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:

$$(exp)\;\; e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 = e_2$$
$$(cmd)\;\; c ::= x := e \mid c_1 ; c_2 \mid \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } e \textbf{ do } c$$

* This material is based upon activities supported by the National Science Foundation under Agreement No. CCR-9612345 [sic]. This paper appears in Proceedings of the 6th Int'l Symposium on Static Analysis, Venezia Italy, 22-24 Sep 1999.

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 μ in the obvious way to map expressions to integers, writing $\mu(e)$ to denote the value of expression e in memory μ.

(UPDATE) $$\frac{x \in dom(\mu)}{(x := e, \mu) \longrightarrow \mu[x := \mu(e)]}$$

(SEQUENCE) $$\frac{(c_1, \mu) \longrightarrow \mu'}{(c_1; c_2, \mu) \longrightarrow (c_2, \mu')}$$

$$\frac{(c_1, \mu) \longrightarrow (c_1', \mu')}{(c_1; c_2, \mu) \longrightarrow (c_1'; c_2, \mu')}$$

(BRANCH) $$\frac{\mu(e) \neq 0}{(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \mu) \longrightarrow (c_1, \mu)}$$

$$\frac{\mu(e) = 0}{(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \mu) \longrightarrow (c_2, \mu)}$$

(LOOP) $$\frac{\mu(e) = 0}{(\mathbf{while}\ e\ \mathbf{do}\ c, \mu) \longrightarrow \mu}$$

$$\frac{\mu(e) \neq 0}{(\mathbf{while}\ e\ \mathbf{do}\ c, \mu) \longrightarrow (c; \mathbf{while}\ e\ \mathbf{do}\ c, \mu)}$$

Fig. 1. Transition semantics

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

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

Definition 1. *A safety property is a set S of traces such that for all traces σ, σ is in S iff every finite prefix of σ 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 an infinite trace that is not in S, then it must have a finite prefix that is also not in S. Hence safety cannot rule out behaviors that amount to reaching some execution state infinitely often.

We also assume that the set of all finite traces in S is recursive. Although this need not be true of a safety property, it seems reasonable given that one typically identifies a safety property with the ability to enforce it at runtime by examining program traces of finite length.

3 A characterization of secrecy

We want to talk about secrecy in programs of our deterministic language so how should secrets be introduced? Well there is nothing intrinsically secret about any integer so we should forget about associating secrecy with values. Instead, we associate secrecy with the *origin* of a value which in our case will be the free variables of a program. So each variable is either high (secret) or low (public). The idea is that any *initial* value of a high variable is assumed to be secret merely by virtue of being stored in a high variable. The initial value of a low variable is not secret.

This origin-view of secrecy differs from the view held by others working with assorted lambda calculi and type systems for secrecy [1,3]. There, secrecy is associated with values like boolean constants. It does not seem sensible to attribute any level of security to such constants. After all, what exactly is a "high-security" boolean? Semantically, there is nothing that makes it high or low. Basic constants can be treated as high or low, and therefore we take the view that they should be typed polymorphically in any type system where levels of classification become (partially-ordered) types.

We need to talk about secrecy violations. But what constitutes a violation? Suppose k is a low variable and h is a high variable with initial value 17. Is the assignment $k := 17$ in violation of secrecy? Presumably not since it just got lucky and does not reliably reveal the value of h as h varies. On the other hand, $k := h$ would be a violation.

As another example, consider

$$k := h;\ k := k - h$$

Does it exhibit a violation? Despite the first assignment, we might still regard the composition as secure since h is only temporarily stored in k which always has final value zero. One might wonder though whether even temporary storage is a violation. It would be if execution could be suspended for some reason, say in an interleaved execution environment, and k's contents inspected. For now, we shall stay with deterministic sequential programs and focus on what they are capable of doing upon normal termination. In this case, the composition would be secure. This also allows us to say that

$$h := k;\ k := h$$

too is secure since there is no way to update h between the assignments.

One can begin to see the subtlety in deciding what constitutes a secrecy violation. In the end, it comes down to what is observable by users and programs. Users can make *external* observations of running programs and system behavior on chosen inputs in order to learn secrets. Running time, resource usage, exceptions and so on are all valuable sources of information, provided by even well-designed programs, that can be observed outside a program and exploited. Programs, in contrast, make *internal* observations in that they are limited to whatever observations their semantics prescribe. Controlling these observations is much more tractable as long as implementations are faithful to the semantics[1] and any program translation preserves the secrecy criterion of interest. With a semantics at least, we have a means of specifying and reasoning about the behaviors of programs and the observations they can make. We shall concern ourselves with internal observations only. This is still useful. For instance, it treats a Trojan Horse in mobile code that attempts to leak client secrets.

So now that we have some intuition behind secrecy, how do we formalize it? There are a number of different techniques such as process calculi equivalence [2], a PER model [6], and operational formulations [8–10]. In order to contrast secrecy with safety, we give a trace-based description. It is useful to first define a notion of configuration equivalence. Memories μ and μ' are equivalent, written $\mu \sim \mu'$, if $\mu(v) = \mu'(v)$ for all low variables v. And $(c, \mu) \sim (c', \mu')$ if c and c' are syntactically equal and $\mu \sim \mu'$.

Definition 2. *Secrecy is a binary relation R on traces where $R(\sigma, \sigma')$ is true unless σ has the form $m_1 \longrightarrow m_2 \longrightarrow \cdots \longrightarrow \mu$, σ' has the form $m'_1 \longrightarrow m'_2 \longrightarrow \cdots \longrightarrow \mu'$, $m_1 \sim m'_1$ and $\mu \not\sim \mu'$. A program is secret if R relates every pair of its traces.*

Basically, secrecy is asserting that the *final* value of any low variable does not depend on the *initial* values of high variables. This definition applies only to deterministic programs. Notice that a program may be secret even though it has a finite trace and an infinite trace whose starting configurations are equivalent. In other words, termination of a secret program can be affected by differences in the initial values of high variables.

4 Contrasting secrecy with safety

Notice that secrecy relates program executions whereas a safety property does not. This is the essential difference between them. There are some interesting consequences of this difference in terms of enforcing secrecy versus safety.

Suppose we take the view that a program may be unsafe but we won't worry about its offending traces unless one of them tries to emerge during the current execution. So we don't try to convince ourselves once and for all that a program is safe. Instead we accept the fact it may be unsafe and put our trust in an

[1] Knowing when an implementation is faithful can also be tricky.

execution monitor to guard against *unsafe* behavior. This is an old idea from operating systems. A monitor works by monitoring the execution of a program and trapping it before it violates the policy being enforced [7]. It relies only on information available at runtime and does not examine the entire program being executed. Recovery from such traps may be possible in some applications. It is in these applications that monitoring is appealing because the complement of deciding whether a program is safe (call it unsafety) may be r.e. when safety is not. When safety is not r.e., we are immediately faced with incompleteness in any sound and r.e. logic for analyzing it. If the logic were complete then safety would be r.e. since we would have a way to accept safe programs: simply hand the given program off to the machine M accepting programs that have proofs in the logic. If M accepts then we accept and we know we're correct because the logic is sound. And if the program is safe, then, by completeness, it has a proof and therefore M will accept it. Incompleteness can be an obstacle in practice, depending on the logic. Execution monitoring avoids it.

Monitoring can also dovetail nicely with a machine M accepting unsafety. M might cycle through all memories (suitably encoded) and run a given program on each of them for at most some fixed number of steps, where the memories and number of steps are governed by pair generation. If the unsafe behavior reveals itself within the number of steps allowed (guaranteed to be detectable by M since the finite traces of a safety property form a recursive set), then M accepts. An execution monitor for unsafety is essentially a lazy version of M. Eventually the monitor might decide that a given program is unsafe but that does not concern us unless the current run demonstrates it.

We need to be clear on the terms soundness and completeness. The monitor is a lazy version of M which accepts unsafety. Since M accepts unsafety, we have that if a program is unsafe then M will say so (completeness) and if M says a program is unsafe, it is indeed unsafe (soundness). Therefore, if M never says a program is unsafe then the program is safe. This we take as a soundness criterion for M (and the monitor) with respect to the safety property at hand. Likewise, if a program is safe, then M never says otherwise. And this we take as a completeness criterion for M relative to safety.

A similar technique can be used to prove that the complement of deciding whether a program is secret is r.e.. One can encode a pair of memories and adopt some convention for determining values of low variables, and then run the given program for at most a fixed number of steps on each memory in a generated pair when the memories are equivalent. If the runs terminate yielding inequivalent memories, then accept. But unlike the complement of safety, the technique here does not dovetail with execution monitoring because it requires two memories. Monitoring involves only one, that of the current execution. So for this notion of secrecy, monitoring cannot be employed as a way to guard against secrecy violations as it was used to guard against safety violations. In fact, we can be more rigorous. As we shall see, one can prove there is no policy, implemented by an execution monitor, that implies secrecy and is complete. In contrast, there are many safety properties that have sound and complete execution monitors. So what alternatives are there for enforcing secrecy?

One approach is to turn to a static analysis whereby we attempt to show once and for all that a given program is secret. But we will be faced with incompleteness in any sound and r.e. system for reasoning about secrecy because determining whether a program is secret is undecidable. Decidable type systems fall into this category [10]. Instead, one may adopt a very expressive logic and use verification conditions for establishing secrecy without worrying about mechanizing proofs. Work along these lines is described in [4].

In the next section, we shall see an example of a program secrecy criterion implied by a policy that is implemented using an execution monitor. It is called weak secrecy. A disadvantage of weak secrecy is that it ignores indirect dependencies caused by branching—hence the term "weak". As a result, some programs satisfy weak secrecy but are not secret. But there are also secrect programs that do not satisfy weak secrecy, reflecting a basic requirement of safety. So neither property implies the other. The monitor is sound but incomplete for weak secrecy. It may trap a program that satisfies weak secrecy.

5 Weak secrecy

Every trace has a corresponding branch-free program formed by sequencing updates from those steps of the trace whose derivations are rooted with updates. For instance, if $k, h \in dom(\mu)$ and $\mu(h) = 0$, then corresponding to the trace

$$
\begin{array}{ll}
\quad\ (k := h;\ \textbf{if}\ h\ \textbf{then}\ k := 1\ \textbf{else}\ k := 0,\ \mu) & (m_1) \\
\longrightarrow (\textbf{if}\ h\ \textbf{then}\ k := 1\ \textbf{else}\ k := 0,\ \mu[k := \mu(h)]) & (m_2) \\
\longrightarrow (k := 0,\ \mu[k := \mu(h)]) & (m_3) \\
\longrightarrow \mu[k := \mu(h)][k := 0] & (m_4)
\end{array}
$$

is the branch-free program $k := h;\ k := 0$. Notice that by rules (LOOP) and (BRANCH), the corresponding program for a trace may be empty.

Now we say that a program is *weakly secret* if every trace of it has a secret branch-free program. For instance, the program in the preceding example is not weakly secret. Traces $m_1 \longrightarrow m_2$ and $m_1 \longrightarrow m_2 \longrightarrow m_3$ do not have secret branch-free programs, but $m_1 \longrightarrow m_2 \longrightarrow m_3 \longrightarrow m_4$ does. It may seem that we still have not defined a criterion for program secrecy that follows from some policy implemented by execution monitoring since we still cast our definition in terms of secrecy which relates program executions. But there is a policy that implies weak secrecy and it can be implemented by an execution monitor.

5.1 A policy for weak secrecy and its monitor

An execution monitor is given in Fig. 2 as a set of rules governing transitions that the monitor can make. Each transition has the form

$$(c, \mu), q \xrightarrow{M} m, q'$$

where q and q' are states $\{k\}$ or $\{k, h\}$. The monitor is equipped to handle executions of programs with only two variables, namely k and h, which are low and high variables respectively. A state indicates those variables whose values at that point are independent of initial values of h. This of course may change during execution depending upon updates to h.

The policy is captured by the (UPDATE) rules. If we take state $\{k\}$ to be the initial state, then the third (UPDATE) rule, for instance, allows a transition to state $\{k, h\}$ because h is the target of the assignment and h does not occur in the right side e. Thereafter, h is treated as a low variable in state $\{k, h\}$. Notice that

(UPDATE)

$$\frac{k \in dom(\mu), \quad h \notin e}{(k := e, \mu), \{k\} \xrightarrow{M} \mu[k := \mu(e)], \{k\}}$$

$$\frac{h \in dom(\mu), \quad h \in e}{(h := e, \mu), \{k\} \xrightarrow{M} \mu[h := \mu(e)], \{k\}}$$

$$\frac{h \in dom(\mu), \quad h \notin e}{(h := e, \mu), \{k\} \xrightarrow{M} \mu[h := \mu(e)], \{k, h\}}$$

$$\frac{x \in dom(\mu)}{(x := e, \mu), \{k, h\} \xrightarrow{M} \mu[x := \mu(e)], \{k, h\}}$$

(SEQUENCE)

$$\frac{(c_1, \mu), q \xrightarrow{M} \mu', q'}{(c_1; c_2, \mu), q \xrightarrow{M} (c_2, \mu'), q'}$$

$$\frac{(c_1, \mu), q \xrightarrow{M} (c_1', \mu'), q'}{(c_1; c_2, \mu), q \xrightarrow{M} (c_1'; c_2, \mu'), q'}$$

(BRANCH)

$$\frac{\mu(e) \neq 0}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu), q \xrightarrow{M} (c_1, \mu), q}$$

$$\frac{\mu(e) = 0}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu), q \xrightarrow{M} (c_2, \mu), q}$$

(LOOP)

$$\frac{\mu(e) = 0}{(\textbf{while } e \textbf{ do } c, \mu), q \xrightarrow{M} \mu, q}$$

$$\frac{\mu(e) \neq 0}{(\textbf{while } e \textbf{ do } c, \mu), q \xrightarrow{M} (c; \textbf{while } e \textbf{ do } c, \mu), q}$$

Fig. 2. An execution monitor

once an evaluation reaches state $\{k, h\}$, it remains in $\{k, h\}$ thereafter. In state $\{k, h\}$, the monitor no longer has any effect on executions. This is where the semantics of Fig. 1 and the monitor merge in the sense that for every command c, $(c, \mu) \longrightarrow m$ if and only if $(c, \mu), \{k, h\} \xrightarrow{M} m, \{k, h\}$.

Remark 1. One can think of the states in a transition as inherited and synthesized attributes. Generalizing the execution monitor to handle more variables can be done by introducing a set I of variables, each of whose value is independent of the initial value of any high variable. There would actually be only two (UPDATE) rules. The first rule's hypothesis would, for an assignment $x := e$, require $\forall h \in \mathit{high}.\ h \in I \vee h \notin e$. Its synthesized attribute would be $I \cup \{x\}$. The second rule's hypothesis would require that x is a high variable and $\exists h \in \mathit{high}.\ h \notin I \wedge h \in e$. Its synthesized attribute would be simply I.

We can regard a set of traces of the monitor as a safety property related to secrecy by the following theorem:

Theorem 1. *Let σ be a trace of the monitor starting in state $\{k\}$. Then every finite prefix of σ has a secret branch-free program.*

If the monitor never traps a given program on any input, when started in state $\{k\}$, then the program is weakly secret. However, a program may be weakly secret yet get trapped (e.g. $k := h - h$). The monitor also traps a secret program:

$$k := h;\ k := k - h$$

Here there is a trace whose branch-free program is just $k := h$ which is not secret. One might consider altering the monitor in some way to admit all executions of this program but then its traces would no longer be prefix closed as a trace for $k := h$ would not exist if the monitor's policy implies secrecy. It follows then that there is no monitor-enforced policy that is sound and complete for secrecy since the set of all traces of every monitor is prefix closed. Simply put, if the monitor executes $k := h$, then it's unsound, and if it doesn't, then it's incomplete.

The monitor also ignores indirect dependencies. For instance, it does not trap

$$\textbf{if}\ h\ \textbf{then}\ k := 1\ \textbf{else}\ k := 0$$

even though the program is not secret.

6 Concluding remarks

Execution monitoring has been a useful mechanism for implementing various policies. It is important to distinguish policies from properties. A policy implies a property, and in some cases, may be more restrictive than it needs to be in order to imply the property. The execution monitor presented here implements a policy that implies weak secrecy in the sense that if it never traps a given program on any input, when started in state $\{k\}$, then the program is weakly secret. It does not however imply secrecy. In fact, no policy implemented by an execution monitor can imply secrecy and be complete.

An interesting direction to pursue is completeness of the monitor for weak secrecy, that is, trying to extend the monitor so that it never traps a weakly secret program. Doing this for a more realistic set of expressions would be challenging. We assumed that expressions are executed atomically and that the monitor can inspect an expression at runtime. But expressions obviously can be far more complex, involving function calls, conditional expressions, exceptions and side effects. One cannot assume these sorts of expressions execute atomically.

Acknowledgments

I would like to thank Geoffrey Smith, for reviewing this paper, and Fred Schneider for helpful discussions on safety and execution monitoring.

References

1. Martín Abadi. Secrecy in programming-language semantics. In *Proc. 15th Mathematical Foundations of Program Semantics*, pages 1–14, April 1999.
2. R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1994/1995.
3. Nevin Heintze and Jon Riecke. The SLam Calculus: Programming with secrecy and integrity. In *Proceedings 25th Symposium on Principles of Programming Languages*, pages 365–377, San Diego, CA, January 1998.
4. K.R.M. Leino and R. Joshi. A semantic approach to secure information flow. In *Proc 4th Int'l Conference on Mathematics of Program Construction*, pages 254–271. Lecture Notes in Computer Science 1422, 1998.
5. John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings 1994 IEEE Symposium on Security and Privacy*, pages 79–93, Oakland, CA, May 1994.
6. A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *Proc. 8th European Symposium on Programming.* Lecture Notes in Computer Science 1576, March 1999.
7. F.B. Schneider. Enforceable security policies. Technical Report TR98-1664, Cornell University, January 1998.
8. Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings 25th Symposium on Principles of Programming Languages*, pages 355–364, San Diego, CA, January 1998.
9. Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proceedings 10th IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.
10. Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.