Handling Declared Information Leakage

[Extended Abstract] *

R. Echahed and F. Prost
Institut d'Informatique et de Mathématiques Appliquées de Grenoble,
Laboratoire Leibniz, 46, av Félix Viallet, 38000 Grenoble, France
{echahed | prost}@imag.fr

ABSTRACT

We address the problem of controlling information leakage in a concurrent declarative programming setting. Our aim is to define formal tools in order to distinguish between authorized, or declared, information flows such as password testing (e.g., ATM, login processes, etc.) and non-authorized ones. We propose to define security policies as rewriting systems. Such policies define how the privacy levels of information evolve. A formal definition of secure processes with respect to a given security policy is given.

Non interference

The problem of the preservation of the confidentiality of data is nowadays a prominent feature of computer systems. This is especially true in a context where programs and data may move around using communication networks. The usual theoretical approach of this problem, initiated by Goguen and Meseguer in [7], uses the notion of non-interference. A great deal of work has been done along these lines. A common feature of these works is that they consider information leakage from a very strict point of view despite the fact that in real world applications, such absolute non-interference properties can hardly be obtained e.g., [12]. The notion of approximate non-interference e.g., [11, 10], along with declassification and other weakenings of non-interference e.g., [15, 6] have been only recently investigated. Following this line of study, we propose a formalization in which the user may declare its security policy. The idea is that the user may *declare* that some functions are allowed to provoke information leakage. We call such functions *declassifying* functions. This formalization extends previous works in the expressivity of security policies.

A typical example of declassifying functions is given by the password mechanisms: think of Automated Teller Machines, log on procedures (on computers, web sites with restricted access...), restricted areas, PIN for cellular phones etc. Schematically, the situation is the following: to access secret areas one has to give the right password to a controller. From a strict non-interference point of view, there is a forbidden information leakage in such situations. Indeed, it is possible to try all possible passwords to be finally granted the access to the restricted area. Therefore private information, e.g. the knowledge of the password, may influence public behavior, e.g. a hacker randomly submitting a number may guess the password by chance.

Nevertheless, this kind of leakage has to be considered as harmless since they are controlled through the declaration of declassifying functions. Indeed, information leakage is only allowed for some *identified* parts, namely the ones checking the password. Note that it makes possible to define a system policy that deters brute-force attacks: for instance the system may become inactive after some unsuccessful attempts (after three non valid PIN, ATMs retain the credit card). Another example of such controlled information leakage is the case of public key cryptography. Suppose that encrypted data are public, since both the encryption key and algorithm are known, it is theoretically possible to encrypt every file of some fixed size to find out which one gives the same encrypted file. It is a broader version of the password verification done in UNIX systems in which encrypted password are stored and readable by anyone.

Note that this scheme is not related with cryptographic issues. Declassifying functions simply play the role of an interface between security levels, they are not required to be cryptographic functions or whatsoever. It is out of the scope of our analysis to check if declassifying functions are really "safe" from a practical or theoretical point of view. What we want to do is to be able to study the security of a process with respect to a given security policy.

We want to formalize the following idea: if a function is declared as declassifying by a security policy, we shall make abstraction of the leakage specifically coming from the use of this function. In other words a process is secure if the only information leakages produced are consistent with the declared security policy.

Execution Model

The computational model used is significant. For instance it is well known that some programs are safe in a simple imperative setting and are not in a more sophisticated setting including concurrency (see [13]).

*A full version of this paper is available at `www-leibniz.imag.fr/~prost/decl_inf_leak.pdf`
An earlier version can be found at `www-leibniz.imag.fr/NEWLEIBNIZ/LesCahiers/2003/Cahier82/ResumCahier82.html`

The computational model used in this paper is a simplified version of the one proposed in [4, 5]; we refer to these works for a more detailed presentation of the complete model and its implementation. Roughly speaking, a program or a component in our framework consists of two parts $K = (F, \Pi\!R)$. $\Pi\!R$ is a set of process definitions and F is a declarative program, i.e., a set of formulæ (let us limit ourselves to Term Rewrite Systems (TRS) in this paper), which we call a *store*.

The execution model of a component can be schematized as follows. Processes communicate by performing actions; they may modify common store F, i.e., by altering it in a non-monotonic way, for example by simply redefining constants (e.g., adding a message in a queue) or by adding or deleting formulæ in F. Hence, the execution of processes will cause the transformation of the store F. Every change of the store is the result of the execution of an *action*. A store F is used to evaluate expressions (i.e. normalization of terms in our setting).

This execution model has a great expressivness : it includes procedure calls (and thus recursivity), unbounded process creation and is without any restriction concerning sequentiality and parallelism (compare with, e.g. [1], where processes of the form $P \parallel Q; R$ are not treated).

Security Policy, secure processes

In this section we present how to declare a security policy by means of (confluent and terminating) rewrite systems. Normal forms of such security policies are elements of a finite lattice representing different privacy levels of data. In the following we suppose given although non specified such a finite lattice of privacy levels $\mathfrak{L}$. Therefore for each function f we associate a set of rewrite rules like e.g. $f(\pi_1, x) \to \pi_2$, where π_1, and π_2 are elements of a lattice of privacy levels. At a first glance these rewrite rules may be seen as security profiles for functions. Such rewrite systems define a *security policy* modeling assumptions made by the programmer about the security behavior of functions. For instance it is sensible to declare that the result of the application of an encryption function onto private data is public. However, it is possible to define a security policy including the following rule $id(x) \to \bot$, where id stands for the classical identity function. In this case, the security policy is somewhat clumsy since $id(d)$, which is evaluated to the lowest privacy level, would be a public version of any information d (secret or not). *Our aim is to provide an expressive language to express security policies, and to give an algorithm to check wether a security policy is fulfilled by a program or not.* The fact that a security policy is clumsy or not is out of the scope of this paper and thus not discussed here. The reader should notice that for each function f is associated (at least) two rewrite systems: a classical one dedicated to data processing (e.g., $f(x) \to x$) and a second one defining a security policy (e.g., $f(\top) \to \bot, f(\bot) \to \bot$).

Apart from security policies, our aim is to give a formal definition of secure processes (threads) in presence of a given security policy. For that we should distinguish authorized information leakage declared within a security policy, from unauthorized ones corresponding to classical interference issues.

This is done through a specific evaluation process in which declassified terms are evaluated in a single store (therefore the evaluation of a declassified term will be the same for two different stores). We develop a notion of process equivalence accordingly. Then it is possible to state a non-interference like property (information may only flow from lower privacy levels to higher ones).

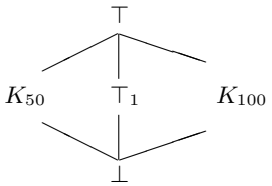
A security policy, $\mathcal{SP}$, is a terminating and confluent TRS defined over the signature $\Sigma \cup \mathfrak{L}$ such that: $\mathcal{SP}$ introduces no junk into $\mathfrak{L}$. I.e., for all ground terms, t, over $\Sigma \cup \mathfrak{L}$, $t!_{\mathcal{SP}}$ (the normal form of t w.r.t. $\mathcal{SP}$) is in $\mathfrak{L}$. $\mathcal{SP}$ introduces no confusion into $\mathfrak{L}$. I.e., $\forall \tau_1, \tau_2 \in \mathfrak{L}, \tau_1 \neq \tau_2 \implies \tau_1 \not\xrightarrow{*}_{SP} \tau 2$. Functions in Σ are monotonic w.r.t. privacy levels: if $\forall \tau_1, \ldots, \tau_n \in \mathfrak{L}, \forall \tau'_i \in \mathfrak{L}, \tau_i \sqsubseteq \tau'_i$ then
$f(\tau_1, \ldots, \tau_i, \ldots, \tau_n)!_{\mathcal{SP}} \sqsubseteq f(\tau_1, \ldots, \tau'_i, \ldots, \tau_n)!_{\mathcal{SP}}$

The no junk condition ensures that every data represented as a (ground) term has a privacy level ($\in \mathfrak{L}$). The no confusion property guaranties that different privacy levels will never be equated by mistake.

Consider the following example. Let $\Sigma = \{f, g, h\}$ be a signature consisting of three function symbols with the following respective arities 2, 2 and 1. Let $\mathfrak{L} = \{\bot, \top\}$ and $\bot \sqsubseteq \top$. The following set of rules defines a security policy $\mathcal{SP}$: $\begin{array}{ll} f(x,y) \to \top & g(x) \to x \\ h(\top, x) \to \bot & h(\bot, x) \to \bot \end{array}$ Now $g(f(\top, \bot))$ has $\top$ as normal form whereas $g(h(\top, \top))$ has $\bot$ as normal form.

Usually privacy level is supposed to increase through computation in order to avoid information leakage. Therefore the evaluation of a function using information of privacy level π should yield a result with a privacy level *greater* than or equal to π. However, this is no longer the case whenever one has to deal with programs which downgrade or declassify data. In the previous example of security policy $\mathcal{SP}$, function h declassifies information. Indeed the rule $h(\top, x) \to \bot$ depends on $\top$ privacy level and gives a result of level $\bot$. It means that this security policy *allows* function h to declassify the information given in its first argument.

One feature of our approach is that it can define subtle security policies. For instance, there are several levels in the power of cyphers, some are unbreakable (Vernam cipher) whereas others offer less security (Caesar cipher). There is also the case of algorithms depending on the size of the used key: an RSA with a 100-digits key is more reliable than an RSA with 50-digits key. It is possible to encode such subtleties within adequate lattice and security policy. Think for instance of a function RSA of arity 2, where the first argument is the key and the second the message to be cyphered. Now consider the following lattice of privacy levels $\mathfrak{L} = \{\bot, \top, \top_1, K_{50}, K_{100}\}$:



The idea behind this definition is that K_{50}, K_{100} represent privacy levels dedicated to keys for encryption algorithms whereas $\top, \top_1$ and $\bot$ are privacy level for data to be encrypted. We could then give the following security policy:

$$\begin{array}{ll} & RSA(\top_1, x) \to x \\ RSA(K_{100}, y) \to \bot & RSA(K_{50}, x) \to \top_1 \\ RSA(\top, x) \to x & RSA(\bot, x) \to x \end{array}$$

Notice that the three last rules are given in order to comply with the definition of a security policy. These rules are not

supposed to be used in practice since RSA's first argument, the encryption key, has to be of privacy level K_{50} or K_{100}.

In other words, encryption with key K_{100} downgrades everything to public level ($\bot$), whereas encryption with a less powerful key downgrades to a more private level ($\top_1$), because the cypher is less powerful. The other rules are given for the sake of completeness (security policy has to be terminating for any argument and the result must be an element of $\mathfrak{L}$). In an actual setting they are of no use since the first argument of an encryption algorithm is meant to be a key, and thus must have a privacy level in $\{K_{50}, K_{100}\}$. This must be checked in an actual system but it is not meaningful here.

We now introduce the notion of declassified terms. As we said earlier a function having a security policy assigning a level less private than its argument *may* declassify information. Indeed, in some ways it depends on private data and gives a result of a more public status. Therefore, they represent potential information flows. We identify terms potentially declassifying information, i.e. terms of public status having subterms of private status. It is possible only through the use of declassifying functions. Let $t = f(t_1, \ldots, t_n)$ with $n \geq 1$ be a term. It is said to be *declassified* if there is $j \in \{1, \ldots, n\}$ such that $\pi(t_j) \not\sqsubseteq \pi(t)$, with $\pi(t)$ denoting the privacy level of t w.r.t. the security policy, or t is a subterm of a declassified term.

Another standard notion in confidentiality is the store equivalence up to some privacy level π. It is defined as usual: two stores are equivalent up to π if they agree on rules (formulae) with privacy level less than or equal to π.

Let us informally recall our aim: we want to abstract from information leakage when they are declared in the security policy. I.e. when the leakage is originated by a declassified term. But we still want to prohibit other ones. To formalize this we introduce a new evaluation process. Roughly the idea is as follows: in the end, to ensure that a process p is safe we will have to prove that, for all privacy level π, the execution of p on two π equivalent stores F, F' will result in two π equivalent stores. This is clearly impossible in general when declassifiying functions are used. Indeed, one has just to consider the following example: let the identity *id* function be declared as declassifying by the security policy. For instance suppose that the specification of the security policy is $id(x) \to \bot$. Then, it is clear that two $\bot$-equivalent stores are not going to stay $\bot$-equivalent after action like $x := id(PIN)$. Thus, we introduce a new store where declassifying functions will be evaluated. The evaluation process is now the composition of two standard evaluations on two stores F, F': on one hand, non-declassified terms are evaluated as usual on F, on the other hand, declassified parts are evaluated in F'. Therefore we define $\mathfrak{s}\_eval(F, F')$ as the evaluation process which computes declassified parts (and their descendants) of a term using rules in F' and the other parts using rules in F. A *declassified operational semantics* based on this declassified evaluation is defined accordingly.

Let F, F' be two stores, and p a process term. We define the declassified operational semantics as a transition relation $\xrightarrow{F'}$. Transitions are of the form $\langle F, p\rangle \xrightarrow{F'} \langle F'', p'\rangle$ where the standard evaluation, namely $eval(F, t)$, is replaced by $\mathfrak{s}\_eval(F, F', t)$.

We are now in position to define the notion of secure process term w.r.t. a given security policy. For that, we introduce a notion of secure process term up to some privacy level π. A process term will be said secure up to π if its *behaviour*, limited to the effect done on data having a privacy level less than or equal to π, does not depend on data with a privacy level higher than π. Formally: a process p is secure up to privacy π, and two π-equivalent stores F_1, F_2 if $\langle F_1, p\rangle \longrightarrow \langle F_1', p'\rangle$ implies that: either $\exists F_2'$ such that $\langle F_2, p\rangle \xrightarrow{F_1} \langle F_2', p'\rangle$, $F_1' \cong_\pi F_2'$ and p' is secure up to privacy π and stores F_1', F_2', or $\langle F_2, p_2\rangle \not\xrightarrow{F_1}$ and for all $F_1^\sharp, p_1^\sharp$ such that $\langle F_1', p_1'\rangle \longrightarrow^* \langle F_1^\sharp, p_1^\sharp\rangle$ we have $F_1^\sharp \cong_\pi F_2$.

Thus, a process term is secure up to level π if within its runs data with privacy level higher than π do not influence data of a privacy status lower than π (stores stay π-equivalent) except when this leakage is authorized by the security policy. We have modeled the hypothesis done by the security policy by evaluating declassified terms on a single store. Note that the apparent distinction between F_1 and F_2 (the definition is not symmetric) is not significant since the definition of secure process must hold for any appropriate equivalent stores F_1, F_2.

We are now able to define precisely the meaning of being safe with respect to a given security policy. A process term is safe whenever it is secure for all privacy levels of $\mathfrak{L}$: a process p of a component is *secure* w.r.t. a security policy iff for all π, all stores F_1, F_2 such that $F \cong_\pi F_1 \cong_\pi F_2$, p is secure up to privacy level π and stores F_1, F_2.

An important point to notice is that when there is no declassifying functions declared in the security policy, our definition of secure process amounts to the one of [3]. Indeed, in this case it is easy to see that $\mathfrak{s}\_eval(F, F')$ is simply the standard evaluation $eval(F, t)$. In other words, it is only the declassified functions that introduce accepted information leakage. Moreover, if there are declassified functions, *but* none of them are used, then we are also back into a situation where security is defined in line with [3].

Safety analysis

Associated with this notion of safety we have developped an analysis algorithm based on some kind of abstract interpretation (in fact an abstract operational semantics). The abstract operational semantics defines new runs of abstract processes. These runs collect inequations over $\mathcal{SP}$ terms. Since the number of these inequations is finite, it is possible to produce the whole set of inequations for a process term. We proved that if a security policy $\mathcal{SP}$ is defined such that all inequations hold, then the analyzed process term is secure in the sense of our definition. That is to say, no unwanted interference happens.

The idea of our analysis is to collect the constraints computed by *all* possible abstract executions of a ground process term, say p. We claim that if the collected constraints are all valid in $\mathfrak{L}$, then the process term, p, is secure for the considered security policy. Crucial for the termination of our analysis is that the abstract store becomes stable during an abstract execution, i.e., after a certain point no more new privacy inequations are created.

Using the fact that the number of non equivalent abstract process calls is finite, we can prove that there are no infinite abstract runs. Since, in addition, the number of rules one may apply is always finite, we can define the result returned by our analysis as the collection of all reachable abstract stores. See [2] for the detailed proof of this claim.

Discussion

We have provided a formal definition of declared interferences. Our proposition enriches traditional approaches on security based on non-interference. With declared interference it is possible to take into account more real-life situations like password verifications, communication of encrypted data through a public channel etc. These situations, to our knowledge, have not been fully addressed from a noninterference point of view. Moreover, the definition of a security policy as a rewriting system is a novelty. It provides a tool to define very subtle security policies (for instance the privacy level of an encryption mechanism with respect to the status of the used key).

There are several recent works related to this paper [8, 14, 9] where the same problem is treated. Robust declassification idea is that attackers may not be able to control what information is released. It is not the same approach as ours since we try to take into account situations where the attacker may have such a control (e.g. brute force attacks on password checking) but only using some identified functions. Another difference comes from the fact that the analysis of [9] is defined for simple imperative programs, that is without concurrency while our analysis algorithm can handle concurrency. [8] is closest to our work, it also deals with declassification and how one can make exceptions to the information flow. The underlying formalization is intransitive noninterference and they apply it to an imperative programming-language with threads. In their formalism declassifications are declared to the level of actions, and it is possible to restrict declassifications to certain parts of the security lattice in the security policy. we differ from this approach in several ways. Firstly declassifications are defined at the level of functions, thus providing *security profiles* more elaborate than the definition of a downgrading from one to another level. Secondly, using rewrite rules to express security policies allows one to define more subtleties on security policies. Indeed we can compute on security levels, for instance the security level of a list may be defined as the join of the security level of its coponents. It is noticeable that our definition cope with the modified strong security condition of [8].

Future works include the study of the quantity of high level information leaked to low level (in line with [11] e.g.), which could also be declared in the security policy. An interesting direction to get closer to reality is to take into account a greater precision on security policies. For instance after three unsuccessful password tries, the system may prohibit a fourth try. It would imply a deeper use of rewriting rules (real computations, not only rules on privacy levels) in security policies.

1. REFERENCES

[1] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109 – 130, 2002. Special issue: "Merci, Maurice, A mosaic in honour of Maurice Nivat" (P.-L. Curien, Ed.).

[2] R. Echahed and F. Prost. Handling harmless interference (preliminary version). 2003. Avalaible at http://www-leibniz.imag.fr/LesCahiers/Cahier82/ResumCahier82.html.

[3] R. Echahed, F. Prost, and W. Serwe. Statically assuring secrecy for dynamic concurrent processes. 2003. proceedings of PPDP'03, preliminary version avalaible at http://www-leibniz.imag.fr/LesCahiers/2002/Cahier40/Resum-Cahier40.html.

[4] R. Echahed and W. Serwe. Combining mobile processes and declarative programming. In J. Lloyd et al., editors, *Proceedings of the* 1^{st} *International Conference on Computational Logic (CL 2000)*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 300 – 314, London, July 2000. Springer Verlag.

[5] R. Echahed and W. Serwe. Integrating action definitions into concurrent declarative programming. *Electronic Notes in Theoretical Computer Science*, 64, Sept. 2002. special issue: selected papers of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001).

[6] R. Giacobazzi and I. Mastroeni. Abstract non-interference. In *Proceedings of the* 31^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, Venice, Italy, Jan. 2004.

[7] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.

[8] H. Mantel and D. Sands. Controlled declassification based on intransitive noninterference. In *2nd ASIAN Symposium on Programming Languages and Systems*, 2004.

[9] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *17th IEEE Computer Security Foundations Workshop*, pages 172–186, 2004.

[10] A. D. Pierro, C. Hankin, and H. Wiklicky. Approximate confinement under uniform attacks. In M. V. Hermenegildo and G. Puebla, editors, *SAS'02 – Static Analysis, 9th International Symposium*, number 2477 in Lecture Notes in Computer Science, Madrid, Spain, September 2002. Springer.

[11] A. D. Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In *CSFW'02 –* 15^{th} *IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, Canada, 2002.

[12] P. Ryan, J. McLean, J. Millen, and V. Gilgor. Non-interference, who needs it ? In *CSFW'01 –* 14^{th} *IEEE Computer Security Foundations Workshop*, pages 237 – 238, Cape Breton, Nova Scotia, Canada, June 2001.

[13] G. Smith and D. M. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the* 25^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, pages 355 – 364, San Diego, Jan. 1998.

[14] S. Zdancewic. A type system for robust declassification. In *Annual Conference on the Mathematical Foundations of Programming Semantics*, 2003.

[15] S. Zdancewic and A. Myers. Robust declassification. In *Proceedings of 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Nova Scotia, Canada, June 2001., 2001.