Implementing Erasure Policies Using Taint Analysis

Filippo Del Tedesco, Alejandro Russo, and David Sands

Chalmers University of Technology, Göteborg, Sweden
{tedesco,russo,dave}@chalmers.se

Abstract. Security or privacy-critical applications often require access to sensitive information in order to function. But in accordance with the principle of least privilege – or perhaps simply for legal compliance – such applications should not retain said information once it has served its purpose. In such scenarios, the timely disposal of data is known as an *information erasure policy*. This paper studies software-level information erasure policies for the data manipulated by programs. The paper presents a new approach to the enforcement of such policies. We adapt ideas from dynamic taint analysis to track how sensitive data sources propagate through a program and erase them on demand. The method is implemented for Python as a library, with no modifications to the runtime system. The library is easy to use, and allows programmers to indicate information-erasure policies with only minor modifications to their code.

1 Introduction

Sensitive or personal information is routinely required by computer systems for various legitimate tasks: online credit card transaction may handle a card number and related verification data, or a biometric-based authentication system may process a fingerprint. Such systems often operate under informal constraints concerning the handling of sensitive data: once the data has served its purpose, it must not be retained by the system.

The notion of erasure studied here is higher-level than the system-level and physical notions of data erasure which might involve, e.g. ensuring that caches are flushed and that hard-drives are overwritten sufficiently often to eradicate magnetic traces of data. The approach to program-based high-level erasure stems from the work of Chong and Myers [3]. That work and its subsequent developments deal with a notion of erasure which is relative to a multilevel security lattice [7]. For the purpose of this paper, we will not consider this extra dimension – so we view data as either *available* or *erased*.

In this paper, we present a new approach to the enforcement of information-erasure policies on programs which adapts concepts from dynamic taint analysis.

Language-based Erasure Our approach for information-erasure has several key features: it is a purely dynamic mechanism, it is based on taint analysis, and it is realised completely as a Python library. To see the benefits of these features, it is useful to consider previous work on erasure in the context of a simple erasure scenario (one which we will further elaborate upon in Section 3): a fingerprint-activated left-luggage locker of the kind that is increasingly common at US airports and amusement parks. When depositing a bag, a fingerprint scan is recorded. The locker can only be opened with the same fingerprint that locked it. From a privacy perspective, there is a clear motivation for an erasure policy: the fingerprint (and any information derived from it) should be erased once a locker has been reopened.

Hunt and Sands [12] described the first approach to the enforcement of Chong-Myers-style erasure properties, reemphasizing two key features missing from [3]: the ability to associate erasure policies with IO (clearly needed in our erasure scenario), and a way to verify that a program correctly erases data by a purely static analysis (a type system). There are two key limitations in Hunt and Sands's approach. Firstly, in order to obtain a clean semantic model, the authors consider a restricted form of erasure policy which is specified in the code in the form: "the value received at this input statement must be erased by the end of the code block which follows it". This is suitable for the simple locker scenario (which is problematic for other reasons) but unsuitable for more complex conditional policies of the kind discussed by Chong and Myers. Secondly, the idea is only elaborated for a toy language. Scaling up to a real language is a nontrivial task for such a static analysis, and would require, among other things, a full alias analysis.

Chong and Myers [5] independently considered the problem of enforcing erasure policies and developed a hybrid static-dynamic approach. In their approach, data is associated with conditional erasure properties which state that data must be erased at the point when some (in principle arbitrary) condition becomes true. An implementation extending the Jif system uses a simple form of condition variables for this purpose [4]. To support such rich policies, they assume a combination of a static analysis and a runtime monitor. The static analysis ensures that all program variables are labeled with consistent policies. For example, if variable x is copied to y and x's policy says that it should be erased at some condition c, then the policy for y should be at least as demanding. It is then the job of the run-time system to detect when conditions become true, and implement the erasure on the behalf of the programmer (by overwriting all variables with a dummy value).

Neither of these approaches can satisfactorily handle the simple locker scenario (and certainly not the more complex variants we will consider later in this paper). The approach described in [5, 4] does not consider input at all, but only one-time erasure of variables – although this is arguably not a fundamental limitation. More fundamentally, both approaches use a semantic notion of erasure which is based on a strict information-flow property. In the locker scenario described previously, there is a small amount of information which is inevitably "retained" by the system, namely the fact that the fingerprint used to unlock the container matches the one used to lock it. This requirement cannot be easily captured by [5, 12] since no retention of information is allowed. Observe that the retained information is not enough to recover the fingerprint which produced it, and therefore we can consider the system as an erasing one. It is not difficult to imagine more complex scenarios (e.g. billing services) that need to retain portions of sensitive information to complete their task, but the amount of retained data is not enough to consider their behavior as a violation of some required erasure policies. Chong-Myers approach includes declassification, but what we need here is instead an erasure dual, the ability to selectively ignore that some information is remembered by the system. This feature might be called *delimited retention*, as it resembles the *delimited release* [21] property that some non-interferent system may exhibit.

Overview In the remainder of this paper we outline our alternative approach. We adopt the idea of *dynamic taint tracking* which is familiar from languages like Perl [1] and a number of recent pragmatic information-flow analysis tools [10, 13, 14, 8, 22]: a specific piece of data which is scheduled for erasure is labeled ("tainted"). As computation proceeds, the labels are tracked through the system ("taint propagation"). When it is time to erase the data, we can locate all the places to which the data has propagated and thereby erase all of them.

By performing a dynamic analysis, we obtain a system that is able to deal with complex conditional-erasure conditions. Taint analysis does not track all information flows; in particular the information flows which result purely from control-flow are not captured. This makes the approach unsuitable for malicious code (the approach presented in [16] could be integrated with our library in order to tackle such flows). However, when implicit information flows [7] are ignored, then the need for yet-more-complex *delimited retention* policies used at branching instructions seems to be unnecessary. In principle, it could be possible to encode any delimited retention policy using implicit flows at the price of writing complex and unnatural code, which supports the idea to explicitly include mechanisms for delimited retention.

We are able to implement erasure enforcement for Python, an existing widely-used programming language, simply by providing a library, with no modification of the language runtime system and no special purpose compiler needed. Python's dynamic dispatch mechanism is mainly responsible to facilitate the implementation of our approach as a library. It could be then possible to implement our enforcement for other programming languages with similar dynamic features as Python.

The programmer interface to the library does not require the program to be written in a particular style or using particular data structures, so in principle, it can be applied to existing code with minimal modifications.

The API for the library is particularly simple (Section 2) and its implementation builds on two well-known techniques from object-oriented programming and security, namely *delegation* [15] (Section 2.1) and *taint analysis* (Section 2.3). To use the library the programmer must identify *erasure sources* – in the case of the simple locker example, it is the input function which returns the fingerprint. Then, the programmer must mark in the code the point at which a given value must be erased. This allows the library to trace the origins of a given value and erase all its destination values (Section 2.4).

Section 3 illustrates the use of the library with an extended example based on the locker scenario, but with more involved policies.

In addition, we explore a new *lazy* form of erasure (Section 4). This form of erasure is triggered "just in time" at the points where data would otherwise escape the system and observably break the intended erasure policy. The advantage of lazy erasure is that it is able to easily express rich conditional-erasure policies, including those involving time constraints (e.g. "erase credit card numbers more than one week old"). Additional related work is described in Section 5.

2 The Erasure Library

This section presents the library to introduce information erasure policies into programs. Both its source code and the examples we are using in this paper are publicly available at `http://www.cse.chalmers.se/~russo/erasure`.

The library API essentially consists of three functions:

`erasure_source(f)` is used to mark that values produced by function `f` might be erased. Henceforth, we will say that such values are *erasure-aware*. In the locker scenario, suppose that the function responsible to perform the scan of a fingerprint and return its value is `getFingerprint`. Then, the programmer might declare (prior to any computation): `getFingerprint=erasure_source(getFingerprint)`. The instruction above can be interpreted as `t=erasure_source(getFingerprint); getFingerprint=t`, where `t` is a temporary variable. As an alternative, if the code for the definition of `getFingerprint()` is available, Python's decorator syntax can be used to obtain the same effect:

```
@erasure_source
def getFingerprint() :
  # body of definition ...
```

`erasure(v)` erases all erasure-aware data which was directly used in the computation of value `v`. The effect is to overwrite the data with a default value. For example, if a locker is locked with a fingerprint stored in a variable of the same name, then the code for the locked state might be:

```
while locked
  tryprint=getFingerprint()                    # get attempt
  locked=not(match(tryprint,fingerprint))      # unlock?
  erasure(tryprint)                            # erase attempt
erasure(fingerprint)                           # now unlock
```

A simple variant `erasure()` erases all erasure-aware data (strings and numbers), and any data computed from them.

`retain(f)` provides an escape-hatch for erasure. It declares that the result of function `f` does not need to be erased. We say that `f` is a *retainer*. It corresponds to declaring an escape hatch in delimited release, or a sanitisation function in a taint analysis. In the example above, we might declare `match` as a retainer: `match=retain(match)`.

Figure 1 contains two interactive sessions[1]. `>>>` is the interpreter prompt, while `raw_input` is the built-in function that reads a line from the standard input. In the left, line 1 gets the string `'A'` as an input and stores it in variable `x`. Then, variable `x` is used in two elements of list `y`. Naturally, when printing the list, we can observe that the second element is `x` and the third one is some data derived from `x`, i.e. `x` concatenated with itself. Now, let us consider a replay of this session in which the programmer wants to delete the information related to the input `x` after list `y` is printed once, which constitutes an information erasure policy. To achieve that, the programmer needs to import our library, indicate that function `raw_input` returns erase-aware values, and call function `erasure` before printing the list for the second time. This revised session is illustrated on the right of Figure 1. Observe that erasure removes data related to `x`. It

[1] We refer to Python 2.7 here, but our techniques can also be applied to previous releases.

```
1 >>> x=raw_input()
2 A
3 >>> y=['E',x, x+x]
4 >>> y
5 ['E','A','AA']
```

```
1 >>> from erasure import *
2 >>> raw_input=erasure_source(raw_input)
3 >>> x=raw_input()
4 A
5 >>> y=['E',x,x+x]
6 >>> y
7 ['E','A','AA']
8 >>> erasure(x)
9 >>> y
10 ['E','','']
```

Fig. 1. Examples of interactive sessions, without and with erasure

is worth noting that the core part of the program has not drastically changed in order to introduce an information erasure policy. The next subsections provide some insights into the implementation of the library.

2.1 Delegation

```
1 >>> x=Erasure('A')
2 >>> type(x)
3 <type 'instance'>
4 >>> x
5 'A'
6 >>> y=x+x
7 >>> type(y)
8 <type 'instance'>
9 >>> y
10 'AA'
11 >>> x.erase()
12 >>> y.erase()
13 >>> (x,y)
14 ('', '')
```

Fig. 2. Mutable strings

Basic types are immutable in Python, which means they cannot be changed in-place after their creation. For instance, every string operation is defined to produce a new string as a result. Having immutable strings goes against the nature of erasure, since removing information stored in a string implies in-place overwriting of its contents by, for instance, the empty string. By using a coding pattern usually known as *delegation*, the library carefully implements a mechanisms that allows the value of a string to be changed as shown by lines 7 and 10 in Figure 1.

Delegation is a composite-based structure that manages a wrapped object and propagates method calls to it. In our library, it is implemented by the class `Erasure`, which wraps an immutable object. Most of the method calls on that class are forwarded to the wrapped object. The forwarding mechanism assures that the results of method calls are also wrapped by the class `Erasure`. By doing so, the only reference to the wrapped immutable object is by a field on the class `Erasure`. As a result, it is possible to encode mutable strings by simply using delegation. Let us consider the example in Figure 2. Line 1 creates an object of the class `Erasure` that contains the immutable string `'A'`, while line 3 states its type is `instance` and not `str`[2]. Line 6 calls the concatenation method on the object `x`, which is forwarded to the concatenation method of the string `'A'`. The result of that, the immutable string `'AA'`, is wrapped by a new object of the class `Erasure` and stored in `y`. Class `Erasure` provides the method `erase` to perform the concrete action of overwriting, with a default value, the class field where the immutable object is stored (see Lines 11–12). Consequently, the wrapped objects have

[2] For semplicity `Erasure` was defined as an old-style class. For a new-style definition the return value of the `type` operator would be `<class 'erasure.Erasure'>`

now become the empty strings. The previous immutable objects, `'A'` and `'AA'`, are no longer referenced and thus will be garbage collected on due course. Programmers are not supposed to deal with the class `Erasure` directly (observe that it is not in the interface of the library). Determining what data must be wrapped by the class `Erasure` is tightly connected to what information must be erasure-aware. The next two subsections describe the internal use of `Erasure` by the different mechanisms of the library.

2.2 The primitive `erasure_source`

Erasure policies are expected to be only applied on a data source (i.e. an input) [12]. In fact, it does not make too much sense to erase information known at compile-time (e.g. global constants, function declarations, etc). In this light, the library provides the primitive `erasure_source` to indicate those sources of erase-aware values. More technically, the argument of `erasure_source` is a function, and the effect is to wrap, by using the class `Erasure`, the immutable values returned by it. As an example, we have the sequence of commands in Figure 3.

```
1 >>> from erasure import *
2 >>> raw_input=erasure_source(raw_input)
3 >>> x=raw_input()
4 A
5 >>> type(x)
6 <type 'instance'>
```

Fig. 3. Example of using `erasure_source`

Note that lines 1–3 are the same as the ones in Figure 1. In this case, the string `'A'`, returned by calling `raw_input`, is wrapped into an object of the class `Erasure`. As shown in Figure 1, users might want to delete a given input value as well as information computed from it. Therefore, the library must be able to automatically call the method `erase` on a given input as well as any piece of data computed from it. In order to do that, the library keeps track of how erasure-aware values flow inside programs by using taint analysis.

2.3 Taint analysis

Taint analysis is an automatic approach to find vulnerabilities in applications. Intuitively, taint analysis keeps track how tainted (untrustworthy) data flow inside programs in order to constrain data to be untainted (trustworthy), or sanitised, when reaching sensitive sinks (i.e. security critical operations). Perl was the first scripting language to provide taint analysis as a special mode of the interpreter called *taint mode* [2]. Similar to Perl, some interpreters for Ruby [24], PHP [17], and Python [14] have been carefully modified to provide taint modes. Rather than modifying the interpreter, Conti and Russo in [6] show how to provide a taint mode via a library in Python.

There is a clear connection between the use of taint analysis for finding vulnerabilities and the problem of implementing an erasure policy. In taint analysis, data computed from untrustworthy values is tainted. In our library, data that is computed from erasure-aware values is erasure-aware. With this in mind, and inspired by Conti and Russo's work, we implement a mechanism to perform taint propagation, i.e. how to mark as erasure-aware data that is computed from other erasure-aware values. From now on, we use taint and erasure-aware as interchangeable terms.

Let us consider tainting and taint propagation in the following example, which is an extended version of the listing in Figure 1:

```
1  >>> raw_input=erasure_source(raw_input)
2  >>> x=raw_input()
3  A
4  >>> x.tstamps
5  set([datetime.datetime(2010, 7, 3, 14, 13, 49, 21585)])
6  >>> y=raw_input()
7  B
8  >>> y.tstamps
9  set([datetime.datetime(2010, 7, 3, 14, 13, 56, 324137)])
10 >>> z=x+y
11 >>> z.tstamps
12 set([datetime.datetime(2010, 7, 3, 14, 13, 49, 21585), datetime.
     datetime(2010, 7, 3, 14, 13, 56, 324137)])
```

As mentioned previously, erasure policies intrinsically refer to some input in the program. Consequently, to enforce erasure policies, it is necessary to identify specific inputs. Our library associates a timestamp to each input, representing the date and time at which the data was provided. Timestamps are stored in the attribute `tstamps` of the class `Erasure`. Thus, the assignment `f=erasure_source(f)` makes the result of `f` erasure aware, and in addition it ensures that each value produced by `f` is (uniquely) timestamped. Line 5 shows the timestamps corresponding to the input that variable `x` depends on. The content of `x.tstamps` is the date and time when the input in line 3 was provided (2010–7–3 at 14:13:49 and some microseconds).

When erasure-aware values are involved in computations, taint information (i.e. timestamps) gets propagated. More specifically, newly created erasure-aware objects are associated to the set of timestamps obtained by merging the timestamps found in the different objects involved in the computation. Taint propagation is implemented inside the delegation mechanism of the class `Erasure` and it is performed after forwarding method calls for a given object.

Line 10 combines two inputs (`x` and `y`) in order to create a new value, which is stored in `z`. Lines 11–12 show effect of taint propagation, as timestamps associated to `z` are those corresponding to the inputs `x` and `y`. At this point, the reader might wonder why timestamps are used rather than a simple input-event counter. By using timestamps, we will be able to program temporal erasure policies (Section 4).

Explicit and implicit flows On most situations, taint analysis propagates taint information on assignments. Intuitively, when the right-hand side of an assignment uses tainted values, the variable appearing on the left-hand side becomes tainted. In fact, taint analysis is just a mechanism to track explicit flows, i.e. direct flows of information from one variable to another. Taint analysis tends to ignore implicit flows [7], i.e. flows through the control-flow constructs of the language.

```
1 if x == 'A': isA=True
2 else: isA=False
3 erasure(x)
```

Fig. 4. An implicit flow

Figure 4 presents an implicit flow where variable `x` is erasure-aware. Observe that variable `isA` is not erasure-aware. In fact, it is built from untainted Boolean constants. Although the value of `x` is erased (Line 3), information about `x` is still present in the program, i.e. the program knows if `x` referred to `'A'`. It is not difficult to imagine programs that circumvent the taint analysis by copying the content of erasure-aware strings into regular strings by using implicit flows [19]. In scenarios where attackers have full control over the code (e.g. when the code is potentially malicious), implicit flows present an effective way to circumvent the taint analysis. There is a large body of literature on the area of language-based security regarding how to track implicit flows [20]. In this work, we only track explicit flows, and thus our method is only useful for code which is written without malice. Despite the good intentions and experience of programmers, some pieces of code might not perform erasure of information as expected. For example, a programmer might forget to overwrite a variable that is used to temporarily store some sensitive information. In this case, taint analysis certainly helps to repair such errors or omissions. How much information are implicit flows able to retain in non-malicious code? As it has been argued for taint analysis [19], we argue that implicit flows are unlikely to account for a large volume of unintended data retention. The reason is that data retention relies on the non-malicious programmer writing more involved and rather unnatural code in order to, for instance, copy tainted (erasure-aware) strings into untainted ones [19]. In contrast, to produce explicit flows, programmers simply need to forget to remove the content of a variable.

2.4 Erasing data

The taint analysis described above allows the library to determine, given a value, which erasure-aware inputs were used to create it. These inputs are identified by a set of timestamps. To perform erasure, however, the library must take these timestamps and track down all primitive values which are built from those inputs (c.f. line 8 in Figure 1). To track which erasure-aware values depend on which inputs, the library internally maintains a dependency table. It is the interaction of taint analysis and this table what determines one of the differences between our approach and [6]. The table maps each timestamp to the set of (references to) erasure-aware values – i.e. objects of the class `Erasure`. If timestamp t is mapped to objects a and b, it means that the only values in the program created by the input value provided at time t are a and b. The dependency table is extended each time an erasure-aware input value is generated. It is updated when erasure-aware values are formed from already existing ones. Primitive `erasure_source` and the taint propagation mechanism are responsible for properly updating the dependency table. Primitive `erasure(v)`, which performs the actual erasure of data, can be then easily implemented. More precisely, calling `erasure(v)` triggers the method `erase` (recall Figure 2) on all the objects which depend on the timestamps associated to `v`. As a result, erasure-aware values derived from the same inputs as `v` are erased from the program. Similarly, calling `erasure()` triggers the method `erase` on *every* object in the dependency table.

3 Extended Example

To give a fuller illustration of the capabilities of our approach, we add some extra functionalities to the locker system described previously that are likely to be found in a real implementation. Firstly, the system is able to keep track of events in a log that a group of special users, called *administrators*, can fetch using their fingerprints. Secondly, since such lockers are typically found in security-critical public infrastructures, we anticipate that there will be communication with some external authority in order

```
1 def lockerSystem():
2   while(True):
3     print 'Welcome to the locker system'
4     fingerprint=getFingerprint()
5     ts=datetime.today()
6     if fingerprint in ADM:
7       log.add('MEMORY DUMP -->'+fingerprint+': '+str(ts))
8       dump(log.getLog())
9     else:
10      suspect=local_police.check(fingerprint)
11      h = hash(fingerprint)
12      if locker.isFree():
13        key = h
14        locker.occupied()
15        print 'Please, do not forget to retrive your goods'
16        log.add('LOCKED -->'+fingerprint+': '+str(ts))
17      else:
18        if key == h:
19          locker.free()
20          print 'Thanks for using the service'
21          log.add('UNLOCKED -->'+fingerprint+': '+str(ts))
22        else:
23          print 'You are not the right owner'
24          log.add('INVALID ACCESS -->'+fingerprint+': '+str(ts))
```

Fig. 5. Locker system

to cross-check the input fingerprints with the ones contained in special records (terrorist suspects, wanted criminals etc.). For simplicity, and without losing generality, we consider a system connected to just a single locker rather than several ones.

The code in Figure 5 shows an implementation of the locker system. As before, function `getFingerprint` reads a fingerprint. Function `datetime.today` returns a timestamp representing the current date and time. Object `log` implements logging facilities. Method `log.add` inserts a line into the log and method `log.getLog` provides the log back inside a container.

When the fingerprint matches one of the administrator's fingerprints stored in the container `ADM`, the `dump` function is executed using `log.getLog` as argument, and the log is output (lines 7-8). Object `local_police` represents a connection to the external authority. Method `local_police.check` cross-checks the fingerprint given as an argument against a database of suspects.

In all other cases (i.e. for locking and opening purposes), the locker only needs a `hash` of the fingerprint, which is assigned to `h`. `locker` represents the state of the locker, which is initially "free" and could become "occupied" during the execution. If the fingerprint does not belong to an administrator, the locker is tested with the `isFree` method. If the answer is positive, the user can store luggage; the hash is then saved in `key` and the locker state is set to `occupied` (lines 13-16). Otherwise, the locker is full and it is released only if the current hash matches with the one used to lock it. In this case the method `free` makes the locker available for the next user (lines 19-21).

When it comes to logging, it is crucial to define what we want and is allowed to log. The program logs four different responses corresponding to the system usage: *'LOCKED'*, *'UNLOCKED'*, *'INVALID ACCESS'*, and *'MEMORY DUMP'*. Naturally, it is important to register the actions performed by the system as well as the time when they occur. Clearly, information erasure emerges as a desirable property when it comes to handle fingerprints. On one hand, *fingerprints corresponding to regular users must be removed from the system (including from its log) after they are used for the intended purpose*, which constitutes the information erasure policy of the locker system (observe the hash of the fingerprint is stored in the system for the authentication purpose, and for the purposes of this example is considered to be OK to store). Fingerprints corresponding to suspects, on the other hand, can be logged as evidence in case of a police investigation. In order to give credit for his or her work, fingerprints from administrators can also be logged. In other words, fingerprints from regular users must be erased after using them, while fingerprints from suspects and administrators can remain in the system. The code shown in Figure 5 does not fulfill the information erasure policy described before. It actually logs the fingerprints of any user, which violates citizens privacy. Although it is relatively simple to detect the violation of the information erasure policy in this example, the same task could be very challenging in a more complex system where there could be multiple sources of sensitive information in several thousands lines of codes.

```
1  from erasure import erasure_source, retain, erasure
2  
3  # Erasure-aware sources
4  getFingerprint=erasure_source(getFingerprint)
5  
6  # Retention statement
7  hash=retain(hash)
8  
9  def lockerSystem():
10       ...
11       suspect=local_police.check(fingerprint)
12       h = hash(fingerprint)
13       if not(suspect):
14         erasure(fingerprint)
15       ...
```

Fig. 6. Locker system patched to fulfill the erasure policy regarding fingerprints

Figure 6 shows how programmers can use the library to make the code fulfill the erasure policy regarding fingerprints. Line 1 imports our library. Line 4 identifies that fingerprints are subjected to erasure policies, i.e. they are erasure-aware values. Line 7 states that `hash` is properly written, namely its outputs cannot be related to its input, and therefore they are not considered to violate any erasure policy. Then, the implementation of function `lockerSystem` is only changed to call `erasure` when the user of the locker is not a suspect (lines 13-14). The rest of the code remains unchanged.

4 Lazy Erasure

The notion of erasure presented in the previous section is very intuitive. To remove all erasure-aware inputs used to compute a given value `v`, it is enough to call `erasure(v)`.

When calling `erasure`, the library immediately triggers the mechanism to perform erasure over the current state of the program. Due to that fact, we call the mechanism implemented by the API in Section 2 *eager erasure*[3].

Eager erasure does not easily capture some classes of erasure policies without major encoding overhead, which might drastically modify the code of the program. In particular, let us consider conditional policies that cannot be immediately decided, e.g. a certain value can only remain in the system for a period of time, after which it has to be erased. Clearly, it is not possible to trigger the erasure mechanism straight away, but the need for erasure has to be remembered in the system and triggered at the right time. To deal with such policies without any additional major runtime infrastructure, the library provides *lazy erasure* as a mechanism to perform erasure at the latest possible moment, i.e. when needed.

Lazy erasure deletes information "just in time" at the points where data would otherwise escape the system and observably break the intended erasure policy. Programmers only need to state what is supposed to be erased and the library triggers the erasure mechanisms at certain output points, i.e. when information is leaving the system.

4.1 The Lazy Erasure API

Lazy erasure adds some additional functions to the API of the library. The other primitives such as `erasure_source` have the same semantics as before.

`erasure_escape(f)` This function is used syntactically in the same manner as `erasure_source` – i.e. as a function wrapper. It is used to identify the functions which are to be considered as "outputs" for the system. These are the functions where an erasure policy could be observable violated – for example writing to a file or communicating with the outside world in some other manner. The lazy erasure policies are enforced by inspecting the arguments to the functions which have been wrapped by the primitive `erasure_escape`.

`lazy_erasure(v,p)` Primitive `lazy_erasure` introduces an erasure policy into the program, but does not perform any actual erasure of information. It receives as arguments a value `v` and a policy function `p`. The policy function (henceforth an *erasure policy*) is a function from timestamps (i.e. timestamps of inputs) to Boolean values. Internally, a policy can use any of the program state, together with the timestamp argument (representing the timestamp of the value to be erased) to make judgment on whether the value should be erased or not. Thus, declaring `lazy_erasure(v,p)` indicates that any input values (and values computed from them) which were used in the creation of `v` should be erased if policy `p` holds for their timestamps. Erasure is then enforced at the output functions indicated by `erasure_escape`.

Two abbreviations are supported: `lazy_erasure(v)`, which is equivalent to `lazy_erasure(v,(lambda t:True))` and thus unconditionally enforces erasure at the erasure-escape points, and `lazy_erasure(p)`, which is an abbreviation for calling `lazy_erasure` with the policy `p` applied to every erasure-aware value in the system.

[3] In functional languages, eager and lazy evaluation are commonly used terms to indicate when evaluation is performed. We use the same terminology for erasure of data rather than evaluation of terms.

```
1  from datetime import datetime, timedelta
2  from erasure import *
3
4  getFingerprint=erasure_source(getFingerprint)
5
6  hash=retain(hash)
7
8  dump=erasure_escape(dump)
9
10 lazy_erasure(fivedays_policy)
11
12 def lockerSystem():
13       ...
```

Fig. 7. Locker system with a lazy erasure policy

4.2 Lazy Erasure Examples

To illustrate how lazy erasure works, we start by encoding a temporal erasure policy that allows to only keep fingerprints (administrators and suspects' ones) for a limited time of five days. The following piece of code implements the condition for such a policy:

```
def fivedays_policy(time):
    return (datetime.today()-time)>timedelta(days=5)
```

Policy `fivedays_policy` takes a timestamp as input and returns whether the timestamp is more than five days old. In Figure 7, we show how to apply the policy in our locker system. Line 8 indicates that before extracting the log from the system, erasure must be performed. Line 10 introduces the erasure policy `fivedays_policy` into the system. As a result, dumping the log triggers erasure on each of its entries which are older than 5 days.

Lazy erasure is particularly useful to express policies that cannot be immediately decided when input data enters the system. To illustrate this, we extend the locker scenario a bit further. A common experience with network connections is the loss of connectivity. To handle this situation properly, we introduce the constant *'no_connection'* to be returned by method `local_police.check` when the connection with the police department cannot be established. Enforcing an erasure policy that depends on the connection to the police department is not as simple as the policies considered previously. On one hand, we would like to have in the log the fingerprints which got the *'no_connection'* answer since they could belong to suspects. On the other hand, fingerprints that got the *'no_connection'* answer and do not belong to suspects must be erased in order to avoid violating users privacy when administrators dump the log.

As a trade-off between preserving fingerprints of suspects and privacy of regular citizens is represented by the enforcement of an erasure policy which depends on the person doing the dumping of the log. If a police agent is included in the set of administrators, then he or she can dump the log if necessary. Since a police agent represents the public authority, the agent has full access to the fingerprints stored in the log. Therefore, all the entries are included in the log, including those ones with the *'no_connection'* answer. In contrast, if the dumper is a regular administrator, the entries with *'no_connection'* are removed from the log. In this way, suspect-related

```
1  def lockerSystem():
2      global police_mode
3      police_mode=False
4      ...
5      if fingerprint in ADM:
6        log.add('MEMORY DUMP -->'+fingerprint+': '+str(ts))
7        if fingerprint=='police':
8          police_mode=True
9        dump(log.getLog())
10       police_mode=False
11     else:
12       suspect=local_police.check(fingerprint)
13       h = hash(fingerprint)
14       if suspect==False:
15         lazy_erasure(fingerprint)
16       elif r=='no_connection':
17         lazy_erasure(fingerprint,role_policy)
18       else:
19         pass
20     ...
```

Fig. 8. `lockerSystem` reimplemented for lazy erasure

data may get lost but privacy is not compromised. Clearly, the erasure policy is more involved than the ones that we have been considered so far. However, we show that it can be easily encoded by our library.

```
1  def role_policy(time):
2    global police_mode
3    return not(police_mode)
```

Fig. 9. Example of a lazy policy based on roles

We start by introducing the Boolean global variable `police_mode` to represent when a police agent is dumping the log. Then, the function `lockerSystem` has to signal whether the person dumping the log is the police agent. Figure 8 shows an extension to `lockerSystem`. In line 3, `police_mode` is initially set to `False`. Immediately before dumping the log (line 9), the administrator identity is checked. If it is a police agent, `police_mode` is set to `True` (line 8). The state is then reset at line 10. If the person dumping the log is a regular administrator, the value of `police_mode` does not change. Observe that line 17 associates the erasure policy `role_policy` to those fingerprints received when the connection to the policy department cannot be established. Consequently, the erasure of the fingerprint depends on the value returned by the policy *at the time of dumping the log*. Figure 9 defines `role_policy`. This policy only returns true when the dumping is done by a regular administrator (line 3). As a consequence, those fingerprints associated with `'no_connection'` are erased immediately before dumping the log provided that `police_mode` is false.

5 Related Work

As we have already explained in the introduction, application level erasure has been studied in [3] and [12]. A simpler form of erasure for Java bytecode is discussed in [11]. In [23], the counterpart of erasing systems (according to the definition given in [12]) has been explored, providing some insights into the obligations of a user who interacts with a system which promises erasure. These works all deal with an attacker model where an attacker can in the worst case inject arbitrary code into the system at a point in time at which erasure is supposed to have occurred. At lower levels of abstraction, for example [9], conditions and techniques to guarantee physical erasure on storage devices are considered. The need for physical erasure comes from a much stronger attacker model where the attacker is not hindered by any abstraction layers. An end-to-end view linking the high-level application level and the low level physical views should be possible, but it has not been previously considered.

To the best of our knowledge, Jif$_E$ [4] is the only system currently implementing application-level erasure. This is based on the Jif compiler which deals with a subset of Java extended with security labels. Unlike the very general model on which it is based [5], the only conditions allowed in Jif$_E$'s conditional erasure policies are a special class of Boolean condition variables. The implementation ensures that whenever such a condition variable changes, any necessary erasures are triggered. It would be simple to mimic this style of implementation (modulo implicit flows) using our primitives.

Erasure can be also related to *usage control*, since it is based on the idea of changing the way data is handled in the system after a certain moment. In [18], the authors present a model to reason on usage control, based on *obligations* the data receiver has to enforce through some *mechanisms*. The model is very general, and erasure can be described as an obligation (actually it is explicitly mentioned as a data owner requirement), but its purpose does not correspond to our approach, which deals with techniques to implement that obligation. The work in [26] extends access control with temporal and times-consuming features, leading to what they call TUCON (Times-based Usage Control) model. This approach allows to reason with policies that deal with the period of time in which a given object is available. Although it would not be very natural (policies here seem to be more user-oriented), it should be also possible to reason about erasure in this framework as well; similar considerations about implementation holds in this case as well. However, concepts from the usage control literature could provide inspiration for a study of the enforcement of a wider class of usage policies at code level.

6 Conclusions and Future Work

We have presented a library-approach to enforce erasure policies. The library transparently adds taint tracking to data sources, making it easy to use and permitting programmers to indicate information-erasure policies with only minor modifications to their code. To the best of our knowledge, this is the first implementation of a library that connects taint analysis and information-erasure policies. From our limited experience, the imperfections of taint analysis (the inability to track implicit flows) serve to keep the policy specifications simple, and enable us to handle examples for which existing approaches would not be sufficiently expressive. We have also introduced the concept of lazy erasure – an observational form of erasure which supports richer erasure policies, including temporal policies, with a simple implementation.

There are a number of directions for further work. One challenge ahead is how to deal with permanent storage like databases or file systems when specifying erasure policies. Policies like “user information must be erased when his or her account is closed” are out of scope in the existing approaches [3, 12], where erasure is performed on internal data structures. User information, on the other hand, is usually placed in databases (e.g. web application) or file systems (e.g. Unix-like operating systems). We believe that it is possible to extend the interfaces for accessing files and databases in order to store data as well as erasure information (timestamps). Those interfaces usually involve handling objects and thus the library needs to be extended to consider them. To achieve that, we could threat objects as just mere containers and apply similar tainting techniques as the ones used for dictionaries. Another important aspect is the evaluation of the overheads caused by the library – in particular, how taint propagation and updates in the dependency table impact on performance. It would also be interesting to evaluate how *precise tainting* [17, 8] could be exploited to obtain more precision when erasing data. Precise tainting associates taint information to characters rather than to whole strings. In our library, if an small part of an string contains some information that should be erased, then the whole string is deleted. By using *precise tainting*, it would be possible, in principle, to only delete those pieces of the string containing the information to erase. Precise tainting usually requires to fully understand the semantics of each function that manipulates erasure-aware values. As for most approaches to dynamic taint analysis, our approach ignores implicit flows. As a consequence programs might retain information indirectly via their control constructs. Rather than fixing this problem, a reasonable alternative might be to bound it. Inspired by preserving confidentiality, the work in [16] develops a mechanism to obtain bounds on the information leaked by implicit-flows. We believe that it is feasible to adapt such mechanism to obtain bounds on the information retained by control constructs. On the theoretical side, it could be important to describe precisely the security condition that taint analysis is enforcing in the presence of delimited retention policies. In fact, to the best of our knowledge, the work by [25] is the only one that presents a security condition for taint analysis using formal semantics.

Acknowledgements Thanks are due to Juan José Conti for his contributions during the library development, to Aslan Askarov for suggesting the luggage-locker example and to our colleagues of the Prosec group, who shared with us their impressions on the topic. This work was partially supported by VR (`vr.se`), SSF (`stratresearch.com`) and the EU FP7 WebSand project.

References

1. The Perl programming language. `http://www.perl.org/`
2. Bekman, S., Cholet, E.: Practical mod_perl. O'Reilly and Associates (2003)
3. Chong, S., Myers, A.C.: Language-based information erasure. In: Proc. IEEE Computer Security Foundations Workshop. pp. 241–254 (Jun 2005)
4. Chong, S.: Expressive and Enforceable Information Security Policies. Ph.D. thesis, Cornell University (Aug 2008)
5. Chong, S., Myers, A.C.: End-to-end enforcement of erasure and declassification. In: CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium. pp. 98–111. IEEE Computer Society, Washington, DC, USA (2008)
6. Conti, J.J., Russo, A.: A taint mode for python via a library. OWASP AppSec Research (2010)
7. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Comm. of the ACM 20(7), 504–513 (Jul 1977)
8. Futoransky, A., Gutesman, E., Waissbein, A.: A dynamic technique for enhancing the security and privacy of web applications. In: Black Hat USA Briefings (Aug 2007)

9. Gutmann, P.: Data remanence in semiconductor devices. In: SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium. pp. 4–4. USENIX Association, Berkeley, CA, USA (2001)
10. Haldar, V., Chandra, D., Franz, M.: Dynamic Taint Propagation for Java. In: Proceedings of the 21st Annual Computer Security Applications Conference. pp. 303–311 (2005)
11. Hansen, R.R., Probst, C.W.: Non-interference and erasure policies for java card bytecode. In: 6th International Workshop on Issues in the Theory of Security (WITS '06) (2006)
12. Hunt, S., Sands, D.: Just forget it – the semantics and enforcement of information erasure. In: Programming Languages and Systems. 17th European Symposium on Programming, ESOP 2008. pp. 239–253. No. 4960 in LNCS, Springer Verlag (2008)
13. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In: 2006 IEEE Symposium on Security and Privacy. pp. 258–263. IEEE Computer Society (2006)
14. Kozlov, D., Petukhov, A.: Implementation of Tainted Mode approach to finding security vulnerabilities for Python technology. In: Proc. of Young Researchers' Colloquium on Software Engineering (SYRCoSE) (Jun 2007)
15. Lutz, M.: Learning Python. O'Reilly & Associates, Inc., Sebastopol, CA, USA (2003)
16. Newsome, J., McCamant, S., Song, D.: Measuring channel capacity to distinguish undue influence. In: PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security. pp. 73–85. ACM (2009)
17. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., Evans, D.: Automatically Hardening Web Applications Using Precise Tainting. In: In 20th IFIP International Information Security Conference. pp. 372–382 (2005)
18. Pretschner, A., Hilty, M., Basin, D., Schaefer, C., Walter, T.: Mechanisms for usage control. In: ASIACCS '08: Proceedings of the 2008 ACM symposium on Information, computer and communications security. pp. 240–244. ACM, New York, NY, USA (2008)
19. Russo, A., Sabelfeld, A., Li, K.: Implicit flows in malicious and nonmalicious code. 2009 Marktoberdorf Summer School (IOS Press) (2009)
20. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Selected Areas in Communications 21(1), 5–19 (Jan 2003)
21. Sabelfeld, A., Myers, A.C.: A model for delimited information release. In: Proc. International Symp. on Software Security (ISSS'03). LNCS, vol. 3233, pp. 174–191. Springer-Verlag (Oct 2004)
22. Seo, J., Lam, M.S.: InvisiType: Object-Oriented Security Policies. In: 17th Annual Network and Distributed System Security Symposium. Internet Society (ISOC) (Feb 2010)
23. Tedesco, F.D., Sands, D.: A user model for information erasure. In: SecCo'09, 7th International Workshop on Security Issues in Concurrency. Electronic Proceedings in Theoretical Computer Science (2009), to appear
24. Thomas, D., Fowler, C., Hunt, A.: Programming Ruby. The Pragmatic Programmer's Guide. Pragmatic Programmers (2004)
25. Volpano, D.: Safety versus secrecy. In: Proc. Symp. on Static Analysis. LNCS, vol. 1694, pp. 303–311. Springer-Verlag (Sep 1999)
26. Zhao, B., Sandhu, R., Zhang, X., Qin, X.: Towards a times-based usage control model. In: Proceedings of the 21st annual IFIP WG 11.3 working conference on Data and applications security. pp. 227–242. Springer-Verlag, Berlin, Heidelberg (2007)