On Secure Information Flow in Reactive Programming Paradigm (extended abstract)

Zhengqin Luo
INRIA Sophia-Antipolis
Zhengqin.Luo@sophia.inria.fr

I. Introduction

The language-based information flow security is a well established theory [Sabelfeld and Myers(2003)]. Usually inputs and outputs of programs are classified as either *public (low)* or *private (high)*. The aim is to ensure that programs are *non-interferent*, which means any public inputs will produce exactly the same public outputs, regardless of what private inputs are. There have been various static and dynamic analysis techniques to ensure this non-interference property for different sequential languages.

However, the results are not very satisfactory for multi-threaded language setting. Consider following three threads in a ML-like while-language, where r and s are ff initially, h is a private input and l is a public output:

$$t1 : \mathsf{if}\ !h\ \mathsf{then}\ r := tt\ \mathsf{else}\ s := tt$$
$$t2 : (\mathsf{while}\ \neg !r\ \mathsf{do}\ l := ff); s := tt$$
$$t3 : (\mathsf{while}\ \neg !s\ \mathsf{do}\ l := tt); r := tt$$

Usual type systems for non-interference will take each thread as typable program, but the value stored in h is assigned to l by standard interleaving semantics, violating the security property. Some type systems for multi-threaded programs are proposed [Smith(2001)], [Boudol and Castellani(2002)], [Sabelfeld(2001)], but they are too conservative by rejecting following programs:

$$p1 : (\mathsf{if}\ !h\ \mathsf{then}\ ()\ \mathsf{else}\ ()); l := 1$$
$$p2 : (\mathsf{while}\ !h\ \mathsf{do}\ ()); l := 1$$

Those type systems forbid any testing of confidential variables followed by assignment of public variables, since implicit synchronizations between threads (by testing shared variables) are usually difficult to track.

The reactive programming paradigm (a.k.a. synchronous programming [Boussinot and de Simone(1996)]), on the other hand, provides another solution for multi-threaded programming. Threads are synchronized by explicitly using *broadcasting signals*, *suspension* construct, and *preemption* construct. Unsimilar to the standard interleaving semantics, each thread will not yield the scheduler until it suspends or terminates.

In this paper, we investigate how information is flowed among threads in reactive programs. We show that there are new ways of leaking information in this programming paradigm. We also show how to define run-time security errors in this particularly setting, to prevent insecure information flow. We implement the semantics as a monitoring abstract machine, which dynamically tracks information flow during execution. We propose a fine-grained termination-insensitive definition of non-interference, particularly for this reactive setting. We also present a type and effect system, which accepts more programs then usual type systems for multi-threaded programs, to guarantee a safety property (no run-time error occurs).

II. Reactive programming paradigm and information flow

We consider here a core language which is a subset of ULM language [Boudol(2004)]. Our language extends the imperative ML-like language with reactive primitive for broadcasting signal (emit), suspension (when) and preemption (watch), together with a cooperative scheduling policy over threads.

A reactive machine $\mathcal{M} = [\mu, \xi, t, T]$ contains a running thread t and a queue of waiting threads denoted by T. All these threads are ordered by the creation time, and they share the same store μ and signal environment ξ. μ maps variables to values, and ξ is the set of currently present signals. Informally, (emit s) adds signal s to ξ. (when s do M) first tests whether signal s is present; if s is present, then M is evaluated; otherwise the thread will suspend itself. (watch s do M) gives up the evaluation of M when s is present at a special moment (explained later).

This reactive machine runs as follows: all threads are cooperatively scheduled, which means the currently running thread t will keep running until it suspends or terminates. When scheduling is needed, the immediate successor (by the order of creating time) of current thread is scheduled to run. Computations are divided by *instants*, in which threads execute until all of them are suspended or terminated. At the end of each instant, preemptions (watch s do M) will take place, and then the signal environment ξ is reset to empty. At the beginning of a new instant, the oldest thread is set to be the currently running thread.

Since threads are synchronized by signals, then emiting and testing signals may leak some information. For instance, the following three threads will again assign h to l.

$$r1 : \mathsf{if}\ !h\ \mathsf{then}\ (\mathsf{emit}\ r)\ \mathsf{else}\ (\mathsf{emit}\ s)$$
$$r2 : (\mathsf{when}\ r\ \mathsf{do}\ l := ff); \mathsf{emit}\ s$$
$$r3 : (\mathsf{when}\ s\ \mathsf{do}\ l := tt); \mathsf{emit}\ r$$

We classify signals as either public or private, because it can convey information by being emitted and tested. Similar to the case of variables, we can consider emitting a signal as storing something in a variable, testing a signal

as reading from a variable.

III. A monitoring semantics

It has been argued in [Boudol(2008)] that, from a programmer's point view, insecure information flow should be seen as a *programming error*. A security-minded semantics is given to a sequential language under consideration, marking any attempt to assign values that elaborated using confidential information to public locations as *run-time error*. In this section we extend the monitoring semantics to the reactive programming paradigm. This semantics dynamically tracks information flow during the execution of the reactive machine.

We assume a security lattice $(\mathcal{L}, \preceq)$ for *confidentiality levels*. Variables and signals are labeled with these levels. $L \preceq H$ means that x_L is less confidential then y_H.

The operational semantics is defined by means of an abstract machine for each independent thread. It augments the standard machines by separately building a *reading level* and a *reactive level* during the execution. The reading level records the confidential level of the information obtained by reading variables. The reactive level records the level of information obtained by testing signals through reactive construct (suspension, preemption). A *running thread* is organized as $t = (\mathsf{pc}, \mathsf{cur}, S, M)$, where pc is reading level, cur is reactive level, S is the stack, and M is the code to evaluate. For space reason, detailed rules of semantics are not given.

When creating a reference (variable), or doing an assignment, or emitting a signal, we check whether the level of target location is not less than the current reading level and reactive level, to prevent one from storing high level information in low level locations:

$$\mathsf{pc} \preceq l \text{ and } \mathsf{cur} \preceq l$$

If the checks are failed, the reactive machine will raise a run-time error.

$$[\mu, \xi, t, T] \rightarrow \mathsf{err}$$

IV. A fine-grained termination-insensitive definition of security

The standard non-interference property does not work quite well with reactive programs, since it only consider terminated programs, while most reactive programs are designed to be running forever. On the other hand, most properties based on bisimulation can deal with non-terminating programs. But they are by nature termination-sensitive or suspension-sensitive, giving the adversary unrealistic power to observe termination or suspension.

We propose another fine-grained termination-insensitive definition of security, where the adversary cannot observe termination or suspension, but it can observe the public part of the memory and the signal environment. Informally, if the adversary can not distinguish two configurations of machine $c_1 = (\mu_1, \xi_1, t_1, T_1)$ and $c_2 = (\mu_2, \xi_2, t_2, T_2)$, then [1] $c_1 \approx_L c_2$ and one of the following holds.

1. $c_1 \rightarrow c_1'$, and $c_2 \rightarrow^* c_2'$, where c_1' is indistinguishable with c_2';
2. $c_1 \rightarrow c_1'$, and for any configuration c_2' that is reachable from c_2: $c_2 \rightarrow^* c_2'$, $c_2 \approx_L c_2'$;
3. Vice versa for c_2.

The intuition for this definition is that if two machines are indistinguishable, then either each step made by one machine can always be matched by the other one, or the other machine can not do anything that is observable by the adversary.

We show that, if a reactive program does not run into security error, then it is secure according to our find-grained definition of security.

V. Type and effect system

We have designed a standard type system which entails the safety property. The design of the type system follows the line of "state-oriented" approach [Boudol(2005)], [Matos and Boudol(2005)]. Furthermore, we take the current reading level pc and reactive level cur into the typing judgment, in order to type the running thread. The judgements of the type and effect system have the form:

$$\mathsf{pc}; \mathsf{cur}; \Gamma \vdash M : e, \tau$$

where Γ is a typing context that maps variable to its type, e is a security effect and τ is a type. We overload $\Gamma \vdash \mu$ to type memory store μ, which means each value in the store is typable.

Our type system is flexible in a way that we do not restrict assignments of low level variables following after testing of high level varibales. We have the type safety result saying that if a program is typable, then no runtime check is required. Informally, we say M is secure when M does not run into security error.

Theorem 1 (Type Safety) If M is typable in typing context Γ for initial levels $(\bot, \bot)$, that is, $\bot, \bot, \Gamma \vdash M : e, \tau$ for some e and τ, and memory μ is typable, that is $\Gamma \vdash \mu$, then M is secure with μ.

VI. Conclusion

In this paper we investigate the information flow problem in a reactive programming setting, which augments the multi-threaded functional language with reactive constructs to achieve cooperative synchronization between threads. We investigate how information can be flowed in this setting, and propose a fine-grained definition of security. Furthermore, we define the security property as a safety property by means of dynamically checking information flow during evaluation. Finally, we show a type system to ensure that typable programs do not need these dynamic checks during the execution.

[1] For ease of presentation, we informally use $\approx_L$ to denote that memories and signal environments coincide on the public part.

References

[Boudol(2004)] Gérard Boudol. ULM: A core programming model for global computing: (extended abstract). In *ESOP*, pages 234–248, 2004.

[Boudol(2005)] Gérard Boudol. On typing information flow. In Dang Van Hung and Martin Wirsing, editors, *ICTAC*, volume 3722 of *Lecture Notes in Computer Science*, pages 366–380. Springer, 2005. ISBN 3-540-29107-5.

[Boudol(2008)] Gérard Boudol. Secure information flow as a safety property. In Pierpaolo Degano, Joshua D. Guttman, and Fabio Martinelli, editors, *Formal Aspects in Security and Trust*, volume 5491 of *Lecture Notes in Computer Science*, pages 20–34. Springer, 2008.

[Boudol and Castellani(2002)] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theor. Comput. Sci.*, 281(1-2):109–130, 2002.

[Boussinot and de Simone(1996)] Frédéric Boussinot and Robert de Simone. The sl synchronous language. *IEEE Trans. Software Eng.*, 22(4):256–266, 1996.

[Matos and Boudol(2005)] Ana Almeida Matos and Gérard Boudol. On declassification and the non-disclosure policy. In *CSFW*, pages 226–240. IEEE Computer Society, 2005. ISBN 0-7695-2340-4.

[Sabelfeld(2001)] Andrei Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Ershov Memorial Conference*, volume 2244 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2001. ISBN 3-540-43075-X.

[Sabelfeld and Myers(2003)] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.

[Smith(2001)] Geoffrey Smith. A new type system for secure information flow. In *CSFW*, pages 115–125. IEEE Computer Society, 2001. ISBN 0-7695-1146-5.