Safety versus Secrecy

(Invited Paper)

Dennis Volpano*
Naval Postgraduate School
Monterey, CA 93943, USA
volpano@cs.nps.navy.mil

Abstract. Safety and secrecy are formulated for a deterministic programming language. A safety property is defined as a set of program traces and secrecy is defined as a binary relation on traces, characterizing a form of Noninterference. Safety properties may have sound and complete execution monitors whereas secrecy has no such monitor.

1 Introduction

It is often argued that information flow is not safety. One argument is refinement based and originates with Gray and McLean [5]. They observed that for nondeterministic systems, a class of information flow properties, namely the Possibilistic Noninterference properties, are not safety properties. The reason is because they are not preserved under replacement of nondeterminism in a system with determinism. An example is an implementation of nondeterministic scheduling using a round-robin time-sliced scheduler [8]. A possibilistic property basically asserts that certain system inputs do not interfere with the possibility of certain events. So nondeterminism is essential to such properties. A safety property, on the other hand, is insensitive to this kind of refinement. Another argument commonly heard is that information flow is a predicate of trace sets whereas safety is a predicate of individual traces. This argument can be applied to deterministic systems. We examine it more carefully and present a secrecy criterion for programs that relates secrecy and safety.

2 A characterization of safety properties

Consider a deterministic programming language with variables:

\[
\begin{align*}
\text{(exp)} & \quad e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 = e_2 \\
\text{(cmd)} & \quad c ::= x := e \mid c_1 ; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c
\end{align*}
\]

* This material is based upon activities supported by the National Science Foundation under Agreement No. CCR-9612345 [sic].
Here \( x \) stands for a variable and \( n \) for an integer literal. Integers are the only values; we use 0 for false and nonzero for true. Note that expressions do not have side effects, nor do they contain partial operations like division.

A transition semantics is given for the language in Fig. 1. We assume that expressions are evaluated atomically. Thus we simply extend a memory \( \mu \) in the obvious way to map expressions to integers, writing \( \mu[e] \) to denote the value of expression \( e \) in memory.

\[
\begin{align*}
(\text{UPDATE}) & \quad x \in \text{dom}(\mu) \\
& \quad (x := e, \mu) \rightarrow (x := \mu[e], \mu')
\end{align*}
\]

\[
\begin{align*}
(\text{SEQUENCE}) & \quad (c_1, \mu) \rightarrow (\mu') \\
& \quad (c_1; c_2, \mu) \rightarrow (c_2, \mu') \\
& \quad (c_1, \mu) \rightarrow (c_1', \mu') \\
& \quad (c_1; c_2, \mu) \rightarrow (c_1', c_2, \mu')
\end{align*}
\]

\[
\begin{align*}
(\text{BRANCH}) & \quad \mu(e) \neq 0 \\
& \quad (\text{if } e \text{ then } c_1 \text{ else } c_2, \mu) \rightarrow (c_1, \mu) \\
& \quad \mu(e) = 0 \\
& \quad (\text{if } e \text{ then } c_1 \text{ else } c_2, \mu) \rightarrow (c_2, \mu)
\end{align*}
\]

\[
\begin{align*}
(\text{LOOP}) & \quad \mu(e) = 0 \\
& \quad (\text{while } e \text{ do } c, \mu) \rightarrow (\mu) \\
& \quad \mu(e) \neq 0 \\
& \quad (\text{while } e \text{ do } c, \mu) \rightarrow (c; \text{while } e \text{ do } c, \mu)
\end{align*}
\]

Fig. 1. Transition semantics

The rules define a transition relation \( \rightarrow \) on configurations. A configuration \( m \) is either a pair \((c, \mu)\), where \( c \) is a command and \( \mu \) is a memory, or simply a memory \( \mu \). We define the reflexive transitive closure \( \rightarrow^* \) in the usual way. First \( m \rightarrow^0 m \), for any configuration \( m \), and \( m \rightarrow^k m'' \), for \( k > 0 \), if there is a configuration \( m' \) such that \( m \rightarrow^{k-1} m' \) and \( m' \rightarrow m'' \). Then \( m \rightarrow^* m' \) if \( m \rightarrow^k m' \) for some \( k \geq 0 \).

A trace is a (possibly infinite) derivation sequence \( m_1 \rightarrow m_2 \rightarrow \cdots \) with finite prefixes \( m_1 \rightarrow m_2 \), \( m_1 \rightarrow m_2 \rightarrow m_3 \), and so on. And if \( \sigma \) is a trace then so is every prefix of \( \sigma \).

**Definition 1.** A safety property is a set \( S \) of traces such that for all traces \( \sigma \), \( \sigma \) is in \( S \) iff every finite prefix of \( \sigma \) is in \( S \). A program is safe if every trace of it belongs to \( S \).

The “only-if” direction guarantees \( S \) is prefix closed, and the “if” direction allows us to reject an infinite trace by examining only a finite amount of it. If there is