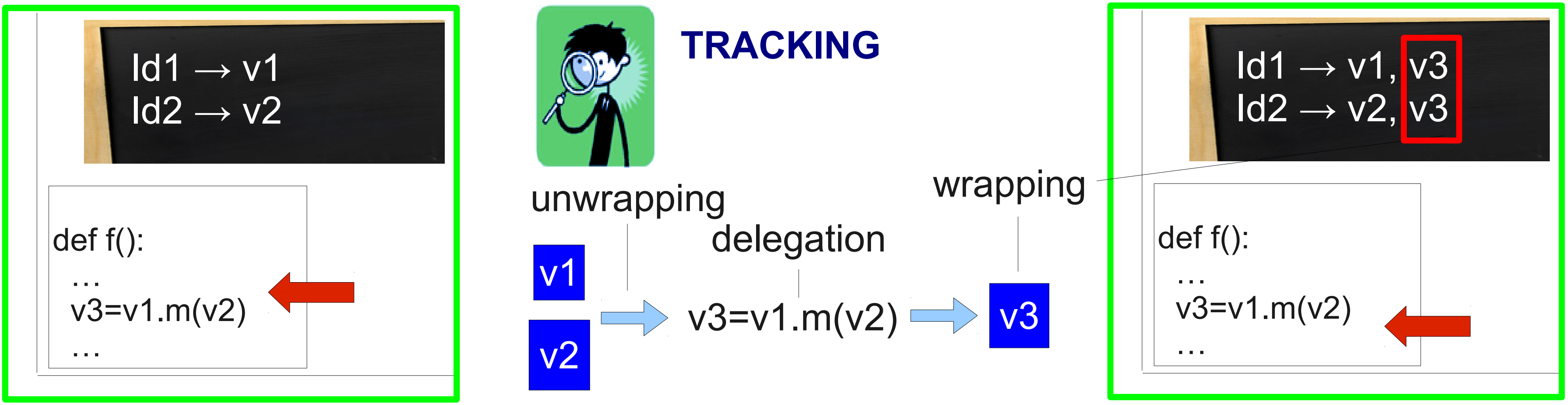
New information triggers a blackboard modification

```
@erasure_source
def aux():
  …
  input
  …
  return val
```

- Identities are time stamps: unique in our sequential implementation and support time-based policies!

Who is ?

- It is the controller (it has two goals)



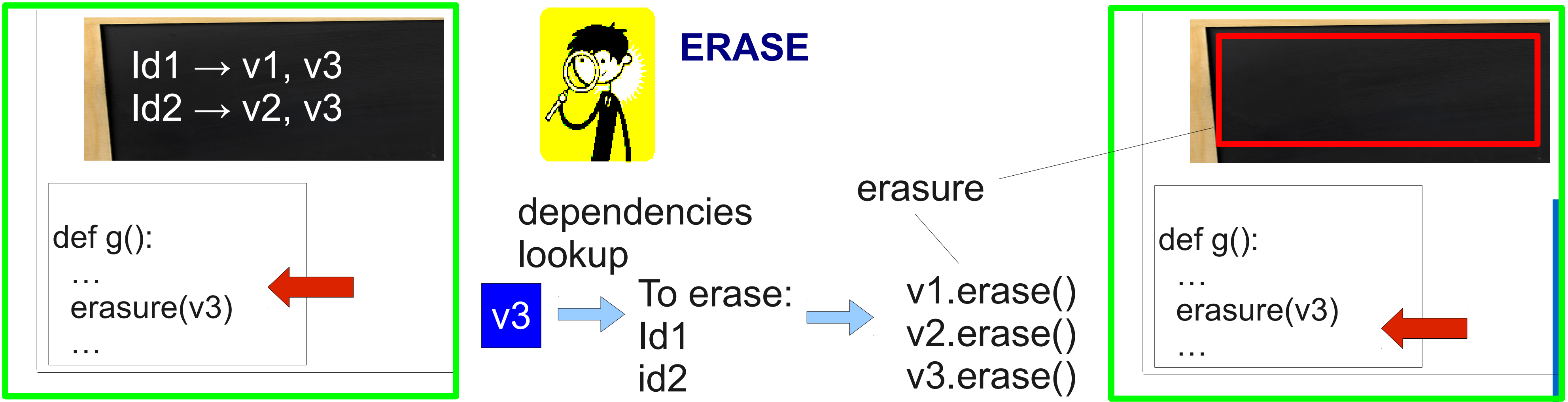
TRACKING

```
Id1 → v1
Id2 → v2
```

```
def f():
  …
  v3=v1.m(v2)
  …
```

```
Id1 → v1, v3
Id2 → v2, v3
```

```
def f():
  …
  v3=v1.m(v2)
  …
```



ERASE

```
Id1 → v1, v3
Id2 → v2, v3
```

```
def g():
  …
  erasure(v3)
  …
```

```
def g():
  …
  erasure(v3)
  …
```

Future work

- On the theoretical side:
  - Which formal guarantees can we prove for our primitives?
- On the practical side:
  - How does the library fit with large existing applications?
  - How do the controller's storage interactions impact on performance?

Conclusion

- Erasure is a property that should be enforced on all systems dealing with sensitive data
- We provided a Python library to get this result for existing code
- The whole library is based on a technique similar to the library for taint-analysis in Python
  - Therefore, it can be applied mostly transparently to existing code
- The approach seems really flexible and promising