Science of Computer Programming 97 (2015) 122–126

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Understanding probabilistic software leaks[☆]

Gregor Snelting

Karlsruhe Institute of Technology, Germany

ARTICLE INFO

Article history:
Received 25 September 2013
Accepted 5 November 2013
Available online 21 November 2013

Keywords:
Software security
Noninterference
Program analysis

ABSTRACT

Probabilistic security leaks in multi-threaded programs exploit nondeterminism and interleaving. Probabilistic leaks does not leak secret values directly, but secret values influence the probability of public events. The article explains probabilistic leaks, and discusses various methods for checking probabilistic noninterference.

This article was written to Paul Klint's 65th birthday; it honours his life-long quest for understanding software. Dear Paul, Happy Birthday!

1. Introduction: language-based software leaks

Software security will be of overwhelming importance for future software systems. In particular, integrity and confidentiality must be guaranteed: critical computations must not be manipulated from outside, and secret values must not flow to public ports. But classical techniques (e.g., certificates) do not perform a fine-grained analysis of program behaviour and thus cannot guarantee integrity and confidentiality. Indeed, the Stuxnet worm used stolen certificates. Additional program analysis must be used to provide true security guarantees. This analysis is called information flow control (IFC).

IFC research started over 20 years ago, and since 15 years an international community emerged which seriously investigates IFC foundations and algorithms. Today, several IFC tools are available, including commercial ones. But note that IFC does not replace more classical software security approaches – IFC adds an additional dimension to security techniques.

In this overview article, we discuss IFC algorithms for sequential and concurrent programs, which check confidentiality.[1] A wealth of research has emerged which analyses source or machine code for confidentiality leaks. Most IFC algorithms guarantee some form of *noninterference*: variables resp. program statements are classified *secret* ("high") or *public* ("low"), and secret values are not allowed to influence publicly visible behaviour. For variable v, its classification $c(v)$ thus is $c(v) = \mathit{Low}$ or $c(v) = \mathit{High}$.[2] The classical noninterference definition for sequential programs is based on low-equivalency of program states: two states s, s' are low equivalent, written $s \cong_{low} s'$, if they coincide on low variables: $\forall v \in \mathit{dom}(s) \cap \mathit{dom}(s') : c(v) = \mathit{Low} \implies s(v) = s'(v)$.[3] A program P is noninterferent if

$$\forall s, s' : \quad s \cong_{low} s' \quad \implies \quad [\![P]\!]s \cong_{low} [\![P]\!]s'$$

☆ The work described in this paper was supported by DFG, grants Sn11/10-2 and Sn11/12-1, in the scope of the SPP 1496 "Reliably secure software systems".

1 Integrity is technically dual to confidentiality, hence any confidentiality analysis can easily be transformed into an integrity analysis, and vice versa.

2 Often complex lattices of security levels are used, not just the two-element lattice $\mathit{Low} < \mathit{High}$.

3 More complex state structures are possible, including nested blocks, heaps etc.

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.11.008

```
1  void main():
2    x = inputPIN();
3    // inputPIN is High
4    // x,y are Low
5    if (x < 1234)
6       print(0);
7    y = x;
8    print(y);
```

```
1  void thread_1():
2    x = input();
3    print(x);
4
5  void thread_2():
6    y = inputPIN();
7    x = y;
```

```
1  void thread_1():
2    x = 0;
3    print(x);
4
5  void thread_2():
6    y = inputPIN();
7    while (y != 0)
8      y−−;
9    x = 1;
10   print(2);
```

Fig. 1. Examples for explicit and implicit leaks (left), for a possibilistic leak (middle), and for a probabilistic leak (right).

```
void main():
  x = inputPIN();
  while (x > 0)
    print("x");
    x−−;
  while (true)
    skip;
```

```
void main():
x=inputPIN();
while (x>0)
  print ("x");
  x−−;
  if (x==0)
    {while (true)
      skip;}
```

```
void main():
  x = inputPIN();
  while (x != 0)
    x−−;
  print(1);
```

```
void main():
  x = inputPIN();
  while (x == 0)
    skip;
  print("x");
  while (x == 1)
    skip;
  print("x");
  ...
  while (x == 42)
    skip;
  print("x");
  ...
```

Fig. 2. All programs contain termination leaks and gradually leak (part of) the PIN. Termination-insensitive IFC algorithms will discover some, but not all of these leaks.

Noninterference expresses that for any two initial low-equivalent program states s, s', the final states after execution of P must also be low-equivalent.[4] That is, different values of a high variable can never influence low program results, which implies that P guarantees confidentiality. Noninterference thus provides security guarantees (only) on the language level; it does not consider physical side channels, corrupt operating systems, defective hardware etc.

In Fig. 1 left,[5] $c(\texttt{inputPIN}) = \textit{High}$ and $c(\texttt{x}) = c(\texttt{y}) = \textit{Low}$; initially, $\mathbf{x} = \mathbf{y} = 0$. Thus all initial states are low-equivalent, but different `inputPIN`s will cause two different final values of `y` to be printed. Hence the noninterference definition is broken: the final states are not always low-equivalent. Worse, if the `inputPIN` is less than 1234, an additional “0” is printed. This example contains explicit and implicit leaks. *Explicit leaks* arise if (parts of) high values are copied to low variables (in the example, we have the assignment chain $\texttt{inputPIN} \to \texttt{x} \to \texttt{y}$). *Implicit leaks* arise, if high values influence the control flow (which may cause further explicit leaks or other observable behaviour); in the example, the `if` is indirectly influenced by the `inputPIN`.

Fig. 1 middle presents a *possibilistic leak*. Such leaks depend on the interleaving of concurrent threads operating on shared memory. A possibilistic leak arises if an interleaving exists which leads to an explicit or implicit leak. In the example, the interleaving statement sequence 2, 6, 7, 3 causes the PIN to be copied to `x` and printed. In Fig. 1 right, a *probabilistic leak* can be seen. The leaking depends on interleaving in a more subtle way: imagine the PIN is very high, hence the loop 7/8 has a long execution time. This will increase the probability that the scheduler will execute statement 3 before statement 9, and “0” is printed. If the PIN is low, the probability is higher that 9 is executed before 3, and “1” is printed. Hence a high value influences the probability of a low event (namely statement 9), which in turn influences the probabilities for different public outputs. Note that every possibilistic leak is also a probabilistic leak, but not vice versa: the example does not have a possibilistic leak, as an explicit or implicit flow from PIN to `x` is impossible for any interleaving. Still, by running the program several times, the attacker gathers information about the secret PIN.

Timing leaks exploit different runtime of e.g., **then**- and **else**- part of an **if**. Sometimes such runtime differences can be measured by physical side channels. Sometimes probabilistic leaks are based on runtime, such as the above example. However not all timing leaks are probabilistic leaks.[6]

Termination leaks are even more subtle, in particular in combination with probabilistic leaks. Consider the programs in Fig. 2. If the attacker can decide whether the observed program is in a loop or not (by means outside program analysis – e.g., physical side channels – as the halting problem is undecideable), she can for all four programs conclude whether the PIN was < 0 (in fact, all four programs have identical low-observable behaviour). It is known that termination leaks in interactive programs can leak an arbitrary amount of information [2]. Standard noninterference does not cover termination leaks, as it assumes termination of all runs.

[4] The noninterference definition can be expanded to cover input and output. For the time being, input statements are assumed to write variables in the initial state s; output statements are assumed to read variables from the final state $[\![P]\!]s$.

[5] Examples are partially extracted from [1].

[6] Timing leaks are an issue of its own and not considered in this paper.