A User Model for Information Erasure

Filippo Del Tedesco
Chalmers University of Technology
Gothenburg, Sweden
tedesco@chalmers.se

David Sands
Chalmers University of Technology
Gothenburg, Sweden
dave@chalmers.se

Hunt and Sands (ESOP'08) studied a notion of *information erasure* for systems which receive secrets intended for limited-time use. Erasure demands that once a secret has fulfilled its purpose the subsequent behaviour of the system should reveal no information about the erased data. In this paper we address a shortcoming in that work: for erasure to be possible the user who provides data must also play his part, but previously that role was only specified informally. Here we provide a formal model of the user and a collection of requirements called *erasure friendliness*. We prove that an erasure-friendly user can be composed with an erasing system (in the sense of Hunt and Sands) to obtain a combined system which is *jointly erasing* in an appropriate sense. In doing so we identify stronger requirements on the user than those informally described in the previous work.

1 Introduction

The requirement that data is used but not retained is commonplace. As an everyday example consider the credit card details provided by a user to a payment system. The expectation is that card details will be used to authorize payment, but will not be retained by the system once the transaction is complete.

The study of erasure policies from a language-based security perspective was initiated by Chong and Myers [2]. Hunt and Sands [5] argue that to give a satisfactory account of erasure, one needs to consider *interactive* systems: the card details are used in the interaction between the customer, the payment system and the bank, and *then* erased; without the interaction, the card details could be dispensed with altogether and erasure would be unnecessary. They present an information-flow based definition of erasure for sequential programs interacting with users through channels, and a type system for guaranteeing erasure properties.

This paper deals with the model of erasure described by Hunt and Sands, and addresses a shortcoming of that work: despite the emphasis on an interactive view of erasure, there is no explicit model of the users of the program. As a consequence certain requirements about the users are described but not formalized. More specifically, users who supply data for erasure are expected to fulfill certain obligations. These obligations are intended to complement the erasure property fulfilled by the program so that together users and the system truly achieve their data erasure goals. Since users and their obligations are not modeled, the previous work is unable to prove this. The main result of this paper is to formalize the obligations of what we will call an *erasure friendly* user, and to show that an erasure friendly user when composed with an erasing system satisfies *composite erasure*, which defines a joint responsibility between user and system in the erasure mechanism. As a result of the formalization of the user we discovered additional obligations required on erasure friendly users which were not described in [5].

In the remainder of this introduction we outline the basic ideas in the definition of an erasing program from [5], and the informal obligations identified for the user of such a program. Finally we outline remainder of the paper.

EPTCS ??, 20??, pp. 1–15, doi:10.4204/EPTCS.??.??

Erasure: the basic idea For the purposes of this paper we will work with a simplification of the erasure notion from [5]. In that work (following [2]) data is labelled using a multilevel security lattice and erasure is from one level to some higher level.

Here we will ignore the multilevel security issues and assume data from one specific user (level), which are requested and erased completely by the system through the following programming construct **input** x **erased in** C.

Operationally this behaves just as an input statement from the user which writes into a variable x, after which the command C is executed. But the intention is that this *specifies an erasure policy*, namely that the data input will not be used beyond the command C. If this property holds for all runs and all erasure blocks then the program is said to be *input erasing*. To make this notion precise a noninterference-style definition is used: suppose that a given computation reaches a statement of the form **input** x **erased in** C, and the user provides an input value u for x, the computation reaches the end of the command C and after this point the program has some observable behavior t. The erasure requirement is that if the user had instead provided some different value v, then the program would still be able to reach the end of the command C and produce the same observable behavior t. This ensures that after the end of the erasure block the observer can learn nothing about the user's input by analyzing the system behavior. Then the definition has also to prevent the secret provider being exploited as an involuntary storage for the secret, requiring that its behavior after C has to be the same independently of the value it has sent. Considering a simple model for user (for example a list of values, like in [5]), this is tantamount requiring that the number of inputs provided by the user to the system has to be the same for all possible C executions.

As an example consider the pseudocode in Figure 1 which depicts a credit card transaction with input from the user. The intention is that the credit card is erased after the completion of the transaction, and this is reflected in the reassignment of the credit card variable on line 8. In fact the credit card is *not* erased correctly here. When the server goes down information about the credit card is retained via the variable `payment`, which is outputted to the log file in the last statement of the program. To make the program input erasing one must additionally overwrite `payment` at the end of the while loop.

Note that there are two kinds of inputs from the user: those subject to erasure, and those (like the input of the shipping address) which are not. Additionally, outputs during the erasure block may contain information that is the subject of the erasure, as for example the echoing of the order information on line 4.

```
1  while serverUp {
2      input creditCard erased in {
3          input shippingAddress;
4          output (creditCard++shippingAddress) to user;
5          payment := process(creditCard);
6          output payment to bank
7          custDatabase := (custDatabase ++ shippingAddress);
8          creditCard := 0
9      }
10 };
11 output all variables to logfile;
```

Figure 1: Example program

User Obligations Certain user obligations and assumptions are implicitly built into the notion of input erasure described above. One is that we clearly cannot expect erasure to "work" for an arbitrary user. For example, if a user adopts the strategy of always appending his credit card number to his shipping address then since the shipping address is included in the customer database the system would inadvertently save the credit card number and the combined system would not be erasing as intended.

A second example from [5] reveals more assumptions about the user: suppose that, before the credit card is erased, the program provides the user with a special offer code with the promise "present this code when you next shop with us for a 10% discount". What if this code is simply an encryption of the credit card number? Although it may be reasonable to assume that the user does not exhibit the deliberately bad strategy described above, how do we ensure that the user does not re-input this code after the transaction (since re-inputing the code will enable the program to recover the credit card number)? Hunt and Sands argue that, in contrast with noninterference, it is *not* reasonable for the user to know the semantics of the system and be able to make perfect deductions about what is safe. It was proposed that the user strategy:

- assumes that the user knows that certain data is scheduled for erasure, and they are notified when the erasure is complete, and
- assumes that the user treats any output from the program as potentially tainted with data currently scheduled for erasure.

The question we answer in this paper is whether this user strategy is indeed sufficient to ensure the desired erasure property.

Outline The approach we take to answer these questions is as follows. The first step (Section 2) is to recall the notion of input erasure for a program (what we will henceforth call the *system*). We deviate from the original formulation by representing this in a syntax-independent form. To do this we make the beginning and end of an erasure session externally visible as communication events which declare, respectively, that the next input from the user will later be erased, and that the input has been erased. We refer to the input erasure property of a system S as $E(S)$.

The second step (Section 3) is to define the interface and structure of a user, U, who will provide the data for erasure. We then specify both the communication mechanism between U and S as well as a number of semantic constraints on user behavior which define when U is what we call *erasure friendly*, $EF(U)$.

Finally (Section 4) we define the desired erasure property, *composite erasure*, for the combined system $EC(U|S)$. We are then able to prove the main theorem, namely that if we combine an input-erasing system with an erasure-friendly user then we obtain a combined system which is erasing:

$$E(S) \;\&\; EF(U) \Rightarrow EC(U|S)$$

Related works are discussed in Section 5, and conclusions and further work are outlined in Section 6.

2 Systems and Abstract Input Erasure

In the previous approach systems were defined through a deterministic imperative language, which was equipped with input-output primitives. As we have already seen in the Example 1, for erasure the most important one is the block structured input command, where the value received through the input operation must be erased at the end of the block. In this section we introduce our system model and corresponding erasure definition. In some respects it is more general and abstract and in other respects it is simplified when compared to the earlier work.

- The generalization comes from handling a model which is independent of a particular programming language syntax. This is possible because we are focussing on the interaction between user and system, rather than the *verification* of the erasure property of systems. What we retain from the earlier model is the assumption of determinism, and a notion of *well-formedness* which includes an abstract counterpart to the use of a block-structured erasure construct.
- The simplification here is that (i) we focus on erasure of data from a single user, and (ii) we assume that the information is totally erased, rather than the more general case of being erased to a higher security level. Because of these simplifications we no longer need to model a multilevel security lattice. Instead we simply assume that there is one other user of the system whose security level is different from the user supplying the data to be erased.

Since we will model erasure properties in terms of inputs and outputs we are not interested in the representation of system's internal state. Thus we are able to describe it through a labelled transition system $S = (\mathscr{S}, \mathscr{L}, \mathscr{T})$, in which the components have the following meaning:

- $\mathscr{S} = \{s_i : i \text{ is an index}\}$ is the set of (distinct) states, in which s_0 is the initial one. The counterpart of this element in the programming language approach was the entire code for the program, together with the empty memory.
- $\mathscr{L} = \{a!v, a?v, a!BE, a!EE, b!v\}$ is the set of labels for S: the first four are used for the communication with the user U through channel a. The other represents an output event on a different channel b. While v represents an ordinary value received or sent during an I/O interaction (which is a member of the domain of values $\mathbb{V}$), BE and EE are considered reserved values, which mark the beginning and end of an erasure "block".
- $\mathscr{T} \subseteq \mathscr{S} \times \mathscr{L} \times \mathscr{S}$ is the set of transactions such that if a pair of values v, s_v exists such that $(s, a?v, s_v) \in \mathscr{T}$, then for each $w \in \mathbb{V}$ a state s_w exists such that $(s, a?w, s_w) \in \mathscr{T}$. Basically we require that S is an *input enabled* system, ready to accept each possible value offered on channel a.

We need to impose some further restrictions on the behavior of the system, the first of which is determinism; we take the definition from [4].

Definition 2.1 (Deterministic System)**.** *A system S is deterministic if:*

- *whenever* $s \xrightarrow{l_1} s_1$ *and* $s \xrightarrow{l_2} s_2$ *and* $l_1 \neq l_2$ *then* $l_1 = a?v_1$ *and* $l_2 = a?v_2$ *for some values* v_1, v_2.
- *If* $s \xrightarrow{l} s_1$ *and* $s \xrightarrow{l} s_2$ *then* $s_1 = s_2$.

Now we can proceed towards the erasure part of system specification. Since this notion is closely related to the system behavior, it is convenient to use the notion of *trace* which records a sequence of possible interactions from a given state.

Definition 2.2 (Traces of S)**.** *Let* $s_i \xrightarrow{l_0} s_{i+1} \xrightarrow{l_1} \cdots \xrightarrow{l_{n-1}} s_{i+n}$ *be a sequence of* $n \geq 0$ *adjacent edges in S. The concatenation of labels* $l_0 l_1 \cdots l_{n-1}$ *is a trace for* s_i*, and can be represented as* $s_i \overset{l_0 l_1 \cdots l_{n-1}}{\twoheadrightarrow} s_{i+n}$*. If we are not interested in the final state we can write it as* $s_i \overset{l_0 l_1 \cdots l_{n-1}}{\twoheadrightarrow}$*, while if we are not interested in the trace per se, but just in saying that* s_{i+n} *is reachable from* s_i *we write* $s_i \twoheadrightarrow s_{i+n}$*. A trace* t_1 *is a prefix of a trace* t_2 *(*$t_1 \preceq t_2$*) iff a (possibly empty) trace* t_3 *exists such that* $t_1 t_3 = t_2$*. The set of all traces of* s_i *is denoted by* $T(s_i)$*, and it always contains the empty trace* ε *which corresponds to a sequence of zero edges from* s_i*. The traces of S are denoted by* $T(s_0)$ *or, equivalently, by* $T(S)$.

The next constraint we impose on the system concerns the structure of the erasure operation. As noted in [5], the user needs to be aware of not only when a value supplied is subject to later erasure, but also the point at which the erasure of an input is intended to be complete. For this purpose we have the outputs $a!BE$ and $a!EE$, by which S communicates the beginning and the end of the erasure session to the user. The protocol we assume is that an input from the user which is subject to erasure is preceded by a BE message, as in $s_1 \xrightarrow{a!BE} s_2 \xrightarrow{a?v} s_3$. A communication $s \xrightarrow{a!EE} s'$ will be the signal to the user that the erasure is now complete. But to which input does the signal refer? Here we use an analog of the block-structure: we assume that the BE and EE are well bracketed so that each EE message uniquely identifies a corresponding BE and input.

To refine the definition of S according to those intuitions we proceed in two steps: first we propose a constraint on system structure related to the use of BE and EE actions, then we state the erasure definition.

Definition 2.3 (Well-Formed System)**.** *Let $s \xrightarrow{[\![v} s_1$ abbreviate $s \xrightarrow{a!BE} s' \xrightarrow{a?v} s_1$. A system S is well formed iff for each $t \in T(S)$:*

- *if $t = t'(a!BE)$, and $s_0 \xrightarrow{t'} \twoheadrightarrow s$, then $s \xrightarrow{[\![v}$ for each possible value $v \in \mathbb{V}$;*
- *in each prefix t' of t, the number of $[\![v$ labels is always greater or equal than the number of $a!EE$ labels;*
- *if $s_0 \xrightarrow{t} \twoheadrightarrow s \not\rightarrow$ then the number of $[\![v$ in t is equal to the number of $a!EE$ in t.*

Let us consider an execution of S such that $s_0 \twoheadrightarrow s_1 \xrightarrow{[\![v} s_2$. If a state s_3 exists such that $s_2 \xrightarrow{u} \twoheadrightarrow s_3 \xrightarrow{a!EE}$ and $a!EE$ corresponds, according to well bracketing, to the $\xrightarrow{[\![v}$ action, then we write $s_2 \xrightarrow{u]\!]}$. Note that u may be infinite, in that case $\xrightarrow{[\![v}$ does not have a matching $a!EE$.

Now the last aspect we need to consider to translate the previous description for systems into the new abstract setting is related to the input erasure definition. In the original definition the user is represented simply by a stream of input values. We represent this in our more abstract setting by specializing S to a particular input stream. (Later we will show that this representation of a user as a stream is indeed sufficient in a certain sense).

Definition 2.4. *Let I be a stream and, for any $n > 0$, let $I(n)$ denote the n-th value of the stream. For any stream I let $S(I)$ denote the refinement of S in which all input nondeterminism has been resolved as follows: $t \in T(S(I))$ if and only if $t \in T(S)$, and the for all $n > 0$, the n-th input label occurring in t has value $I(n)$.*

Note that the transition system for $S(I)$ is just a single trace (potentially infinite) of S.

Definition 2.5 (Abstract Input Erasure $E(S)$)**.** *Let S be a deterministic, well-formed system. Let $i(t)$ be an operator over traces which counts the number of input labels (of the form $a?v$) contained in t. Consider an arbitrary stream I and the associated refinement $S(I)$, such that $s_0 \xrightarrow{t} \twoheadrightarrow s_1 \xrightarrow{[\![v} s_2 \xrightarrow{u]\!]} s_3 \xrightarrow{z} \twoheadrightarrow$. Suppose that the input v is the n-th input of the trace. Then S is input erasing if for any I' which differs from I only at position n, we have that $S(I')$ has the trace $s_0 \xrightarrow{t} \twoheadrightarrow s_1 \xrightarrow{[\![w} s'_2 \xrightarrow{u']\!]} s'_3 \xrightarrow{z} \twoheadrightarrow$ and $i(u) = i(u')$.*

As we can see, the definition says that after erasure, the subsequent behavior of the system is independent of the value supplied at the beginning of the erasure session. The only constraint on the behaviour during the erasure session is that the number of inputs does not depend on the value which is subject to erasure (for more details on the motivation for the latter point see [5]).

3 Erasure Friendly Users

In this section we define the general user model, and the requirements that we need to impose on such a user, *erasure friendliness*, to ensure that its composition with an input-erasing system achieves composite erasure.

Since a reasonable scenario for erasure should involve either human or computer agents, the model of the user of an input erasing system should not make strong assumptions about the internal structure of the user (e.g., defining it through a particular language) as that would only really be suitable for modeling a computer agent. As a first approximation we suppose that we want a representation which, hiding all internal details, allows us to specify some behavioral constraints for the user and then observe its interactions with S to prove its soundness with respect to the notion of erasure.

Following the approach we used with systems, a labelled transition system seems to be suitable because it hides internal evolution and shows precisely the possible sequences of interaction. But in this case we need to consider a subtle aspect: the notion of erasure is meaningful if we have a handle on the secrets which are to be erased, since in particular we need to vary these secrets in order to observe the effect of the variation. A labelled transition system is too abstract to make this easy. Instead we add just enough structure to the labelled transition system to make it possible for us to control the erasure related information.

We therefore propose to represent the user through two components:

- a behavioral component U, the LTS, in which we include the computational aspect of the interaction with the system;
- a memory component δ, the store of secrets from which the U component has to fetch all the erasure related information before sending them to the system.

Let us define these concepts precisely.

Definition 3.1 (Abstract User)**.** *An abstract user is given by an LTS $U = (\mathscr{S}, \mathscr{L}, \mathscr{T})$ such that:*

- $\mathscr{S} = \{u_i : i \text{ is an index}\}$ *is the set of (distinct) states, in which u_0 is the initial one.*
- $\mathscr{L} = \{a?v, a!v, a?BE, a?EE, i?v \mid i \in \mathscr{I}\}$*, assuming $v \in \mathbb{V}$ like we did in the system description, is the set of labels for U, where the first four elements represent the counterpart of S actions, while the last one represents the input request for a value coming from δ. The idea is that the memory δ is a set of labelled values, where $\mathscr{I}$ are the labels used to distinguish them. Since the only property we require on $\mathscr{I}$ is to be a set of unique indexes, we can simply consider $\mathbb{N}^+$ as a representative for $\mathscr{I}$.*
- $\mathscr{T} \subseteq \mathscr{S} \times \mathscr{L} \times \mathscr{S}$ *is the set of transitions. We assume that also for users the input enabled condition holds, hence we have branching over all possible values of the domain in each input action of U.*

It is worth pointing out that while S is deterministic, U does not need to be deterministic, and this makes our framework quite expressive.

In order to understand the intention of the $i?v$ actions, let us define the δ component: δ is simply a function $\mathbb{N}^+ \to \mathbb{V}$ such that for each index it provides the value corresponding to that index.

As for the system, we can define the notion of an instance $U(\delta)$, which is obtained by refining the behavior of an abstract user U through a memory component δ. In other words, the traces of $U(\delta)$ are also traces of U, but with the restriction that for any transition labelled $i?v$ we have that $v = \delta(i)$.

Note that plugging together U and δ is not a way to resolve the nondeterministic behavior, since users can be purely nondeterministic, independently of input operations. [1]

Now that we have a general structure for U, we can specify some notational conventions about its behavior. The notion of trace is the same we gave for systems, hence let us consider $T(U) = T(u_0)$ as the set of traces for U, and $T(U(\delta)) \subseteq T(U)$. As for systems we introduce some convenient abbreviations for certain common sequences of interactions: let $u \xrightarrow{[\![i,w} u'$ be an abbreviation of $\exists u_2, u_3 . u \xrightarrow{a?BE} u_2 \xrightarrow{i?v} u_3 \xrightarrow{a!w} u'$, and $u' \xrightarrow{t]\!]}$ be a shorthand for $\exists u_1 . u' \xrightarrow{t} u_1 \xrightarrow{a?EE}$.

With a clear definition for the user we can specify the communication model for U and S which will allow us to explore in detail the message exchanges through the channel a. For a given U and S, $U|S$ denotes the transition system consisting of states formed from the product of the states of U and S respectively, with initial state $u_0|s_0$, and transitions given by the rules in Figure 2.

$$\frac{u \xrightarrow{a?v} u' \qquad s \xrightarrow{a!v} s'}{u|s \xrightarrow{v} u'|s'} \qquad \frac{u \xrightarrow{a!v} u' \qquad s \xrightarrow{a?v} s'}{u|s \xrightarrow{v} u'|s'}$$

$$\frac{u \xrightarrow{a?BE} u' \qquad s \xrightarrow{a!BE} s'}{u|s \xrightarrow{BE} u'|s'} \qquad \frac{u \xrightarrow{a?EE} u' \qquad s \xrightarrow{a!EE} s'}{u|s \xrightarrow{EE} u'|s'}$$

$$\frac{s \xrightarrow{b!v} s'}{u|s \xrightarrow{b!v} u|s'}\, b \neq a \qquad \frac{u \xrightarrow{i?v} u'}{u|s \xrightarrow{i?v} u'|s}\, i \neq a$$

Figure 2: transition rules for $U|S$

As we can see, the labels for the joint environment $U|S$ are different from both user and system, since they show the value of the message which is passing in a communication between user and system. But since we are modeling erasure using a noninterference-style, it is quite reasonable to have such a "transparent" channel between U and S.

Following the same approach we adopted with S, there are some aspects of the user behavior we have to constrain in order to ensure its positive attitude towards erasure. Since some of them are related to its internal structure, we need the counterpart of system well formedness, which we define as follows.

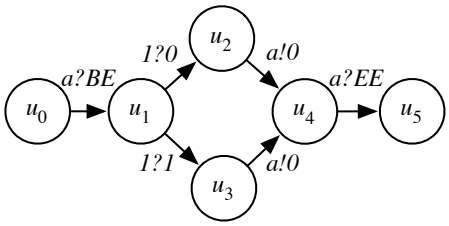
Definition 3.2 (Well Formed User)**.** *A user U is well formed iff for each trace $t \in T(U)$:*

- *if $t = t'(a?BE)$, and $u_0 \xrightarrow{t} u$, then there exists an i such that for all v, $u \xrightarrow{i?v} u_v \xrightarrow{a!v} u'_v$;*
- *if $t = t_1(i?v)t_2$, $t_2 \neq \varepsilon$, then t_1 ends in $a?BE$ and t_2 begins with $a!v$;*
- *in each prefix t' of t the number of $[\![i,v$ labels is always greater or equal than the number of $a?EE$ labels, while if $u_0 \xrightarrow{t} u \not\rightarrow$ then the number of $[\![i,v$ in t is equal to the number of $a?EE$ in t.*

While the last requirement is similar to the feature requested on system side, the first two state how the user actually exploits the memory saying that whenever a value is sent to the system at the beginning of an erasure exchange, it has been fetched from the memory. In order to explain why we need such property, let us consider the following example.

[1] We can remark here that it would also be rather natural to define δ as a transition system and to compose U and δ. But in any case the parallel composition would not be a standard CCS-style one since sometimes we will need to witness the values passed from δ to U.

Example 3.3. As we have already said, the erasure property we are going to specify is described in a noninterference style, which means that we need a way to change secrets and let them flow to the system in a controlled way. Suppose that we have such user U:



Although U implements erasure protocol properly, it sends values for erasure independently of δ, hence we cannot track the dependence between the value sent and subsequent behaviour by varying δ.

There are some other remarks on user well formedness it is worth pointing out:

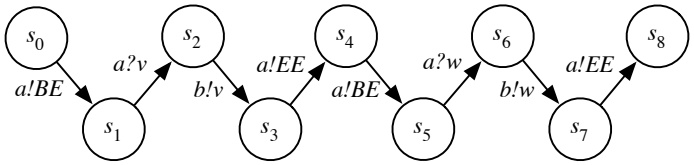
- we require that local read operations are only performed for the purpose of erasure related exchanges. Although this requirement is not strictly necessary to characterize an erasure friendly user, it makes the presentation of our results simpler, without restricting the range of user behaviors we are able to describe;
- since we are defining the counterpart of well formed systems, the conventions related to well nested erasure blocks hold also for users.

Now we have a way to characterize a user which uses δ as a secret repository and applies the erasure protocol properly, but this is obviously not enough to ensure a *responsible* use of such secrets. For sure we need to constrain the user in a way that a particular secret is used at most once during its interaction with S. This is what we dub *secret singularity*.

Definition 3.4 (Secret Singularity)**.** *A user U is secret singular if for any i and any trace $t \in T(U)$, a label of the form $i?v$ (for any v) occurs at most once in t.*

The following example should help to understand why we require secret singularity to user.

Example 3.5. Let us consider the following system S, which contains a chain of two non-nested erasure sessions and where v and w represent generic elements of $\mathbb{V}$ (hence one should imagine as many input edges as the size of that domain).



This system obviously satisfies the input erasure definition of the previous section. But let us consider the user in Figure 3. Suppose that the domain for the secret contains the values 0 and 1, then just two out of the four output operations admitted by S are performed by U, namely $(a!0)(a!0)$ and $(a!1)(a!1)$. This happens because the secret is used twice, and we cannot prove that the behavior of S and U after the first

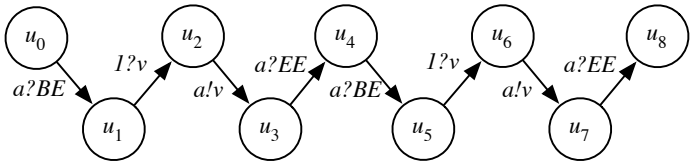


Figure 3: user for Example 3.5

erasure session is independent of secret value received (as we will see, this will be a requirement of the definition of composite erasure). Note that secret singularity is not satisfied by U, which is instead well formed.

In order to understand secret singularity, one should consider we are trying to build a structure in which each secret is completely independent of the others. Whenever the same portion of δ is used more than once, we immediately loose this independency, because changing that portion creates effects in more than one communication of the user to the system. It is therefore useful to think about the i component of an $i?v$ request to δ as a timestamp representing the moment in which the user requests a secret: in this way secret singularity allows the *actual value* of a secret being used twice, but requires that δ supports this duplication (i.e. contains two locations $i \neq j$ such that $\delta(i) = \delta(j)$).

Two other constraints on the user seem to be necessary to support the erasure mechanism properly:

- its behavior outside of an erasure phase should not depend on values of secrets (what we call *secret confinement*), and
- such dependency should always preserve a stream-like behavior, which means that sequences of outputs proposed by the user have to be independent of both secrets and feedbacks received from the system, in order to ensure user will not be an unconscious storage for S. We call this property *stream ability*.

Secret confinement is a property of the user behavior outside each erasure related part of its structure. Stream ability, on the other hand, is specific to the part of the behavior within an erasure block. So, in order to describe those properties, we must define a way to separate those complementary parts of U's traces, and then define their intended structure according to the informal requirements we have just stated.

We focus first on the actions of a user which are not part of an erasure block. The following definitions facilitate this.

Definition 3.6 ((Incomplete) User Erasure Session)**.** *A subtrace t is an* user erasure session *if $u_0 \twoheadrightarrow u \xrightarrow{t}\!\!\!\!\!\twoheadrightarrow u_v \twoheadrightarrow$ and $t = (\![i,v)(t_v\!])$ such that it is balanced from the erasure point of view (so, for example, in t_v the number of a?BE events is equal to the number of a?EE events).*

A user incomplete erasure session *is a subtrace t such that $u_0 \twoheadrightarrow u \xrightarrow{t}\!\!\!\!\!\twoheadrightarrow$ and $t = (\![i,v)t'$ where $(\![i,v)t''$ is not an erasure session for any $t'' \preceq t'$.*

Erasure sessions which share the same initial state are then grouped together in erasure zones.

Definition 3.7 (Erasure Zone and Incomplete Erasure Zone)**.** *The (incomplete) erasure zone of a user state u, $E_\bullet(u)$, is defined as*

$$E_\bullet(u) = \{t \mid u \xrightarrow{t}\!\!\!\!\!\twoheadrightarrow,\ t \text{ is an (incomplete) erasure session}\}$$

For each erasure zone we then define the frontier, namely the set of states which are immediately outside the erasure zone.

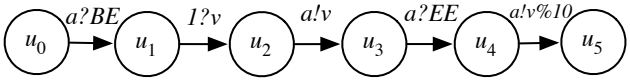
Definition 3.8 (Erasure Frontier)**.** *Let $E_\bullet(u)$ be an erasure zone. The union of u_v such that $\exists u'.u \xrightarrow{[\![i,v} u' \xrightarrow{t_v]\!]} u_v$ is called erasure frontier of $E_\bullet(u)$ and it is denoted by $E_\circ(u)$.*

Now we have everything we need to state secret confinement:

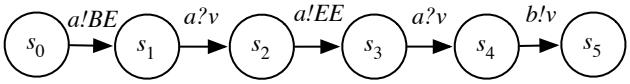
Definition 3.9 (Secret Confinement)**.** *Let U be a well formed user. U satisfies secret confinement iff for each state u we have that $\forall u_v,\ u_w \in E_\circ(u)$, $T(u_v) = T(u_w)$.*

It turns out that, for users satisfying secret confinement, differences among secrets are not reflected in differences among behaviors outside erasure zones, since each state in the frontier shows the same set of traces. We can therefore say that, for those users, secret values are *confined* inside erasure zones. The following example shows how can a system fool a user which does not satisfies secret confinement.

Example 3.10. Let us consider the following user U:



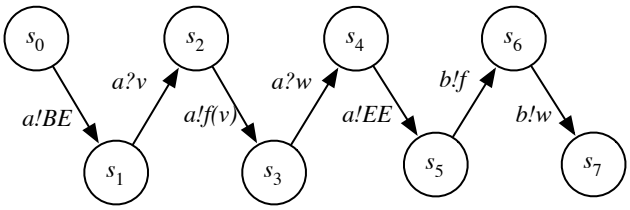
in which $v\%10$ represent the less significant digit of v. The final state of U after the erasure session shows dependency on secret value, and it is quite easy then to figure out a system like:



which is input erasing but it captures (and communicates) a portion of U secret.

Now we need to consider the internal property of erasure zones. We have already said each member of the zone has to show a stream behavior, which basically consists in accepting any input from S without being influenced by it. This is necessary because if S is malicious, U can be tricked into storing erased data on behalf of S.

Example 3.11. Let us consider the following system S:



which represents a slight variation ofthe discount example (Section 1).

Let us also consider the user U in Figure 4.

Although U satisfies secret confinement (after the reading operation the secret is sent and U does not perform any particular action on it), S is able to send the secret back to the user and wait U to send it

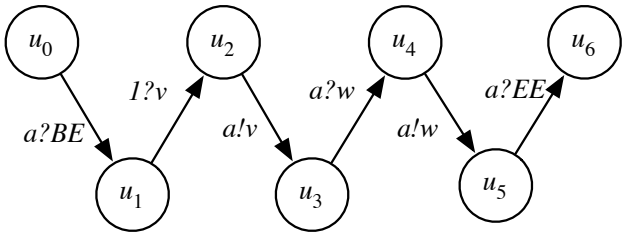


Figure 4: system for Example 3.11

back again: in this way it will be possible for S to send the encoding function f and the value $f(v)$ to an attacker without breaking the input erasure property.

But how can we actually impose the stream behavior among elements in $E_\bullet(u)$? Since we are not interested in input actions, the idea is that we have to define a relation between traces in which the output values sent to the system depend only on predetermined decisions, which have to hold independently on δ content and system responses. We thus may find three possible cases while trying to compare two traces:

- if two non erasure outputs are performed, then the value has to be the same;
- if two erasure outputs are performed, then the index of the secret used in the transaction has to be the same, because δ is not under U's control;
- if one erasure output has to be matched with a non erasure one, then the two values have to be the same.

The following output equality relation between traces states those requests formally.

Definition 3.12 (Output Equality)**.** *$\stackrel{o}{=}$ is the largest symmetric binary relation defined over the set of rules contained in Figure 5.*

$$\frac{t \stackrel{o}{=} t'}{([\![i,v)t \stackrel{o}{=} (a!v)t'} \qquad \frac{t \stackrel{o}{=} t'}{([\![i,v)t \stackrel{o}{=} ([\![i,w)t'} \qquad \frac{t \stackrel{o}{=} t'}{(a!v)t \stackrel{o}{=} (a!v)t'}$$

$$\frac{t \stackrel{o}{=} t'}{lt \stackrel{o}{=} t'} \text{ if } l \notin \{([\![i,v), (a!v)\} \qquad \varepsilon \stackrel{o}{=} \varepsilon$$

Figure 5: Output Equality

t and t' are output equivalent if $t \stackrel{o}{=} t'$.

Now we can state the stream ability for a user U.

Definition 3.13 (Stream Ability)**.** *A well formed user U has the stream ability if for all states u it holds that $\forall\, t,\ t' \in E_\bullet(u)$, $t \stackrel{o}{=} t'$.*

If a user satisfies all the definitions we stated so far, it is doing all its best to behave in a proper way with respect to erasure. We define such attitude as *erasure friendliness*.

Definition 3.14 (Erasure Friendly User $EF(U)$)**.** *Let U be a well formed user which satisfies secret singularity, secret confinement and stream ability. Then we define U as an erasure friendly user, and we denote all its properties with the notation $EF(U)$.*

So far we worked mainly on the user structure, but we also need to rule out the possibility that U is able to deadlock S by performing the wrong communication, and we have to do that because S is performing erasure only if it is able to reach the end of the erasure session. To avoid such kind of scenarios we impose *liveness* on the user side, according to the following definition.

Definition 3.15 (Liveness)**.** *Let $S = (\mathscr{S}, \mathscr{L}, \mathscr{T})$ be a well formed system. We say $s \in \mathscr{S}$ is a system input state if $s \xrightarrow{a?v} s_v \in \mathscr{T}$*[2]*. Let U be a well formed user. Then U has the liveness property for S if:*

- *$u_0|s_0 \twoheadrightarrow u_1|s_1$ and s_1 is a system input state imply that $\exists\, u_2$ such that $u_1 \xrightarrow{u} \twoheadrightarrow u_2 \xrightarrow{a!v}$ and $u_1|s_1 \twoheadrightarrow u_2|s_1 \rightarrow$;*
- *$u_0|s_0 \twoheadrightarrow u_1|s_1$ and $s_1 \rightarrow s_2$, s_1 not a system input state, imply that $\exists\, u_2$ such that $u_1 \twoheadrightarrow u_2$ and $u_1|s_1 \twoheadrightarrow u_2|s_2$.*

4 Composite Erasure and Soundness of $U|S$

Since we are reasoning about $U|S$, let us import some notational conventions from the previous sections. If we assume that both U and S are well formed we are sure that each time a *BE* communication is performed in a trace of $U|S$, an appropriate reading operation $i?v$ and a communication v will follow immediately. So let $u|s \xrightarrow{[\![i,v} u'|s'$ be an abbreviation for $u|s \xrightarrow{BE} u_1|s_1 \xrightarrow{i?v} u_2|s_1 \xrightarrow{v} u'|s'$ and let $\xrightarrow{t]\!]}$ denote a finite erasure session.

We can now state the definition for the erasure property which involves both system responsibility towards secret provided by user, and user's caution in treating potentially dangerous information from system in a safe way.

Definition 4.1 (Composite Erasure $EC(U|S)$)**.** *Let S be a deterministic, well-formed system, with initial state s_0. Let U be a well-formed user with the liveness property with respect to S and let δ an arbitrary memory for U, such that together they define an instance $U(\delta)$. Consider then a trace $u_0|s_0 \xrightarrow{t} \twoheadrightarrow u_1|s_1 \xrightarrow{[\![i,v} u_2|s_2 \xrightarrow{t']\!]} u_3|s_3 \xrightarrow{z} \twoheadrightarrow$ of $U(\delta)|S$.*

Then S and U are composite erasing ($EC(U|S)$) if for any instance $U(\delta')$, such that δ' differs from δ only at position i, we have that $u_0|s_0 \xrightarrow{t} \twoheadrightarrow u_1|s_1 \xrightarrow{[\![i,w} u'_2|s'_2 \xrightarrow{t'']\!]} u'_3|s'_3 \xrightarrow{z} \twoheadrightarrow$.

The input erasure definition has two conditions to ensure that the scenario after erasure is independent of secrets. The first is related to the system behavior (equality between traces), while the second refers to the status of the streams associated to S, which have to be consumed in the same way. Here the second condition does not correspond to an explicit constraint, but since the behavior z includes both contribution of S and U, it is implicit in the equality of trace.

Now the main result of this work can be stated, which relates local properties on both user and system sides to the notion of composite erasure we have just defined.

Theorem 4.2 (Soundness of $U|S$ wrt composite erasure)**.** *Let S be an input erasing system ($E(S)$) and U be an erasure friendly user ($EF(U)$) which satisfies the liveness condition for S. Then $U|S$ satisfies the composite erasure properties $EC(U|S)$.*

Proof. Consider a memory δ for U and suppose that $U(\delta)|S$ produces a trace $u_0|s_0 \xrightarrow{t} \twoheadrightarrow u_1|s_1 \xrightarrow{[\![i,v} u_2|s_2 \xrightarrow{u]\!]} u_3|s_3 \xrightarrow{z} \twoheadrightarrow$.

[2] Recall that input enabled condition on S makes s able to accept each possible input $w \in \mathbb{V}$.

Now consider a δ' differing from δ only at position i, where $\delta'(i) = w$. We then need to prove $U(\delta')$ together with S produces the trace $u_0|s_0 \overset{t}{\twoheadrightarrow} u_1|s_1 \overset{[\![i,w}{\longrightarrow} u_2'|s_2' \overset{u']\!]}{\longrightarrow} u_3'|s_3' \overset{z'}{\twoheadrightarrow}$ such that $z = z'$.

Let us start showing that $U(\delta')|S$ produces $u_0|s_0 \overset{t'}{\twoheadrightarrow} u_1'|s_1'$ such that $t = t'$, $s_1 = s_1'$ and $u_1 = u_1'$. Applying user well formedness and secret singularity we have that t is a trace which does not contain local reads on index i. Hence also $U(\delta')|S$ behaves according to t, reaching the same intermediate states $U(\delta)|S$ was able to reach, thus we have $t = t'$, $s_1' = s_1$ and $u_1' = u_1$.

Now we need to show $u_1|s_1 \overset{[\![i,w}{\longrightarrow} u_2'|s_2' \overset{u']\!]}{\longrightarrow} u_3'|s_3'$ when the system is communicating with the user instance $U(\delta')$. Due to S determinism, s_1 emits a request of a secret value as it happens in the $U(\delta)|S$ execution. Then, since U is input enabled an well formed, it will fetch the secret contained in i-th location of δ' (namely w) and send it to the system. This is represented by the first step $u_1|s_1 \overset{[\![i,w}{\longrightarrow} u_2'|s_2'$.

According to the definition of erasure zone, both $[\![i,v$ and $[\![i,w$ are the first actions of two erasure sessions contained in the same erasure zone $E_\bullet(u_1)$, hence stream ability applies. This, together with user liveness, implies that it exists a state u_3' such that $u_2'|s_2' \overset{u']\!]}{\longrightarrow} u_3'|s_3'$ and the sequence of output contained in u' and u are the same. This property, which holds for an arbitrary value related to secret i, allows us to apply input erasure condition on S side, having $s_3 \overset{\alpha}{\twoheadrightarrow}$ and $s_3' \overset{\alpha}{\twoheadrightarrow}$.

Since secret confinement holds, we have that each trace of u_3 finds a correspondent trace on $T(u_3')$, hence due to S determinism we have $u_3'|s_3' \overset{z}{\twoheadrightarrow}$ as required. □

5 Related Work

This work has studied an interactive form of block structured erasure. More expressive non-block structured erasure notions have been studied by Chong and Myers [2, 3], where the erasure point is associated with an arbitrary condition on the computation history of the system, but these do not consider an interactive system. Erasure and declassification for multithreaded programs is studied in [6], although what the authors call erasure is merely the much weaker notion of upgrading the security classification of a variable.

Erasure as defined here is a close relative of noninterference. Noninterference for interactive systems have been studied extensively - for example in language based setting [7] enforce language based security conditions in an imperative language which allows input-output interactions, as well as nondeterminism. This result is obtained through the notion of *strategy*, which is used to represent the behavior of the user (secrets provider) as a function of previously exchanged values. In [4] the results of [7] are refined by classifying the strategies according to their expressive power and proving that it is sufficient to model the user strategy as a simple stream of values in the case that the system is deterministic. With respect to these works our modeling goal is rather different. A user, for example, is modeled as having a stream-like behavior not because it is *sufficient* (as in the noninterference case [4]) but because it is *necessary* in order that the user does not feed data into a system which depends on previously erased values.

Erasure properties are related to *usage control* policies (e.g., [8, 9, 10]) – however the fundamental difference appears to be that the usage control area is related to access control rather than information flow control, so for example the scenario in the introduction where the user unwittingly stores a secret on behalf of the system would seem to be out of reach of those methods.

Another area where connections to erasure may be seen is in a recent account of *opacity* at the level of transition systems [1], although at the time of writing deeper connections between the concepts remain to be established.

6 Conclusions and Further Work

The definition of *information erasure* studied in [5] considers an interactive environment, composed by an erasing system (data processor) and many users (secrets providers), but did not define the latter precisely. Here we tried to provide a formal but abstract model of the user and a collection of requirements called *erasure friendliness* on its behavior, which are sufficient to ensure *composite erasure*, namely the erasure property obtained by composing an erasure friendly user together with an erasing system.

In doing so we identify stronger requirements on the user than those informally described in the previous work, which are basically a countermeasure for the liberal interactions an erasing system can perform inside an erasure session. In particular it turns out that erasure related data, as well as system feedbacks, should not be examined by the user during an erasure session (i.e. communicated or received without understanding their actual value).

The results we achieved in this work suggest a wide range of future developments.

Completeness Considering our contribution in isolation, a natural enhancement is trying to prove the completeness on our approach: we would like to show that erasure friendliness is not only a sufficient condition but a necessary one. In other words if a user is not erasure friendly then we would like to show the existence of an input erasing system for which composition yields system which is not jointly erasing.

Multilevel and Mutual Erasure One of the simplifications we made to the earlier definitions of erasure was to treat only "complete erasure". In the general scenario we have a multilevel lattice and erasure from one level to a higher level. Adding the multilevel security structure on the system side, in which each security level comes equipped with a communication channel, allows to recover the original notion of "erasure below a certain level b". But to have this it is necessary to rearrange the system representation, in order to examine contents of output actions of S below the intended erasure level. More difficult is to recover the system full input-output behavior: according to our first intuitions on the problem, allowing the system to perform I/O operations in each channel requires the classification of each principals of the communication according to three possible roles (secret provider, secret observer – a user higher than erasure level, and secret co-manager – a user who receives information tainted by other users' secrets, which are supposed to be erased). Such complexity increases the number of possible unwanted leaks of secrets, and seems to require a further strengthening of behavioral constraints for system users.

Programming with Erasure The definition of erasure that we have studied is, like noninterference, a very strong property. In practice – as with noninterference – this means that one may have to program systems in a particular style in order to satisfy erasure, or to introduce controlled mechanisms to weaken the erasure requirements, analogous to *declassification*. As an example, consider the case of a credit card transaction where the card can be denied by the bank. It would be natural for the system (merchant) to request a new card from the user. But if this were done in the same session then this would not be erasing: the number of inputs from the user would depend on the first card number. This can be solved in two ways (i) by structuring the system so that the first interaction is concluded as a failed interaction, and inviting the user to engage in a subsequent purchase (where hopefully the user will try a valid card), or (ii) by *revoking* the erasure promise on discovering an invalid card. It would seem that the various *dimensions of declassification* [11] would be suitable to study erasure revocation mechanisms.

Implementation: A Safe Wallet There are also implementation perspectives related to our results: since we represented the user through two distinct components (U for the computation, δ as a secret repository), we have begun looking at how it would be possible to define an advanced memory component (a "wallet") W which acts as a proxy between a user and a system. The idea would be to achieve an erasure-friendly user (modulo liveness issues) as a composition $A|W(\delta)$ of an *arbitrary* user A and a generic safe wallet W containing secrets δ. Since it is reasonable to consider the wallet as a real implemented agent, we believe it will be an effective mechanism to enforce our behavioral properties in a purely syntactic manner. The wallet W, together with a certified input erasing system S (a trusted authority mechanism could certify that property, using the type system proposed in [5] for example), would therefore create a composite erasing environment without requiring the real user to constrain his own behavior.

Acknowledgements Sebastian Hunt provided valuable inputs throughout the development of this paper. Many thanks to our colleagues in the ProSec group at Chalmers and to the anonymous referees for useful comments and observations. This work was partially financed by grants from the Swedish research agencies VR and SSF, and the European Commission IST-2005-015905 MOBIUS project.

References

[1] Jeremy Bryans, Maciej Koutny, Laurent Mazaré & Peter Y. A. Ryan (2008): *Opacity generalised to transition systems*. *Int. J. Inf. Sec* 7(6), pp. 421–435. Available at `http://dx.doi.org/10.1007/s10207-008-0058-x`.

[2] S. Chong & A.C. Myers (2005): *Language-based information erasure*. *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop* , pp. 241–254.

[3] Stephen Chong & Andrew C. Myers (2008): *End-to-End Enforcement of Erasure and Declassification*. In: *Proceedings of the IEEE Computer Security Foundations Symposium*. pp. 98–111.

[4] D. Clark & S. Hunt (2006): *Noninterference for Deterministic Interactive Programs*. In: *FAST 2008*. IEEE Computer Society, Washington, DC, USA, pp. 190–201.

[5] S. Hunt & D. Sands (2008): *Just Forget it – The Semantics and Enforcement of Information Erasure*. In: *Programming Languages and Systems. 17th European Symposium on Programming, ESOP 2008*, number 4960 in LNCS. Springer Verlag, pp. 239–253.

[6] Li Jiang, Lingdi Ping & Xuezeng Pan (2007): *Handling Information Release and Erasure in Multi-Threaded Programs*. In: *CIS '07: Proceedings of the 2007 International Conference on Computational Intelligence and Security*. IEEE Computer Society, Washington, DC, USA, pp. 824–828.

[7] Kevin R. O'Neill, Michael R. Clarkson & Stephen Chong (2006): *Information-Flow Security for Interactive Programs*. In: *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*. IEEE Computer Society, Washington, DC, USA, pp. 190–201.

[8] Jaehong Park & Ravi Sandhu (2002): *Towards usage control models: beyond traditional access control*. In: *SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies*. ACM, pp. 57–64.

[9] Jaehong Park & Ravi Sandhu (2004): *The UCONABC usage control model*. *ACM Trans. Inf. Syst. Secur.* 7(1), pp. 128–174.

[10] Alexander Pretschner, Manuel Hilty & David Basin (2006): *Distributed usage control*. *Commun. ACM* 49(9), pp. 39–44.

[11] A. Sabelfeld & D. Sands (2005): *Dimensions and Principles of Declassification*. In: *Proc. IEEE Computer Security Foundations Workshop*. pp. 255–269.