Computer Languages, Systems & Structures 46 (2016) 20–43

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Specification and static enforcement of scheduler-independent noninterference in a middleweight Java

Zeinab Iranmanesh [a], Mehran S. Fallah [a,b,*]

[a] Department of Computer Engineering and Information Technology, Amirkabir University of Technology (Tehran Polytechnic), P.O. Box: 15875-4413, Tehran, Iran
[b] School of Computer Science, Institute for Research in Fundamental Sciences (IPM), P.O. Box: 19395-5746, Tehran, Iran

ARTICLE INFO

Article history:
Received 31 August 2015
Accepted 11 May 2016
Available online 18 May 2016

Keywords:
Covert channels
Multithreaded object-oriented programming
Scheduler-independent noninterference
Security type systems

ABSTRACT

We introduce a new timing covert channel that arises from the interplay between multithreading and object orientation. This example motivates us to explore the root of the problem and to devise a mechanism for preventing such errors. In doing so, we first add multithreading constructs to Middleweight Java, a subset of the Java programming language with a fairly rich set of features. A noninterference property is then presented which basically demands program executions be equivalent in the view of whom observing final public values in environments using the so-called high-independent schedulers. It is scheduler-independent in the sense that no matter which scheduler is employed, the executions of the program satisfying the property do not lead to illegal information flows in the form of explicit, implicit, or timing channels. We also give a provably sound type-based static mechanism to enforce the proposed property.

1. Introduction

Language-based techniques have proven effective in specifying and enforcing information-flow policies. For a given program, the security policy is essentially specified as a mapping from the set of logical information receptacles in that program to a lattice of security classes representing levels of information sensitivity. The intended meaning of such a policy is that no execution of the program should lead to an information flow from more to less sensitive information receptacles. The compile-time and run-time analyses are then so conducted that the policy can be enforced. In particular, many type systems, e.g., [1–5], have been proposed to this end.

Any violation of an information-flow policy involves illegal information flows in the form of explicit, implicit, or timing channels. Illegal explicit flows occur when private (high) variables are assigned to public (low) variables. Implicit flows and timing channels are more demanding and result from insecure control flows [6]. In general, giving a precise definition of illegal flows is an arduous task and highly depends on how powerful the attackers are considered. The capabilities of

* Corresponding author at: Department of Computer Engineering and Information Technology, Amirkabir University of Technology (Tehran Polytechnic), P.O. Box: 15875-4413, Tehran, Iran. Tel.: +98 21 6454 2718.
E-mail address: msfallah@aut.ac.ir (M.S. Fallah).

http://dx.doi.org/10.1016/j.cl.2016.05.003
1477-8424/

attackers should, in truth, be taken into account when specifying the policy. To do so, scholars give semantic notions of secure information flows in the form of noninterference properties. Such a property indeed defines allowable executions and is intended to be satisfied only by those programs not leading to illegal flows.

Devising a noninterference property as such is a challenging problem and has been a main research topic in the area of language-based security. Another problem is to find appropriate mechanisms to enforce a given information-flow policy in programs of a given programming language. These two problems interact. The security policy should prohibit the new attacks resulting from the features added to the language. Enforcing an effective policy in a language with a rich set of features is also difficult. In this paper, we take this area of study one step further and tackle the problem where the interplay between object orientation and multithreading leads to new illegal channels.

Some illegal information channels are peculiar to object-oriented programming. A number of security type systems have been proposed to prevent such channels [2,7–12]. Similarly, multithreading contributes to the appearance of new illegal channels. If the timing behavior of a thread depends on sensitive information, the run of that thread may induce timing channels. The example code below illustrates how such a channel may be created as a result of the concurrent execution of threads `th1` and `th2`. In this code, `h` is a variable containing high values and `l` is a variable that can be observed by low observers. If the initial value of `h` is positive, the execution of the code under a round-robin scheduler is more likely to lead to `l=0`. For a nonpositive `h`, however, `l=1` is more probable than `l=0`. The covert channel given in this example is an internal timing channel as it leaks information through low variables. It differs from external timing channels where the leakage becomes viable through measuring the time taken by the code at run-time.

```
th1: (if h > 0 then sleep(100); else skip;); l=0;
th2: sleep(50); l=1;
```

In this paper, we focus on internal timing channels. There are, in general, three principal classes of methods for eliminating such covert channels. The protection/hiding-based methods are intended to devise schedulers that hide security-critical timing behavior of threads [5,13–16]. The second class is comprised of external-timing-based methods that add dummy computations to a leaky construct [3,17–19]. Low-determinism-based methods constitute another class in which any race on low values is prevented [4,20,21].

As stated above, there have been many attempts to bring information-flow security to the programs that make use of objects or threads. Enforcing security policies in the presence of the two features at the same time, however, has not been studied well. Only few works address information-flow security in multithreaded object-oriented languages. In [22], while there are objects and threads in the language, only the part representing concurrency is dealt with formally. There is also an extension of the protection/hiding-based method into multithreaded object-oriented programs that suggests using special schedulers as part of the trusted computing base [23]. Although interesting, this solution requires programs to run on machines with such schedulers.

In this paper, we first introduce a new internal timing channel that arises from dynamic dispatch in the presence of multithreading. To ascertain how such channels may be created and prevented in a language, we first add constructs for multithreading to a subset of Java known as Middleweight Java [24]. The next step is to give an appropriate noninterference property. The property is such that the example covert channel and its ilk do not occur in programs satisfying the property no matter which scheduler is employed by the run-time system. It is indeed a scheduler-independent security property. Finally, we give a security type system and prove that it is sound. The soundness of our type system ensures that any well-typed program of our language satisfies the proposed noninterference property.

In brief, we specify and enforce a scheduler-independent information-flow policy in a subset of the full-blown programming language Java so that typable programs are provably secure. The proposed language is such that its programs can be run as ordinary Java programs if the annotations specifying the security policy are eliminated. The following are the main outcomes of this paper.

- A multithreaded model language for Java with a fairly rich set of features making it analogous to Java while remaining amenable to in-depth analysis.
- A demonstration of how new internal timing channels may emerge as a result of the coexistence of object orientation and multithreading.
- A more permissive scheduler-independent noninterference property addressing internal timing channels.
- A type system that provably enforces the noninterference property in programs of the presented language.

The new timing channel will be scrutinized in Section 3 where we develop a middleweight Java with the features of object orientation and multithreading. We can, however, outline the very nature of the channel by the following piece of code.

```
th1: x.m1(); l=0;
th2: x.m2(); l=1;
```

In this code, `x` is a variable of type `C` which is a class containing the two methods `m1` and `m2` and having two subclasses in which `m1` and `m2` are overridden. The code may leak sensitive information if the subclass from which `x` is instantiated at run-time is determined by some confidential values. The leakage indeed happens when in one of the subclasses of `C` the time