Runtime Enforcement of Noninterference by Duplicating Processes and their Memories

Maximiliano Cristiá[⋆] and Pablo Mata

Flowgate Security Consulting
Rosario, Argentina
{mcristia, pmata}@flowgate.net

Abstract. This paper presents a formal model for enforcing noninterference running one process of a program per level in the security lattice. The I/O effects of these processes are isolated from one another by restricting each processes to write only to output channels at the same or higher levels. This approach is intended to be implemented in general purpose operating systems. It is therefore more compatible with existing code. A Linux implementation is briefly described.

1 Pointers and Noninterference

In [1] the authors survey the past three decades of information flow security, focusing on work that uses language-based techniques for the specification and enforcement of security policies for data confidentiality. They describe an approach based on type systems for information flow. More precisely, in a security-typed language, the types of program variables and expressions are augmented with annotations that specify policies on the use of typed data. Then, these policies are enforced by compile-time type checking, adding basically no run-time overhead.

When we tried to develop a model based on those techniques for a general purpose operating system and existing applications, we found a problem when we faced programs like the one listed in Figure 1. The program would be used to read a secret value with the intention of disclosing it. To understand the program consider the following: `sec_malloc()` allocates memory in a secure way –for instance it writes zeroes on the memory–; `a` and `b` are pointers like in the C programming language [2]. The problem we found in applying traditional techniques to programs using low level pointers is that either (a) programs are rejected as insecure, or (b) they are accepted but are of reduced utility. Static techniques need to know in advance the security level from where `read()` reads and the

[⋆] Mr. Cristiá is also partially supported by CIFASIS – French Argentine International Center for Information Systems and Sciences – Rosario, Argentina

```
a := sec_malloc(5);
b : = a;
read(x);
for i := 0 to x - 1 do
  *(a + i) := 1;
endfor
for j := 1 to 5 do
  print(*b);
  b := b + 1;
endfor
```

Figure 1: `x` stores a secret value between 1 and 5; the program leaks it.

security level where `print()` prints. Then, if we assume that the latter is low (L) and the former is high (H), compile-time mechanisms will reject the program (i.e. they will not compile it) either because there is a flow of information from `x` into the cells pointed to by `a` –which, by the way, are the same as the ones pointed to by `b`–, or because they cannot deal with C-like pointers at all. Under the same assumptions, dynamic techniques must label `a` and `b`, and not the cells pointed to by them, with H, because otherwise there would be a leakage, making it impossible to output cells containing low level data using the same pointers. It is worth to mention that accurate labeling of memory cells arises only at the presence of pointers, and not with regular variables. The drawbacks we found in applying language-based techniques are along the same lines as the ones noted in [3] when it is necessary to enforce complex dynamic security policies.

The goal of this paper is to present a model (section 3) that warranties classic noninterference [4] of terminating programs written in a high level programming language (section 2) including pointers, I/O, interactions with the underlying operating system (OS) and with a standard semantics, that overcome the problems mentioned in this section. As shown in section 3, noninterference is enforced by an operating system providing a pure runtime mechanism which, at the same time, does not suffer of level creep. Section 4 briefly introduces the implementation of the model in the Linux kernel as part of the Flowx project. The proof that the whole method enforces noninterference is included in the Appendix.

2 The Programming Language and the Operating System

The language considered in this paper (Figure 2) has pointers, I/O interactions and system calls, besides the standard elements and structures

$Expr ::= \mathbb{N} \mid var \mid *var \mid \&var \mid Expr \boxplus Expr$

$BasicSentence ::=$

$\quad \mathsf{skip} \mid var := Expr \mid *var := Expr \mid \mathsf{syscall}(\mathbb{N}, arg_1, \ldots, arg_n)$

$ConditionalSentence ::=$

$\quad \mathsf{if}\ Expr\ \mathsf{then}\ Program\ \mathsf{fi} \mid \mathsf{while}\ Expr\ \mathsf{do}\ Program\ \mathsf{done}$

$BasicAndConditional ::= BasicSentence \mid ConditionalSentence$

$Program ::= BasicAndConditional \mid Program\ ;\ Program$

Figure 2: The grammar of our programming language.

of a C-like programming language. For simplicity we will consider that variables store just natural numbers and the address of any variable is also a natural number. If x is a variable, then $\&x$ is its address, and $*x$ is the content of the variable whose address is the contents of x. In this way any variable can be both a plain variable or a pointer. Expressions can, thus, be formed by variables, the universal binary operator $\boxplus$ and the & and $*$ operators applied to a variable.

The syscall instruction is meant to call OS services of any kind. Its integer argument represents each of these services. The other arguments vary from service to service. When a program executes a syscall instruction the OS process takes control of the computer hardware and the program waits to be resumed. Only one process is active at any given time. In this paper we deal only with two OS general services: $\mathsf{syscall}(0, dev, var)$, called $\mathsf{read}(dev, var)$, which reads val from input device dev; and $\mathsf{syscall}(1, dev, expr)$, called $\mathsf{write}(dev, expr)$, which writes $expr$ to output device dev.

2.1 Language Semantics

The semantics of our language (Figure 3) is defined by stating how the program state changes for every sentence of the language. Formally, the language semantics, LS, is a function defined as follows:

$$LS : Memory \to Program \to Memory$$

where $Memory$ is a function from VAR –the set of all variables– to $\mathbb{N}$ representing the state of the process' memory.

Since our language includes pointers we assume that there exists a bijective function, named $addr$, which returns the variable stored in a given address. Besides, we assume that this function is fixed for all executions

$$LS(M, \mathsf{skip}) = M \tag{LS-skip}$$
$$LS(M, x := e) = M \oplus \{x \mapsto eval(M, e)\} \tag{LS- := }$$
$$LS(M, *x := e) = M \oplus \{\overrightarrow{x, M} \mapsto eval(M, e)\} \tag{LS-$*$}$$
$$LS(M, \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{fi}) = \textbf{if}\ eval(M, e)\ \textbf{then}\ LS(M, P)\ \textbf{else}\ M \tag{LS-if }$$
$$\begin{aligned} &LS(M, \mathsf{while}\ e\ \mathsf{do}\ P\ \mathsf{done}) = \\ &\qquad \textbf{if}\ eval(M, e)\ \textbf{then}\ LS(M, P\ ;\ \mathsf{while}\ e\ \mathsf{do}\ P\ \mathsf{done})\ \textbf{else}\ M \end{aligned} \tag{LS-while }$$
$$LS(M, P_1\ ;\ P_2) = LS(LS(M, P_1), P_2) \tag{LS- ; }$$

Figure 3: Language semantics. $M \in Memory$, $x \in VAR$, $e \in Expr$.

$$eval(M, n) = n \tag{eval-$\mathbb{N}$}$$
$$eval(M, x) = M(x) \tag{eval-VAR}$$
$$eval(M, *x) = M(\overrightarrow{x, M}) \tag{eval-$*$}$$
$$eval(M, \&x) = addr^{-1}(x) \tag{eval-\&}$$
$$eval(M, e_1 \boxplus e_2) = eval(M, e_1) \boxplus eval(M, e_2) \tag{eval-$\boxplus$}$$

Figure 4: Expression evaluation. $M \in Memory$, $n \in \mathbb{N}$, $x \in VAR$, $e_1, e_2 \in Expr$.

of the same program. If x is a program variable and $M \in Memory$ let $\overrightarrow{x, M}$ be equal to $(addr \circ M)(x)$.

Expressions are evaluated according to the definition of a function called *eval* which takes an element of *Memory* and an expression and returns a natural number, as shown in Figure 4.

2.2 Terminating Programs

We can prove that our security model verifies classical noninterference only if it is assumed that all loops within any program will end in a finite number of iterations. Hence, we first define what we call *well-founded* while, and then we assume that all loops within any program are well-founded while's.

Definition 1 (well-founded while). *Let* while e do P done *be a loop in a program where* $e \in Expr$ *and* $P \in Program$. *Let* $M_k, M_{k+1} \in Memory$ *be such that* $LS(M_k, P) = M_{k+1}$ *for* $k \in \mathbb{N}$. *The loop is said to be a well-founded* while *if and only if there exists a partial function* $wf : Memory \nrightarrow \mathbb{N}$ *such that* $wf(M_k) \in \mathbb{N}$, $wf(M_{k+1}) < wf(M_k)$ *and* $eval(M_k, e) \iff wf(M_k) > 0$ *for all* $k \in \mathbb{N}$.

Assumption 1 (Well Defined Programs). *Any loop within any program submitted for execution is a well-founded* while.

3 Dynamic Enforcement of Noninterference

Information flow will be controlled by a state machine called *security machine* or SM. We assume that SM stands between processes and hardware having the ability to decide whether to let processes interact with the environment or not at each particular sentence. For all practical purposes SM can be thought as part of the operating system, more on this in section 4. Formally, SM is a function defined as follows:

$$\begin{aligned} SM :& SState \times Env \times Memory \times Program \\ & \to SState \times Env \times Memory \end{aligned}$$

where $SState$ and Env are defined by:

$$DEV \triangleq il \mid ih \mid ol \mid oh \quad Env \triangleq DEV \to \text{ seq } \mathbb{N} \quad LEVEL \triangleq L \mid H$$
$$SState \triangleq [m : Memory, dl : DEV \to LEVEL]$$

DEV is the set of I/O device names. For simplicity, we assume that there are only two input devices (il and ih) and two output devices (ol and oh). The intention is that through il (ih) users will input low (high) level data, and through ol (oh) they will see low (high) level data. State variable dl represents the security level at which each I/O device is currently working. In turn, Env represents the I/O devices connected to the OS. The sequence of natural numbers in the input devices have not yet been processed and the sequences in the output devices have already been sent to the environment.

SM's transitions are listed in figures 5 and 6. In general, the rules say that a program is run twice or within two contexts: one, represented by $S.m$, is meant to store low level values, and the other, represented by M, is meant to hold high level values. Note that the evaluation of conditions and expressions is performed in both contexts. The only piece of information shared between both contexts is the low level information, as shown in rule SM-read(il). Rules SM-if and SM-while are quite complicated because it is possible that the program might make different decisions when expressions are evaluated over $S.m$ or M. The key rule of the model is SM-write(ol) because it says that output sent to a low output device comes solely from the low memory, $S.m$. Further, if a program has a sentence of the form if $a = 0$ then write($ol, 1$) fi, then it will be evaluated and executed twice: over M and over $S.m$. The important point here is that the output seen at the low level output device will be the same regardless of the value of a. If a equals 0 or not with respect to M, then the inner sentence will be executed anyway, as shown by the first two cases of the SM-if rule. If the value of a came from a high level input device, then this behavior is consistent with noninterference. Last but not least, if the value of a came from a low level input device, then $S.m(a) = M(a)$ –by rule SM-read(il)– making the output seen at the low level output device consistent with the view of a low level user.

3.1 The Security Condition

Theorem 2 in the Appendix states the noninterference property verified by SM and LS. The property is interesting if there are input and output devices working at both L and H; then we define IO to be the function $\{il \mapsto L, ol \mapsto L, ih \mapsto H, oh \mapsto H\}$.

3.2 Analysing the Program of Figure 1

The program listed in Figure 1 is secure and fully usable if run on a system implementing SM, although it would be rejected by language-based techniques because they would consider that there is a potential leak on `x`. Let's assume that `read()` reads from a H device and `prints()` on a L device. When `x` is read, its value is copied only in M; $S.m$ stores garbage in the same memory cell. Since the output device is at L, only values stored in $S.m$ influence the output (by rule SM-write(ol)): then noninterference is warranted because the low level output depends only on low level inputs. Now say both devices work at L. Then `x` is not a secret and consequently it can be disclosed. If both devices work at H, then output is influenced only by M and so the high level user reads the real secret value of `x`. Finally, if the input device works at L and the output device at H, both memories shares the value in `x` implying that the high user can read it.

4 Implementing the Model in the Linux Kernel

The goal of this section is to show that there exist practical implementations of SM. We wanted to implement SM over Linux preserving a reasonable level of usability and performance, and full compatibility with existing software. We briefly and broadly describe the current implementation of Flowx which uses the Linux Security Modules (LSM) technology

$$SM : SState \times Env \times Memory \times Program \to SState \times Env \times Memory$$

$$SM(S, E, M, \mathsf{skip}) = (S, E, M) \qquad \text{(SM-}\mathsf{skip}\text{)}$$

$$\begin{aligned} &SM(S, E, M, x := expr) = \\ &\quad ([m \leftarrow LS(S.m, x := expr), dl \leftarrow S.dl], E, LS(M, x := expr)) \end{aligned} \qquad \text{(SM- :=)}$$

$$\begin{aligned} &SM(S, E, M, *x := expr) = \\ &\quad ([m \leftarrow LS(S.m, *x := expr), dl \leftarrow S.dl], E, LS(M, *x := expr)) \end{aligned} \qquad \text{(SM-}*\text{)}$$

$$\begin{aligned} &SM(S, E, M, \mathsf{read}(il, x)) = \\ &\quad ([m \leftarrow S.m \oplus \{x \mapsto head \circ E(il)\}, dl \leftarrow S.dl], \\ &\quad E \oplus \{il \mapsto tail \circ E(il)\}, M \oplus \{x \mapsto head \circ E(il)\}) \end{aligned} \qquad \text{(SM-}\mathsf{read}(il)\text{)}$$

$$\begin{aligned} &SM(S, E, M, \mathsf{read}(ih, x)) = \\ &\quad (S, E \oplus \{ih \mapsto tail \circ E(ih)\}, M \oplus \{x \mapsto head \circ E(ih)\}) \end{aligned} \qquad \text{(SM-}\mathsf{read}(ih)\text{)}$$

$$\begin{aligned} &SM(S, E, M, \mathsf{write}(ol, e)) = \\ &\quad (S, E \oplus \{ol \mapsto \langle eval(S.m, e)\rangle \natural E(ol)\}, M) \end{aligned} \qquad \text{(SM-}\mathsf{write}(ol)\text{)}$$

$$\begin{aligned} &SM(S, E, M, \mathsf{write}(oh, e)) = \\ &\quad (S, E \oplus \{oh \mapsto \langle eval(M, e)\rangle \natural E(oh)\}, M) \end{aligned} \qquad \text{(SM-}\mathsf{write}(oh)\text{)}$$

$$\begin{aligned} &SM(S, E, M, \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{fi}) = \\ &\quad \begin{cases} SM(S, E, M, P) & \textbf{if } eval(S.m, e) \wedge eval(M, e) \\ (SM(S, E, M, P).1, \\ \quad SM(S, E, M, P).2, M) & \textbf{if } eval(S.m, e) \wedge \neg eval(M, e) \\ (S, E', SM(S, E, M, P).3) & \textbf{if } \neg eval(S.m, e) \wedge eval(M, e) \\ (S, E, M) & \textbf{if } \neg eval(S.m, e) \wedge \neg eval(M, e) \end{cases} \\ &\quad \text{where } E' = E \oplus \{ih \mapsto SM(S, E, M, P).2(ih), \\ &\qquad\qquad\qquad oh \mapsto SM(S, E, M, P).2(oh)\} \end{aligned} \qquad \text{(SM-}\mathsf{if}\text{)}$$

$$SM(S, E, M, P_1 \ ; \ P_2) = SM(SM(S, E, M, P_1), P_2) \qquad \text{(SM- ;)}$$

Figure 5: Security Machine semantics (part 1). Symbol $\natural$ means sequence concatenation.

Let $\mathcal{P}While$ be P ; $\mathsf{while}\ e\ \mathsf{do}\ P\ \mathsf{done}$

$$SM(S,E,M,\mathsf{while}\ e\ \mathsf{do}\ P\ \mathsf{done}) =$$

$$\begin{cases} SM(S,E,M,\mathcal{P}While) & \textbf{if } eval(S.m,e) \wedge eval(M,e) \\ (SM(S,E,M,\mathcal{P}While).1, \\ \quad SM(S,E,M,\mathcal{P}While).2, M) & \textbf{if } eval(S.m,e) \wedge \neg eval(M,e) \\ (S,E',SM(S,E,M,\mathcal{P}While).3) & \textbf{if } \neg eval(S.m,e) \wedge eval(M,e) \\ (S,E,M) & \textbf{if } \neg eval(S.m,e) \wedge \neg eval(M,e) \end{cases} \qquad \text{(SM-while)}$$

$$\text{where } E' = E \oplus \{ih \mapsto SM(S,E,M,\mathcal{P}While).2(ih), \\ oh \mapsto SM(S,E,M,\mathcal{P}While).2(oh)\}$$

Figure 6: Security Machine semantics (part 2).

and some minor modifications to the kernel itself in order to implement the model described earlier.

The broad idea of the implementation is to simulate the existence of as many computer as security levels are used in the system. Each of these "virtual" computers executes process up to a given security level. However, all of them share the inputs up to a given security level. For instance, if computer A is executing at level L_A and computer B is executing at level L_B, with $L_A \succeq L_B$, then A and B will share all the input data classified at level L_B or less. All "virtual" computers are connected to the same set of I/O devices, but the OS decides which inputs are sent to which computers. On the other hand, each "virtual" computer is allowed to write information on any output device classified at its level or higher; for instance A (B) will be allowed to write information on any device with a security level greater than or equal to L_A (L_B).

Flowx simulates these "virtual" computers by executing processes as follows. Initially, each process is classified at L and it is spawned into two processes once it wants to access information at H. This means that if a process access information only at L then it will behave as a regular process. When the process is divided its memory is duplicated for the two new processes, so each of them will have its own, disjoint memory space ($S.m$ and M). The spawned process must start execution from the exact same point where its creator violated the security policy. Flowx, thus, moves the program counter of the new process so it starts at that point. Once the new process is created, both will run independently although they will agree on low values since they will be connected with the same I/O devices. Flowx implements this feature by buffering all the input received on an input device and then delivering it to the copies of the processes using this device that are working at a security level greater than or equal to the security level of the input device; the buffer is emptied once all copies have requested the input. If an initial single process will read input from an input device working at, say, H, when it actually requests data from that device, the OS spawns it and the (real) input is delivered only to the copy classified at H while the copy classified at L receives a sequence of constant values. Obviously, this will make both processes to follow a different path along the program's code but this is precisely the whole point: if the program is a Trojan horse being used by an attacker then he will see the same output at his low level terminal regardless of the values entered by high level users; but if the program is legitimate and the user is a high level official working from his high level terminal, then he will receive the expected output –from the high level copy of the process. There is yet another situation that deserves to be analyzed. A high level user working from his high level terminal wants to process and see low level data. In this case Flowx will not duplicate the process, since it does not access high level data, and thus the low level process will be allowed to write on the high level terminal.

It is very important to remark that Flowx achieves this behavior without changing a single system call signature. In this way, Flowx preserves compatibility with existing application software and it is independent of the number of security levels that processes will use during its lifetime. However, it also shows an important performance penalty both in execution speed and in memory consumption. We think that this problem can be almost eliminated *in practice* by adding more processors or cores and memory or by restricting the number of security levels with which a process can simultaneously work during its lifetime. Say a server running Linux executes over processor $\mathcal{P}$ and has $\mathcal{N}$ bytes of memory. Now say that the user wants to divide its information into $\mathcal{M}$ securiy levels. Then, to run Flowx with a similar efficiency, the user needs a server with $\mathcal{M}$ processors like $\mathcal{P}$ and $\mathcal{M} \times \mathcal{N}$ bytes of memory because, in the worst case, Flowx needs to run $\mathcal{M}$ processes for each process that would be run on the Linux box. We think that the tension between hardware cost and better security will be resolved in favor of the last for many organizations and user communities.

The key remaining question, though, is whether such a system will be acceptable usable for an end user or not in terms of how difficult is to use it, but this is the topic of future work with Flowx.

5 Discussion and Related Work

We developed the ideas presented in this paper when we tried to implement a language-based model in a UNIX-like operating system. Besides the problem with pointers that we highlighted in section 1, moving from the language-based perspective to a runtime mechanism, gave us some tips about the first approach. Static methods require programmers to understand the very difficult problem of noninterference because secure compilers will prompt an error every time there is an illegal flow within a program. Hence, programmers will need to understand these errors and this imply that they will need to understand the whole problem of MLS –besides the type system, its deductive rules and so on. Programmers programming general purpose applications would find very hard to understand all these issues. Our approach free them to understand noninterference because it will never rise an error saying something like "illegal information flow". Another insight we gained was that language-based methods tend to reject programs due to illegal information flows, although these flows depend on where the information comes from and where it is going to, which is not always addressed by these methods. The method presented in this papers do avoids those problems because the enforcement mechanism is inside the OS and not inside the programming language.

Since non-termination may leak more than just one bit [5], we have to face the problem of proving that the model is noninterfering when non-terminating programs are also considered. However,t we clearly see that the implementation is secure in this respect because processes at one level cannot see the processes at higher levels at all –except when the physical computer cannot run more processes.

5.1 Comparison with Similar Approaches

Most papers about noninterference approach the problem from the language based perspective as was surveyed in [1]. However, some work can be found proposing runtime enforcement. In [6, 7] the authors propose a monitoring system controlling the flow of information within a process and nullifying dangerous flows. In [8] Cavadini propose a method that slices a possible dangerous program into secure programs; he combines the static and the runtime monitoring approaches, although he still analyzes the program text. Also Shroff and others [9] propose another runtime monitoring system. However, as far as we searched there is no method dealing with programs including C-like pointers, what makes it possible to run just one version of the program.

Although the static, language-based approach is quite different from ours, the analysis of some works along these lines was important for us because they show that similar results can be drawn. Hunt and Sands in [10] develop a family of flow-sensitive type systems. A type system is flow-sensitive if it allows program variables to change their initial access class. They propose a program transformation by adding a set of variables per each sensitivity level, and then adding an equal number of instructions. In a sense, this transformation is the static counterpart of our proposal. Hedin and Sands in [11] treat non-opaque pointers by defining another flow-sensitive type system which suggests a similar program transformation. Note that this transformation leads to similar performance penalties than our technique but it reject programs and cannot deal with a dynamic number of security levels. In [12] the authors, like us, do not assign types –or security labels– to program variables since they observe that, for instance, a program with no high inputs is secure no matter what information flows occur. Their language includes I/O like in this article. Banerjee and Naumann in [13] proves classical noninterference for a Java-like language including pointers and many other features of object oriented programming languages. [14] extends the previous work by studying the problem of noninterference in the presence of stack inspection. Two things of that paper are worth noticing with respect to our work: (a) authors recognize that specifying static analyzes for confidentiality have not seen much use, and (b) that they parametrize classes over security levels, which is a way to avoid some limitations of the typed-based techniques. In [15] a proof for a noninterference property of a λ-calculus including references is given. Zdancewic and Myers [16] work with a low level language and consider to analyze the output of a compiler rather than source code.

There are also approaches combining static and dynamic techniques. In [17] the authors deals with security policies that depend on which principals interact with the system. Also [18] presents a dependent type system to control information flow within programs where security classes of data can vary dynamically.

6 Conclusions and Future Work

We believe that this paper proves that it is possible to impose a notion of noninterference in a general purpose computing system by using a runtime mechanism. The idea born from the realization that if the user can interact with as many computers as security classes he is allowed to work with, and if each of these computers processes information at just one level, then we have noninterference. The problem was, then, to transparently simulate this system in one computer.

The main theoretical problem we need to face is to prove that our model is still secure at the presence of non-terminating programs. We believe that this more of a formal problem since in our implementation non-terminating high level processes cannot be seen by low level users or programs. We will work on using coinductive techniques like simulations and bisimulations. Also we need to augment the language with a goto statement to analyze unstructured programs. This would take us to an assembly-like language whose programs can change their image at runtime. Besides, it is necessary to prove a conditional noninterference property when system calls that allow some users to change the security level of resources are also considered. Regarding the implementation we still need to solve some issues like the adequate use of cryptography, adapting the system to a graphical environment, providing a better interface for security administrators, etc.

Acknowledgments

Gilles Barthe supported this work from its beginning and suggested the idea for the introduction. Without Gilles perhaps we couldn't get so far. Thanks the reviewers for their detailed analysis and comments.

References

1. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1) (2003) 5–19
2. Kernighan, B.W., Ritchie, D.M.: The C Programming Language Second Edition. Prentice-Hall, Inc. (1988)
3. Zdancewic, S.: Challenges for information-flow security. In: In Proc. Programming Language Interference and Dependence (PLID. (2004)
4. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 Berkeley Conference on Computer Security. (1982) 11–22 IEEE Computer Society Press.
5. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security, Berlin, Heidelberg, Springer-Verlag (2008) 333–348
6. Le Guernic, G., Jensen, T.: Monitoring information flow. In: Workshop on Foundations of Computer Security (FCS'05). (2005) 19–30

7. Le Guernic, G.: Automaton-based confidentiality monitoring of concurrent programs. In: CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium, Washington, DC, USA, IEEE Computer Society (2007) 218–232
8. Cavadini, S.: Secure slices of insecure programs. In: ASIACCS '08: Proceedings of the 2008 ACM symposium on Information, computer and communications security, New York, NY, USA, ACM (2008) 112–122
9. Shroff, P., Smith, S., Thober, M.: Dynamic dependency monitoring to secure information flow. In: CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium, Washington, DC, USA, IEEE Computer Society (2007) 203–217
10. Hunt, S., Sands, D.: On flow-sensitive security types. In: Proc. Principles of Programming Languages, 33rd Annual ACM SIGPLAN - SIGACT Symposium (POPL'06), Charleston, South Carolina, USA, ACM Press (2006) 79–90
11. Hedin, D., Sands, D.: Noninterference in the presence of non-opaque pointers. In: CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations, Washington, DC, USA, IEEE Computer Society (2006) 217–229
12. O'Neill, K.R., Clarkson, M.R., Chong, S.: Information-flow security for interactive programs. In: 19th IEEE Workshop on Computer Security Foundations, Washington, DC, USA, IEEE Computer Society (2006) 190–201
13. Banerjee, A., Naumann, D.A.: Secure information flow and pointer confinement in a java-like language. In: CSFW '02: Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02), Washington, DC, USA, IEEE Computer Society (2002) 253
14. Sun, Q., Naumann, D.A., Banerjee, A.: Modular and constraint-based information flow inference for an object-oriented language. In: 11th International Static Analysis Symposium. (2004)
15. Pottier, F., Simonet, V.: Information flow inference for ml. ACM Trans. Program. Lang. Syst. **25**(1) (2003) 117–158
16. Zdancewic, S., Myers, A.C.: Secure information flow via linear continuations. Higher Order Symbol. Comput. **15**(2-3) (2002) 209–234
17. Tse, S., Zdancewic, S.: Run-time principals in information-flow type systems. ACM Trans. Program. Lang. Syst. **30**(1) (2007) 6
18. Zheng, L., Myers, A.: Dynamic security labels and noninterference. In: FAST'04: Workshop on Formal Aspects in Security and Trust, Boston, MA, USA, Springer (2004) 27–40

Appendix: Proofs

Theorem 1 (Invariants).

$$\forall S \in SState; E_1, E_2 \in Env; M_1, M_2 \in Memory; P \in Program \bullet$$

$$P \text{ verifies Assumption 1} \tag{H1}$$

$$S.dl = IO \tag{H2}$$

$$E_1(il) = E_2(il) \tag{H3}$$

$$SM(S, E_1, M_1, P) = (S'_1, E'_1, M'_1) \tag{H4}$$

$$SM(S, E_2, M_2, P) = (S'_2, E'_2, M'_2) \tag{H5}$$

$$\implies E'_1(il) = E'_2(il) \wedge S'_1.m = S'_2.m$$

Proof. The proof is by induction on the program structure. Assume $x \in \mathit{VAR}$ and $e \in \mathit{Expr}$. The base cases skip, $\mathsf{read}(ih, x)$, $\mathsf{write}(ol, e)$ and $\mathsf{write}(oh, e)$ follow directly from H4-H6 and the corresponding SM rule.

$x := e$] $E_1'(il) = E_2'(il)$ from SM- $:=$ and H3. LS- $:=$ and SM- $:=$ imply $S_1'.m = LS(S.m, x := e)$ and $S_2'.m = LS(S.m, x := e)$, then $S_1'.m = S_2'.m$.

$*x := e$] This case is proved as the previous one.

$\mathsf{read}(il, x)$] Follows immediately from SM-$\mathsf{read}(il)$ and H3.

For the inductive cases, we assume that $P, P_1 \in \mathit{Program}$ are programs verifying the theorem; we call these HI and HI1, respectively .

if] We proceed by cases according to SM-if .

$\mathit{eval}(S.m, e) \wedge \mathit{eval}(M_1, e) \wedge \mathit{eval}(M_2, e)$. Then from SM-$\mathsf{if}$

$SM(S, E_1, M_1, \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{fi}) = SM(S, E_1, M_1, P)$

$SM(S, E_2, M_2, \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{fi}) = SM(S, E_2, M_2, P)$

Hence this case is proved by HI.

Cases $\mathit{eval}(S.m, e) \wedge \mathit{eval}(M_1, e) \wedge \neg \mathit{eval}(M_2, e)$ and $\mathit{eval}(S.m, e) \wedge \neg \mathit{eval}(M_1, e) \wedge \neg \mathit{eval}(M_2, e)$ in a similar way.

$\neg \mathit{eval}(S.m, e) \wedge \mathit{eval}(M_1, e) \wedge \mathit{eval}(M_2, e)$. Then from SM-$\mathsf{if}$

$SM(S, E_1, M_1, \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{fi}) = (S_1, \tilde{E_1}, SM(S_1, E_1, M_1, P).3)$

$SM(S, E_2, M_2, \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{fi}) = (S, \tilde{E_2}, SM(S, E_2, M_2, P).2)$

where $\tilde{E_1}(il) = E_1(il)$ and $\tilde{E_2}(il) = E_2(il)$ from SM-if

Hence this case is proved by H3.

Case $\neg \mathit{eval}(S.m, e) \wedge \mathit{eval}(M_1, e) \wedge \neg \mathit{eval}(M_2, e)$ is proved in a similar way. While $\neg \mathit{eval}(S.m, e) \wedge \neg \mathit{eval}(M_1, e) \wedge \neg \mathit{eval}(M_2, e)$ is trivial from SM-if .

while] This is similar to the previous one but it is necessary to apply the induction principle over the number of iterations of the loop, in each case. This is possible due to H1.

$\mathsf{composition}$] From SM- ; we have

$$\begin{aligned} &SM(S, E_1, M_1, P\ ;\ P_1) \\ &\quad = SM(SM(S, E_1, M_1, P), LS(M_1, P), P_1) = SM(S_1', E_1', M_1', P_1) \\ &SM(S, E_2, M_2, P\ ;\ P_1) \\ &\quad = SM(SM(S, E_2, M_2, P), LS(M_2, P), P_1) = SM(S_2', E_2', M_2', P_1) \end{aligned}$$

Since $S_1'.m = S_2'.m$ and $E_1'(il) = E_2'(il)$ from HI, then we can apply HI1 to $SM(S_1', E_1', M_1', P_1)$ and $SM(S_2', E_2', M_2', P_1)$, and from here the theorem.

Theorem 2 (Noninterference).

$$\forall S \in SState; E_1, E_2 \in Env; M_1, M_2 \in Memory; P \in Program \bullet$$
$$P \textit{ verifies Assumption 1} \quad \text{(H1)}$$
$$S.dl = IO \quad \text{(H2)}$$
$$E_1(il) = E_2(il) \quad \text{(H3)}$$
$$E_1(ol) = E_2(ol) \quad \text{(H4)}$$
$$SM(S, E_1, M_1, P) = (S'_1, E'_1, M'_1) \quad \text{(H5)}$$
$$SM(S, E_2, M_2, P) = (S'_2, E'_2, M'_2) \quad \text{(H6)}$$
$$\implies E'_1(ol) = E'_2(ol)$$

Proof. The proof is by induction on the program structure. Assume $x \in VAR$ and $e \in Expr$. The base cases skip, $x := e$, $*x := e$, $\mathsf{read}(il, x)$, $\mathsf{read}(ih, x)$, $\mathsf{write}(ol, e)$ and $\mathsf{write}(oh, e)$ follow directly from H4-H6 and the corresponding SM rule.

For the inductive cases, we assume that $P, P_1 \in Program$ are programs verifying the theorem; we call these HI and HI1, respectively .

if] We proceed by cases according to SM-if .

$eval(S.m, e) \wedge eval(M_1, e) \wedge eval(M_2, e)$. Then from SM-if

$$SM(S, E_1, M_1, \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{fi}) = SM(S, E_1, M_1, P)$$
$$SM(S, E_2, M_2, \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{fi}) = SM(S, E_2, M_2, P)$$

Hence this case is proved by HI.

Cases $eval(S.m, e) \wedge eval(M_1, e) \wedge \neg eval(M_2, e)$ and $eval(S.m, e) \wedge \neg eval(M_1, e) \wedge \neg eval(M_2, e)$ are proved in a similar way.

$\neg eval(S.m, e) \wedge eval(M_1, e) \wedge eval(M_2, e)$. Then from SM-if

$$SM(S, E_1, M_1, \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{fi}) = (S, \tilde{E_1}, SM(S, E_1, M_1, P).3)$$
$$SM(S, E_2, M_2, \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{fi}) = (S, \tilde{E_2}, SM(S, E_2, M_2, P).2)$$

where $\tilde{E_1}(ol) = E_1(ol)$ and $\tilde{E_2}(ol) = E_2(ol)$ from SM-if

Hence this case is proved by H4.

Case $\neg eval(S.m, e) \wedge eval(M_1, e) \wedge \neg eval(M_2, e)$ is proved in a similar way. While $\neg eval(S.m, e) \wedge \neg eval(M_1, e) \wedge \neg eval(M_2, e)$ is trivial from SM-if .

while] This is similar to the previous one but it is necessary to apply the induction principle over the number of iterations of the loop, for each case. This is possible due to H1.

composition] From SM- ; we have

$$SM(S, E_1, M_1, P\ ;\ P_1)$$
$$= SM(SM(S, E_1, M_1, P), LS(M_1, P), P_1) = SM(S'_1, E'_1, M'_1, P_1)$$
$$SM(S, E_2, M_2, P\ ;\ P_1)$$
$$= SM(SM(S, E_2, M_2, P), LS(M_2, P), P_1) = SM(S'_2, E'_2, M'_2, P_1)$$

Since $S'_1.m = S'_2.m$ and $E'_1(il) = E'_2(il)$ from Theorem 1 and $E'_1(ol) = E'_2(ol)$ from HI, then we can apply HI1 to $SM(S'_1, E'_1, M'_1, P_1)$ and $SM(S'_2, E'_2, M'_2, P_1)$, and from here the theorem.