Draft of an Introductory Survey to Dynamic Information Flow Security *

Gurvan Le Guernic
KTH – Royal Institute of Technology
Stockholm, Sweden

February 23, 2012

Abstract

Access control is the most common security mechanism used in order to enforce confidentiality and integrity security policies. However, as demonstrated by recent security issues regarding iPhone and Android, access control is not sufficient for today's personal computer security needs. The reason being that once access is granted no more control takes place. Modern personal devices may allow untrusted software to access personal data for local computation and access the web for advertisement reasons. In such cases, access control is unable to prevent personal data to be leaked to the web. Information flow security mechanisms are needed to enforce such policies. Information flow security mechanisms ensure that sensitive data is not sent to unauthorized recipient and is not altered by untrusted sources. This paper introduces the main notions related to information flow security and review the principal approaches followed in order to dynamically enforce information flow policies.

1 Introduction

Information leakage is a real world problem that can go pretty bad [49] and affect every body [23], even indirectly [51]. Access control is the most widely used technique to tackle this problem. However, no access control mechanism is bullet proof [3] or can protect against legitimate users. A survey [53] showed that for 2006, except for physical theft (46%), the majority of registered data breaches were due to incompetence (28%) or insider malfeasance (7%), rather than outside hackers (19%). The situation is even worse with add-ons [43] or smartphone applications [63]. Once the access has been granted, there is no more real control of the dissemination of data. A process may need to access sensitive data in order to fulfill its goal. A user may agree to allow this process to access confidential data for the fulfillment of the process purpose. However,

*This work was partially supported by the EU-funded FP7-project HATS (grant № 231620).

it is likely that the owner does not want the process to spread the information it has been given access to. Unfortunately, trusted application sources [27] or access control do not help in this case. Those are the reason why, in addition to access control, user-centric end-to-end information flow control mechanisms are of high interest to protect data even after access has been granted.

The goal of information flow control is to ensure the sole existence of safe, or secure, information flows in a process, or more generally in an information system. There exist different properties defining different levels of secure information flows. In the language-based information flow security community, the most widely used formal property for describing "secure information flows" is *noninterference*. A process is said to be noninterfering if and only if an attacker is unable to deduce information about the secret manipulated by looking only at the publicly observable outputs of the process. Section 2 formalizes the main notions used in information flow security. Variants of noninterference can be and are applied to system-wide security. Different techniques exist to verify or ensure that a process or system is noninterfering. This survey focuses on dynamic mechanisms, for more information on static language-based mechanisms the reader is referred to Sabelfeld and Myers survey [58]. Section 3 explores the state-of-the-art of dynamic information flow security mechanisms described at the computing base level. Those mechanisms apply at different levels, mainly binary code or system events. They take the form of monitors and involve enhanced execution environments (to the extent of enhanced hardware). Based on those techniques, practical tools have been proposed, such as the architectural frameworks RIFLE [64] and Raksha [14]. The majority of information flow security mechanisms applying on the computed load are compile-time analyses. Those analyses rely on different well-known techniques as type systems, abstract interpretation or constraint resolution [58]. Fewer mechanisms applying on the program under scrutiny are dynamic analyses. Those are described in Sect. 4. They can serve as the basis for the development of special purpose interpreters, program transformation, or testing environments. Finally, Sect. 5 concludes this survey. The vast majority of work presented in this paper is theoretical, but practical tools are starting to emerge. With the increase of personal digital devices carrying personal data and requiring to communicate *specific data complying with different security policies* to the outside world, the need for security mechanisms at the granularity of information flow increases; as well as the public interest for such mechanisms, as attested by the lawsuits against Apple [54] and Google [41], or the good coverage received in generalist medias [4, 40] by TaintDroid [16] (a recent information flow security mechanism).

2 Information Flow Security

What does it mean for a system to be safe with regard to information flows? There is no intrinsic definition of "safe information flow". An atomic definition of "safety" for information flows can be useful for security mechanisms checking or enforcing safe information flow behaviors. However, as information flows

are highly combinable, an atomic definition of "secure information flow" is not helpful for end-users trying to understand the level of security provided by a given system. Therefore, a formal definition at the system level of what are secure information flows is required. There exist different system level properties defining the notion of security with regard to information flows.

In 1973, Lampson introduces the notion of *confinement* [32]. A confined process has really limited capabilities to interact with the rest of the system and with the outside world. It is a general notion aiming at enforcing that a process or information system does not unduly leak confidential data. Lampson states that a process can be confined only if it is not able to maintain secret information longer than its own execution. Such processes are called *memoryless* [19]. Additionally, it is required to call only secure or confined programs, and to communicate with the outside world using masked outputs. A masked output can only send values belonging to a predefined set of safe output values. Lipner [37] explores the available techniques which can be used to confine a process. Sandboxing [22] is a popular generic technique to enforce a confined execution of a program [52]. However, even if confinement is really restrictive, which could be a problem by itself, masked outputs do not guaranty absence of leakage. The allowed output values can be reused as a new alphabet to encoded disallowed output values.

Cohen [12] introduces the notion of *strong dependency*. It borrows from classical information theory [2] the idea that there exists an information flow from input i to output o in a process P whenever *variety* in i is conveyed to o by the execution of P. In other words, for deterministic processes, o is strongly dependent [12] on i if and only if there exist at least two executions of P whose inputs differ only in i and whose outputs differ in o. Goguen and Meseguer [21] define their notion of *noninterference* as the absence of strong dependencies. A process, P, is said to be *noninterfering* if the values of its public (or low) outputs do not depend on the values of its secret (or high) inputs. Formally, noninterference of P is expressed as follows: given any two initial input states σ_1 and σ_2 that are indistinguishable with respect to low inputs ($\sigma_1 =_L \sigma_2$), the executions of P started in states σ_1 and σ_2 are *low-indistinguishable* ($[\![\sigma_1 \vdash \mathtt{P}]\!] \sim_L [\![\sigma_2 \vdash \mathtt{P}]\!]$); i.e. there is no observable difference in the public outputs. Noninterference for a process P is then stated as follows:

$$\forall \sigma_1, \sigma_2 : \ \sigma_1 =_L \sigma_2 \ \Rightarrow \ [\![\sigma_1 \vdash \mathtt{P}]\!] \sim_L [\![\sigma_2 \vdash \mathtt{P}]\!]$$

In the simplest form of the *low-indistinguishable* definition, public outputs include only the final values of low variables. In a more general setting, the definition may additionally involve intentional aspects such as power consumption, computation times, etc.

Some variants of the notion of noninterference have been proposed. For example, Giacobazzi and Mastroeni [20] define a notion of *abstract noninterference*. It states that there is no strong dependency between some properties of the secret inputs and some properties of the public outputs; for example, their parity or sign. The properties on outputs are those that the attacker can distinguish. The properties on inputs are those that must be kept secret.

Noninterference is a strong and precise high level definition of secure information flows. It is non decidable in general [26], and sometimes stronger than needed. However, the main idea underlying it defines the general notion of secure information flow used in the remainder of this chapter: secret (resp. untrusted) data should not influence publicly observable (resp. trusted) data.

2.1 Defining information flow

Information must not be considered equivalent to data. As stated by Ashby [2], a data carries more information than its intrinsic value. For example, if someone is allowed a single word to describe the dominant color of a picture, if the data "green" is used, the majority of people will not only get from this word that the picture is dominantly green, but also that it is *potentially* a landscape picture and that, the picture is *definitely* not a black and white one. Pushing further, if people receiving the data "green" know from which set of pictures the described one comes from, then they may even know precisely which picture it is. In this example, the *data* carried by the message is "green" but the *information* it carries depends on the context and is usually more important than the intrinsic information contained in the data "green". The techniques described in this chapter are generally interested in tracking information flows and not only data flows. One key consequence is that information can flow along more channels than data.

Defining information channels In a computer-based system, information can flow along many different channels; such as registers, file system, computation time, or even energy consumption. In order to fully understand which level of protection is given by an information flow security mechanism, it is important to know and understand which type of channels the security mechanism takes into consideration.

Lampson [32] defines three different categories of channels which can be used by unsafe information flows. The first one is the category of *legitimate channels*. Such channels are based on mechanisms intended for information transfer which are needed for the legitimate operation of the information system. For example, a web browser needs to have access to the Internet. However, an information flow control mechanism must prevent the web browser to create an undesired flow from confidential data on the computer to a particular web site.

The second category concerns *storage channels*. Such channels make use of mechanisms which are not primarily intended to transfer data, but to store it. The information is first transferred from the source of the flow to a storage location and then, later, from the storage location to the destination of the flow. Such channels are used to attempt to fool the information flow control mechanism by delaying in time and space the realization of the undesired flow.

The last category of channels contains the so called *covert channels*. Such channels use mechanisms which are not intended for the manipulation (transfer, computation or storage) of information. They are based on the encoding and

transfer of information from the source of the flow into the side effects of legitimate mechanisms. Information contained in those side effects are then decoded and transferred to the destination of the flow. For example, such channels can use locks, intended for preventing race conditions, or even the computation time of a given process.

It is in general impossible to construct an information flow control mechanism which is both sound and complete [26]. The channels which must be overseen by an information flow control mechanism, and the constraints which must be put on them, depend on the desired equilibrium between the security of secret and/or trusted data and the efficiency and capabilities of the information system.

Classifying information flows The majority of information flows between data containers can be classified along two dimensions: direct/indirect and explicit/implicit. Direct flows from A to B use legitimate channels intended for data transfer between A and B, such as variable assignment. Whereas indirect flows use channels which are not intended for data transfer. They usually rely on a third entity as file locks or the program counter itself. For example, branching statements test an expression to decide which branch will be executed. If a branch contains a data transfer to a container A then its data is replaced by a new one, and this fact gives information about the value of the test expression. Therefore, an information flow from the test expression to A occurs through the program counter. This is an indirect flow.

Explicit flows are created by the occurrence of a specific event, usually a data transfer. Whereas implicit flows are created by the fact that a specific event does not occur. For example, getting back on the branching statement example, once it is known that the branching statement has been executed, whereas the container A contains its previous data or the new one, an indirect information flow from the test expression e to A exists (it is possible to deduce the value of e from the data contained in A). If the data transfer to A takes place then the flow from e to A is explicit, otherwise it is implicit. This applies for any event once it is know that the event should have already occurred if it had to. For example, if a program terminates in less that t seconds if a value v is true and run for ever if v is false, then an indirect flow from v to an entity u observing the termination of the program will ultimately exists whatever the value of v. Once the program terminates, an explicit indirect flow from v to u exists. Once the program has run for more that t seconds, an implicit indirect flow from v to u exists.

All works on information flow do not agree on usage of those terms. This is mainly due to the fact that some techniques do not take into consideration implicit flows; or that others, as static techniques, consider all potential executions without distinguishing particular executions, or traces, and therefore do not distinguish between explicit and implicit flows as described above. An implicit flow can exist in a given execution due to the nonoccurence of an event e, if and only if there exists at least another execution for which the event e

occurs. As a consequence, an *implicit* flow can occur in a given execution only if it is *explicit* for at least another execution. Therefore, a static mechanism, handling all the potential executions of a process at once, is not required to do an explicit distinction between "explicit" and "implicit" flows. Usually, those works use the terms "direct" or "explicit" for the notion called "direct" in this chapter, and the terms "indirect" or "implicit" for the notion "indirect".

3 Computing-Base Mechanisms

Some information flow security mechanisms are described at the level of the computing base. They are supported either by an "hardware" extension of the processor or of a virtual machine (VM), or by an extension of the operating system. They analyze the events triggered by the execution of the process under scrutiny, and not by an analysis of the process itself. The major advantage of such mechanisms is that they can usually be applied to any process without requiring or modifying their source code. They can therefore handle the majority of existing programs as well as dynamically generated code.

3.1 Processing Level Security Mechanisms

Information flow security mechanisms applying on the processing unit associate a security level, sometimes called *data mark*, to every storage location of the machine. Storage locations designate, for example, the registers, the program counter or every memory word. In the case of an "hardware" extension, the security labels are often incorporated into the storage locations themselves. Additionally, the computing machinery is extended so that every atomic operation, in addition to its normal behavior, propagates the security levels according to the security policy enforced. In some cases, additional low level code is added to the program so that new variables contain the security labels of the original storage locations.

Taint analyses The main drawback of a description at the computing machinery level is the difficulty to deal with implicit indirect flows, i.e. information flows created by pieces of code which *are not executed*. Therefore, several papers restrict themselves to tracking direct flows and, to some extent, explicit indirect flows [13, 73]. Those analyses are sometimes called *taint analyses* [60], and share many principles with *data flow analyses* [59, 28]. They look at the problem of security under the aspect of integrity and do not take care of information flowing indirectly because two branches of a computation do not modify the same variables. Their aim is usually to prevent an attacker to gain control of a system by giving *spurious* inputs to a program which may be buggy but is not malicious. They are used to prevent attacks based, for example, on buffer overflow or format string vulnerabilities as those used by some worms or Trojan horses.

Haldar et al. [24] develop a plug-in to add labels at the object granularity to existing JVM. Each time an object O_1 reads a data store in an object O_2, its label climbs up the flow lattice in order to be higher than the label of O_2. Yoshihama et al. [73] follow a different approach by duplicating the JVM. The copy of the JVM works on the security labels of data manipulated by the original JVM. A reference monitor is included in the original code in order to synchronize the two JVM. Suh et al. [62] propose a low overhead extension of every register and memory location in the processor with an additional bit for integrity, but they do not consider any kind of indirect flows. Similarly, Raksha [14], Panorama [71], and Minos [13] extend a processor registers and memory words with a 1 bit data mark to track spurious data. Implemented as an enhanced *SimpleScalar* processor simulator [9], the work by Shuo Chen et al. [11] considers that, each time a tainted pointer or data is dereferenced, it is due to an attack. Chen et al. states that this allows them to protect users against a wider spectrum of control data and non-control data attacks. Instead of extending a processor with data marks, Haibo Chen et al. [10] track information flows using existing special features of modern processors (in this case, *deferred exception tracking* and *speculative execution* on the Itanium processor). This different approach allows them to claim only 1% of overhead for server applications. A few months ago, Enck et al. [16] implemented a taint analysis inside the VM of Android in order to track how applications handle sensitive data. They claim an overhead of only 14%.

Dealing with implicit indirect flows is difficult Yet, some computing machinery techniques take into consideration implicit indirect flows [17, 57, 8, 64]. This feature is necessary when dealing with confidentiality issues against malicious programs. Fenton [17] describes the addition of data marks to the abstract computer model of Minsky [42]. Data marks are fixed, except for the program counter's data mark which is computed dynamically. This means that storage locations can either contain only secret information or only public information. Fenton [17] shows that, assuming public registers are accessible only at the end of the computation, its abstract machine is secure with the fixed data marks, but would not be with variable ones. Saal and Gat [57] and Brown and Knight [8] describe, without formally proving something similar to noninterference, an abstract machine in which some storage locations have a fixed data mark and others have a dynamically computed one.

The basic ideas behind those three works [17, 57, 8] are the same. When storing a value v to a fixed data mark storage location fl, the machine checks that the data mark of fl is higher or equal to the least upper bound of the data mark of v and the one of the program counter. If it is not the case then the storage operation is ignored, i.e. as if the operation was NOP. When storing a value v to a dynamic data mark storage location dl, the machine updates the data mark of dl to the least upper bound of the data mark of v and the one of the program counter. With just this mechanism, the program counter's data mark will monotonically increase at each conditional jump operation. It

will then end up with the highest possible data mark, and the machine will not be able to compute anything useful. In order to allow the machine to safely decrease the data mark of the program counter and come back to a previous level, it is possible to push a pair, composed of the program counter's data mark and a return address, into a stack. Then, it is possible to pop this pair from the stack and set the program counter's data mark and value to those in the pair previously pushed into the stack. This feature allows a program to execute a procedure, whose behavior depends on a secret value, and still assign values to low level data mark locations after returning from the procedure call.

The machines of Fenton [17], Saal and Gat [57] and Brown and Knight [8] do not seem to take into consideration implicit indirect flows. So, how can they be secure from a point of view similar to noninterference? An implicit indirect flow from an origin o to a destination d exists whenever an assignment to d is not executed because of the value of o. As any flow, it causes insecurity whenever o has a higher level than d and d does not influence the low-observational behavior of the program the same way depending on the fact that the previous assignment has been executed or not. With the machines described previously, this type of flows does not exist on fixed data mark when o has a higher level than d because in that case, even if the assignment is executed, the value of d is not modified. So, the machine of Fenton [17], contrary to what Venkatakrishnan et al. [66] state, securely handles implicit indirect flows. Problems arise when the destination of the flow has a dynamic data mark. It is then impossible to take into consideration the implicit indirect flow to increase d's data mark to a level at least as high as the one of o because the machine does not see the operation causing this flow. To solve this problem, Saal and Gat [57] propose to push into the stack the values and data marks of every dynamic data mark storage location whenever the program counter is pushed into the stack. Whenever the program counter is popped out of the stack, the values and data marks of every dynamic data mark storage location are reset to their values at the time the program counter was popped. Therefore, assignments, which have been executed or could have been executed since the last program counter push, do not have anymore influence on the dynamic data mark storage locations. It is then perfectly safe with regard to implicit indirect flows.

Vachharajani et al. [64] propose a mix of hardware extension and binary instrumentation. They propose an architectural framework, called RIFLE, to enforce confidentiality. As in other works, the machine described extends every storage location, general purpose registers and memory words, with a data mark. The semantics of instructions is extended to update data marks in accordance with direct flows and a special set of registers dedicated to security labels. The original binary code is instrumented in order to encode any indirect flows using the security label registers. RIFLE provides a virtual machine implementing this new semantics.

3.2 System Level Security Mechanisms

Still at the computing base level, some information flow security mechanisms are described at the operating system level. Those dynamic analyses are in charge of controlling principals' (users or processes) actions and prevent those actions that may break the secure information flow policies. Such analyses do not study program code, either source or binary. They usually monitor principals' actions to prevent them to create unsafe information flows.

In 1969, Weissman [68] describes a security control mechanism which dynamically computes the security level of newly created files. In this mechanism, a *security level history* is associated to each job running on the computer. This level monotonically increases as the job opens existing files. However, it is bound by the security level of the job itself. The security level history is used when labeling newly created files. In this mechanism, the security level of the information output by a given job is approximated as the least upper bound of the security level of previously accessed files. Asbestos [65] and HiStar [74] are modern OS evolutions of this approach. In his thesis, Rotenberg [55] describes the hardware and software of the *Privacy Restriction Processor*, a machine similar to Weissman's work [68]. Extending this approach, Woodward [69] presents its *floating labels* method. This method deals with the problem of over-classification of data in computer systems implementing the MAC security model. To improve on this problem, objects receive two types of security levels. One is the upper bound of the security level this object is allowed to contain. The other one is the security level of what it really contains. This latter level is dynamically computed using a mechanism similar to the security level history. Flume [30] is slightly different from the approaches presented above. Instead of dynamically computing the security level of processes and system objects, processes and system objects have an initial label. Flume lets processes modify their label (to a limited extent specified by the system configuration) and check, for any information flow, that flows from the source label to the destination label are authorized. This ability for processes to modify their label allows them to declassify and endorse data. This gives more flexibility to the system, but it becomes more difficult for the system administrator to understand the overall end-to-end information flow policy enforced by the system.

Nagatou and Watanabe [45] propose a monitoring mechanism to detect and prevent unauthorized information flows through covert channels in a system serving multiple principals. For example, they try to prevent a principal to transmit sensitive data to a lower security level principal p_2 by deleting files for which p_2 has read access. The inputs to the monitoring mechanism are events. An event reflects an action that a principal is attempting. Each event is associated a security level which reflects the security level of the information carried by the action. This level is approximated by the security level of the user generating the action. The definition of information flow security used is similar to Goguen and Meseguer's notion of noninterference [21]. An action of security level l is noninterfering if and only if the system would have reacted the same way if all the previous actions whose security level is higher than l had not

been executed. The monitoring mechanism is based on an emulator for every security level l which emulates the state of the system if only the actions of a security level lower or equal to l had been executed. An action of security level l is authorized if the system and the emulator for the level l react the same way to the action. Following a related approach but with a process granularity rather than whole system, Devriese and Piessens [15] propose to execute a process once for each security level in parallel. I/O operations of level l are operated by the execution at level l. Processes share computation results with other processes allowed to access it. Interestingly, they show that protected execution may even be faster than unprotected ones due to the automatic parallelization.

Following an approach inspired by Bell and LaPadula's work [6, 5], Zimmerman et al. propose an intrusion detection system based on information flow control called BLARE [76]. The system works by monitoring information flows between system objects (files, pipes, sockets, shared memory buffers, ...). If an illegal information flow occurs, the system considers that it is due to an illegal intrusion. An initial set of legal flows between system objects is provided to the system. In order to prevent an attacker to create illegal flows by combining initially legal flows, BLARE updates the set of legal flows after each occurrence of a legal information flow. However, BLARE consider processes as black boxes and, in order to be conservative, that a process creates an information flow from every input to every output. Hiet et al. [25] notice that this approximation leads BLARE to trigger to much false positives. They propose to combine BLARE with JBLARE, a mechanism tracking information flows inside Java programs. This type of information flow security mechanisms is studied in the next section.

4 Computed-Load Mechanisms

Many works have been done on software information flow security techniques described at the computed load level. In 2003, Sabelfeld and Myers published a good survey on mainly static language-based mechanisms for information flow security [58]. The main advantage of security mechanisms described at the language level is that they can rely on a more precise semantics of the code of the process under scrutiny. This gives those techniques a better understanding of how information flows during the process execution. The majority of language-based information flow security techniques are based on static analyses, but some dynamic analysis have started to emerge in the recent years. Sabelfeld and Myers's survey [58] and the bibliography of Le Guernic's thesis [34] give additional references to language-based information flow security.

Dynamic techniques for information flow security aim at tracking at run time the information flows occurring in the specific execution under evaluation. Some of them are described at the language level. The development of language-based dynamic information flow analyses is more recent than for static analyses. With dynamic analyses, it is usually more difficult to achieve a precision level equivalent to the one provided by static analyses. This is due to

the fact that some pieces of code are not executed by an execution, but those unexecuted statements can still be the source of information leakage due to implicit flows. This fact implies that dynamic analyses are required to use some sort of static analyses of unexecuted code to approximate their influence on the overall information flows occurring during the execution. Moreover, dynamic analyses have to deal at run time with unsafe information flows. Depending on how the dynamic mechanism reacts to the discovery of unsafe information flows, new covert channels may be created [36]. Those new covert channels are usually not taken into account by the dynamic mechanism. Therefore, dynamic analyses have to be really careful with the way they deal with unsafe information flows in order to avoid unwanted information leakage. However, dynamic analyses still have some appealing advantages over static analyses. They are usually more user-centric in the sense that final users have more control over the precise security policy which has to be enforced by the security mechanism. Additionally, they allow to safely attempt to use some of the safe executions of a program which can not be proved to be safe for all its executions.

Dynamic information flow techniques described at the language level can follow two approaches: execution monitors and program transformation. *Execution monitors* track at run time the security level of data. Every data container is associated with a security label, or even policy objects [72]. This label is updated each time the value of the container is modified, or may have been modified if a secret value was different. Every time a sensitive action occurs, either publicly display a value or trigger operation requiring high level of trust, a policy manager verifies that the security labels of values involved are "safe enough". If it is not the case, a recovery action is taken in place of the attempted unsafe operation. Language-based execution monitors are described by providing a special semantics which can serve for the development of a special interpreter or as the theoretical basis of a program transformation. *Program transformations* for information flow security often rely on a simple flow-sensitive static analysis to obtain the security level of variables at some key program points. Instruction manipulating security level values are then added to the program to handle more complicated information flows. Many program transformations for information flow security can be seen as monitor inlining.

Taint and need analyses The main drawback of dynamic techniques is the difficulty to deal with implicit indirect flows (i.e. information flows created by pieces of code which *are not executed*). Therefore, several papers restrict themselves to tracking direct flows and, to some extent, explicit indirect flows [47, 50]. This type of analysis is sometimes called *taint analysis*. It looks at the problem of security under the aspect of integrity and does not take care of information flowing indirectly through branching statements containing different assignments. Its aim is usually to prevent an attacker to gain control of a system by giving *spurious* inputs to a program which may be buggy but is not malicious. It is used to prevent attacks based, for example, on buffer overflow or format string vulnerabilities as those used by some worms or Trojan

horses. Even if those analyses have some practical use, they are not fit to deal with confidentiality seen as secure information flow. Extending standard taint analyses, Xu et al. [70] propose a taint analysis for C. Their analysis takes into account direct flows, as any taint analysis, but also some explicit indirect flows. However, because they claim it is not necessary for the type of attacks they want to prevent, their dynamic analysis does not take into consideration all explicit indirect flows or any implicit indirect flow. Recently, Lam and Chiueh [31] proposed a framework for dynamic taint analyses which, however, does not take any type of indirect flow into consideration. The notion of *need* is similar to the notion of taint. A need analysis determines what parts of a "program" are needed, or required, to "evaluate" this program. For a λ-term t_λ, a need analysis determines which sub-terms of t_λ are needed in order to be able to evaluate t_λ to normal form. Gandhe et al. [18] and Abadi et al. [1] propose two dynamic need analyses based on labeled λ-calculi. However, such analyses do not have to track implicit indirect flows. Therefore, the dynamic analyses described at the code level presented thus far are not powerful enough to be used for checking properties similar to noninterference.

There exist some real-world language-based tools to ensure information flow security. The most well known is the scripting language *Perl* which includes a *taint* mode [7, 67] to enforce simple integrity policies based on a dynamic data flow analysis. In this mode, the direct information flows originating with user inputs are tracked. It is done in order to prevent the execution of "bad" commands. However, this analysis does not take any indirect flows into consideration.

Trying to handle indirect flows Some instrumented semantics, used as the bases for the development of static analyses, are good candidates as dynamic information flow analyses. For example, Ørbæk [48] gives an instrumented semantics of a first order language with pointers before extracting from it two static *trust* analyses. The notion of trust is equivalent to a stronger notion of taint. It aims at determining which output values can be trusted to be exact even if some input values are not trusted to be exact. The instrumented semantics is a standard semantics with the addition of label to values. The evaluation rules are accordingly modified in order for the labels to reflect the trustworthiness of their associated value. The instrumented denotational semantics can be seen as a dynamic analysis of `trust`.

Other approaches fail short of providing a sound information flow monitor because they do not study enough the effect of the correction mechanism on the overall information flows. Venkatakrishnan et al. [66] propose an information flow security program transformation for a simple deterministic procedural language. The transformation returns a program in which every original variables is associated a boolean variable which reflects its level of security. The correction mechanism chosen is to stop the execution as soon as an output of a secret data on a public channel is detected. Moreover, their transformation does not approximate information flows with the same precision in a branch which

is executed and in a branch which is not executed. Similarly, Masri et al. [38] present a dynamic information flow analysis for structured or unstructured programs. Their algorithm achieves a good level of precision for a quite complete language. However, it does not study deeply the dynamic correction of "bad" flows and lacks formal statements and proofs of the correctness of the correction mechanism. It suffers from misconception similar to those of the transformation of Venkatakrishnan et al. [66]. For example, the dynamic correction mechanism proposed is to stop the execution as soon as a potential flow from a secret data to a public *sink* is detected. As explained by Le Guernic and Jensen [36], not achieving the same precision depending on the branch executed or stopping the execution too early can create new covert channels leaking secret information.

More recently, Shroff et al. [61] proposed an interesting approach to handle *implicit* indirect flows (indirect flows whose assignments which create them are not executed, as opposed to *explicit* indirect flows) in a language including alias and method calls. It relies on the simple idea that an implicit indirect flow can exist in an execution if and only if there exists another execution for which this indirect flow is explicit. Therefore, the proposed dynamic information flow analysis tracks direct flows and collects explicit indirect flows dynamically. The information collected about indirect flows is transferred from one execution to another using a cache mechanism. After an undetermined number of executions, the analysis will know about all indirect flows in the program and thus will then be sound with regard to the detection of all information flows. In order for their mechanism to be sound from the beginning, they propose an extension relying on a prior static analysis of the indirect flows.

In his thesis [34] and other papers, Le Guernic proposes different information flow monitors which are proved sound with regard to the notion of noninterference. One of them, based on an automaton tracking information flows, handles concurrent programs [33]. Another one, based on a special purpose semantics, uses the values computed at run time for the original variables of the program in order to increase its precision [35].

McCamant and Ernst [39] propose a *quantitative* [44] dynamic information flow analysis for sequential programs. This type of analyses can help eliminate false positives [46] triggered by analyses handling indirect flows (sometimes called implicit flows) [29].

5 Conclusion

Information flow security policies are the bases of strong and appealing confidentiality and integrity properties. Access control mechanisms are widely deployed in today's information systems. They are quite efficient to guaranty that a user accessing sensitive data has the legitimate right to do so. However, once access has been granted, there is no more verification about what happens to the sensitive data. Information flow security mechanisms rely on strong information theories. A security label is associated to every persistent data in the system and inputs of processes. During processing tasks, newly created data receive a

security label reflecting the sensitivity of data used for their creation. Whenever a sensitive operation occurs, the security label of data involved is verified to ensure that no "bad behavior" will occur, releasing secret information to unauthorized users or relying on untrusted information for the operation of sensitive tasks. This constant involvement of the security mechanism is the reason why information flow security policies are considered end-to-end security policies.

Dynamic information flow security mechanisms can be implemented at different levels. Some apply on the computing base, either the processing unit or the operating system. Others apply on the computed load, dynamically analyzing the program to execute. Mechanisms applying on the processing unit require the development of a new processor or virtual machine, but usually allow to securely execute existing programs without modifying them and with a good precision with regard to integrity concerns. Modified operating systems can be executed on existing hardware, but, as processes are usually seen as black boxes, the level of precision is too coarse grain and too much false positives are triggered in many cases. Dynamic program analyses are usually sound and therefore do not trigger false negatives, however they are often too restrictive and may be difficult to deploy on everybody's information system. In order to build an efficient information flow security mechanism, the solution is probably to combine mechanisms deployed at different levels in a way similar to what is done, for example, by RIFLE [64], Laminar [56] or TaintEraser [75].

References

[1] M. Abadi, B. W. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *Proc. Int. Conf. on Functional Programming*, pages 83–91, 1996.

[2] W. R. Ashby. *An Introduction to Cybernetics*. 1956.

[3] D. Barrett. Hackers penetrate nasdaq computers. The Wall Street Journal (Tech) website, 2011.

[4] BBC. Google android apps found to be sharing data. BBC News website, 2010.

[5] D. E. Bell and L. J. a. L. LaPadula. Secure Computer Systems: A Mathematical Model. Technical Report MTR-2547, Vol. 2, MITRE Corp., 1973.

[6] D. E. Bell and L. J. a. L. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., 1973.

[7] G. Birznieks. Cgi/perl taint mode faq, 1998.

[8] J. Brown and T. F. Knight, Jr. A minimal trusted computing base for dynamically ensuring secure information flow. Technical Report ARIES-TM-015, MIT, 2001.

[9] D. Burger and T. Austin. The simplescalar tool set.

[10] H. Chen, X. Wu, L. Yuan, B. Zang, P.-c. Yew, and F. T. Chong. From speculation to security: Practical and efficient information flow tracking using speculative hardware. *SIGARCH Comput. Archit. News*, (3):401–412, 2008.

[11] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Proc. Dependable Systems and Networks*, pages 378–387, 2005.

[12] E. S. Cohen. Information Transmission in Computational Systems. *Operating Systems Review*, (5):133–139, 1977.

[13] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proc. Symp. Microarchitecture*, pages 221–232, 2004.

[14] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proc. Symp. Computer Architecture*, pages 482–493, 2007.

[15] D. Devriese and F. Piessens. Noninterference through Secure Multi-execution. In *Proc. Symp. Security and Privacy*, pages 109–124, 2010.

[16] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. Symp. Operating Systems Design and Implementation*, pages 1–6, 2010.

[17] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, (2):143–147, 1974.

[18] M. Gandhe, G. Venkatesh, and A. Sanyal. Labeled lambda-calculus and a generalized notion of strictness (an extended abstract). In *Proc. Asian C. S. Conf. on Algorithms, Concurrency and Knowledge*, pages 103–110, 1995.

[19] I. Gat and H. J. Saal. Memoryless execution: A programmer's viewpoint. *Software: Practice and Experience*, (4):463–471, 1976.

[20] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc Symp. Principles of programming languages*, pages 186–197, 2004.

[21] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. Symp. Security and Privacy*, pages 11–20, 1982.

[22] L. Gong. Java security: present and near future. *IEEE Micro*, (3):14–19, 1997.

[23] N. S. Good and A. Krekelberg. Usability and Privacy: A Study of KaZaA P2P File Sharing. In *Security and Usability: Designing Secure Systems that People Can Use*, chapter 33, pages 651–668. 2005.

[24] V. Haldar, D. Chandra, and M. Franz. Practical, dynamic information flow for virtual machines. In *Proc. W. Programming Language Interference and Dependence*, 2005.

[25] G. Hiet, L. Mé, B. Morin, and V. V. T. Tong. Monitoring both os and program level information flows to detect intrusions against network servers. In *Proc. W. Monitoring, Attack Detection and Mitigation*, 2007.

[26] A. K. Jones and R. J. Lipton. The enforcement of security policies for computation. In *Proc. Symp. Operating System Principles*, pages 197–206, 1975.

[27] Kevin. Update: Security alert: Droiddream malware found in official android market. Official Lookout Blog, 2011.

[28] U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. 2009.

[29] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *Proc. Conf. Information Systems Security*, pages 56–70, 2008.

[30] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standrad os abstractions. In *Proc. Symp. Operating Systems Principles*, pages 321–334, 2007.

[31] L. C. Lam and T.-c. Chiueh. A general dynamic information flow tracking framework for security applications. In *Proc. Computer Security Applications Conf.*, pages 463–472, 2006.

[32] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, (10):613–615, 1973.

[33] G. Le Guernic. Automaton-based Confidentiality Monitoring of Concurrent Programs. In *Proc. Computer Security Foundations Symp.*, 2007.

[34] G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, 2007.

[35] G. Le Guernic. Precise Dynamic Verification of Confidentiality. In *Proc. Verification W.*, 2008.

[36] G. Le Guernic and T. Jensen. Monitoring Information Flow. In *Proc. W. Foundations of Computer Security*, pages 19–30, 2005.

[37] S. B. Lipner. A comment on the confinement problem. In *Proc. Symp. Operating System Principles*, pages 192–196, 1975.

[38] W. a. W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. In *Proc. Symp. Soft. Reliability Engineering*, pages 198–209, 2004.

[39] S. McCamant and M. D. Ernst. A simulation-based proof technique for dynamic information flow. In *Proc. W. Programming Languages and Analysis for Security*, 2007.

[40] E. Mills. What's that android app doing with my data? CNET news website, 2010.

[41] E. Mills. Google sued over Android data location collection. CNET news website, 2011.

[42] M. L. Minsky. *Computation: Finite and Infinite Machines*. 1967.

[43] Mozilla. Add-on security vulnerability announcement: Mozilla sniffer. Mozilla's Add-ons blog, 2010.

[44] C. Mu. Quantitative information flow for security: a survey. Technical Report TR-08-06, King's College London, 2008.

[45] N. Nagatou and T. Watanabe. Run-time detection of covert channels. In *Proc. Conf. Availability, Reliability and Security*, pages 577–584, 2006.

[46] J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *Proc. W. Programming Languages and Analysis for Security*, pages 73–85, 2009.

[47] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. Network and Distributed System Security Symp.*, 2005.

[48] P. Ørbæk. Can you trust your data? In *Proc. Conf. Theory and Practice of Software Development*, pages 575–590, 1995.

[49] L. Page. Smut-swapping sailors leak secret missile specs. *The Register*, 2007.

[50] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proc. Symp. Microarchitecture*, pages 135–148, 2006.

[51] S. Ranger. Tk maxx customers urged to check card statements. *Silicon.com*, 2007.

[52] C. Reis, A. Barth, and C. Pizano. Browser security: lessons from Google Chrome. *Commun. ACM*, pages 45–49, 2009.

[53] B. Rosenberg (Privacy Rights Clearinghouse). Chronology of data breaches 2006: Analysis, 2007.

[54] J. Rosenblatt. Apple sued over applications giving information to advertisers. Businessweek website, 2011.

[55] L. J. Rotenberg. *Making Computers Keep Secrets*. PhD thesis, Massachusetts Institute of Technology, 1973.

[56] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: practical fine-grained decentralized information flow control. In *Proc. Conf Programming Language Design and Implementation*, pages 63–74, 2009.

[57] H. J. Saal and I. Gat. A hardware architecture for controlling information flow. In *Proc. Symp. Computer Architecture*, pages 73–77, 1978.

[58] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. on Selected Areas in Comm.*, (1):5–19, 2003.

[59] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proc Symp. Principles of programming languages*, pages 38–48, 1998.

[60] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). *Proc. Symp. Security and Privacy*, pages 317–331, 2010.

[61] P. Shroff, S. F. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proc. Computer Security Foundations Symp.*, 2007.

[62] G. E. Suh, J. W. Lee, D. X. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proc. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2004.

[63] S. Thurm and Y. I. Kane. Your Apps Are Watching You. The Wall Street Journal (Tech) website, 2010.

[64] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proc. Symp. Microarchitecture*, 2004.

[65] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst.*, 2007.

[66] V. N. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *Proc. Conf. Information and Communications Security*, pages 332–351, 2006.

[67] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*, section Handling Insecure Data, pages 558–568. 2000.

[68] C. Weissman. Security controls in the adept-50 timesharing system. In *Proc. AFIPS Fall Joint Computer Conf.*, pages 119–133, 1969.

[69] J. P. L. Woodward. Exploiting the dual nature of sensitivity labels. In *Proc. Symp. Security and Privacy*, pages 23–31, 1987.

[70] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proc. Security Symp.*, pages 121–136, 2006.

[71] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proc. Conf. Computer and Communications Security*, pages 116–127, 2007.

[72] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proc. Symp. Operating Systems Principles*, pages 291–304, 2009.

[73] S. Yoshihama, T. Yoshizawa1, Y. Watanabe, M. Kudoh, and K. Oyanagi. Dynamic information flow control architecture for web applications. In *Proc. Europ. Symp. Research in Computer Security*, pages 267–282, 2007.

[74] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. Symp. Operating Systems Design and Implementation*, pages 263–278, 2006.

[75] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: protecting sensitive data leaks using application-level taint tracking. *Operating Systems Review*, pages 142–154, 2011.

[76] J. Zimmermann, L. Mé, and C. Bidan. An improved reference flow control model for policy-based intrusion detection. In *Proc. Europ. Symp. Research in Computer Security*, pages 291–308, 2003.