Termination-Insensitive Noninterference Leaks More Than Just a Bit

Aslan Askarov[1], Sebastian Hunt[2], Andrei Sabelfeld[1], and David Sands[1]

[1] Chalmers University of Technology, Sweden
[2] City University, London

Abstract. Current tools for analysing information flow in programs build upon ideas going back to Denning's work from the 70's. These systems enforce an imperfect notion of information flow which has become known as *termination-insensitive noninterference*. Under this version of noninterference, information leaks are permitted if they are transmitted purely by the program's termination behaviour (i.e., whether it terminates or not). This imperfection is the price to pay for having a security condition which is relatively liberal (e.g. allowing while-loops whose termination may depend on the value of a secret) and easy to check. But what is the price exactly? We argue that, in the presence of output, the price is higher than the "one bit" often claimed informally in the literature, and effectively such programs can leak all of their secrets. In this paper we develop a definition of termination-insensitive noninterference suitable for reasoning about programs with outputs. We show that the definition generalises "batch-job" style definitions from the literature and that it is indeed satisfied by a Denning-style program analysis with output. Although more than a bit of information can be leaked by programs satisfying this condition, we show that the best an attacker can do is a brute-force attack, which means that the attacker cannot reliably (in a technical sense) learn the secret in polynomial time in the size of the secret. If we further assume that secrets are uniformly distributed, we show that the advantage the attacker gains when guessing the secret after observing a polynomial amount of output is negligible in the size of the secret.

1 Termination-Insensitive Noninterference

Does the following program leak its secret?

```
for i = 0 to secret                                (Program 1)
    output i on public_channel
```

Let us assume that the secret is a natural number. The program simply counts from zero up to the value of the secret, so it is clearly not secure. What about the following minor variation?

```
for i = 0 to secret                                (Program 1a)
    output i on public_channel
while true do skip
```

The only difference here is that after performing its output the program goes into a non productive infinite loop. Is it reasonable to consider program 1a to be secure if program 1 is not? Now consider the following program:

S. Jajodia, and J. Lopez (Eds.): ESORICS 2008, LNCS 5283, pp. 333–348, 2008.

```
for i = 0 to maxNat (                          (Program 2)
    output i on public_channel
    if (i = secret) then (while true do skip)
)
```

Program 2 is semantically equivalent to program 1a. But it has an important difference. Program 2 is deemed acceptable by state-of-the-art information flow analysis tools such as Jif [MZZ+08], FlowCaml [Sim03], and the SPARK Examiner [BB03,CH04]. Such tools are, at their core, built on ideas going back to Denning and Denning's seminal paper about certifying programs for secure information flow [DD77]. The programs 1 and 1a, for example, would be rejected as insecure because they contain a "high" loop (a loop depending of the value of a secret) which assigns to a "low" variable (a public channel) causing an *implicit* information flow from secret to public.

For program 2 however, a Denning-style certification (and in particular all the concrete tools mentioned above) would say that the program is secure. Such an analysis would reason as follows: the outer loop is "low" because the loop condition does not refer to the secret, and so the output statement is permitted. The if-expression, on the other hand, is considered secure simply because it does not raise any exceptions or assign to anything at all.

In order to justify Denning-style analyses, an imperfect notion of information flow which has become known as *termination-insensitive noninterference*[1] is widely used. Under this version of noninterference, information leaks are permitted if they are transmitted purely by the program's termination behaviour. But what is the price to pay for having a relatively liberal security condition? Program 2 above shows that, in the presence of output, the price is higher than the "one bit" often claimed informally in the literature, and effectively such programs can leak all of their secrets.

Note that the same issue arises with other forms of abnormal termination than divergence. As we illustrate in Section 6, a stack/heap overflow or other computation with an uncaught runtime exception instead of the infinite loop would lead to the same problems, which suggests that we cannot reduce the termination channel to a special case of a timing channel. The results in this paper are not limited to any particular form of abnormal termination, although, for simplicity, we model only divergence explicitly.

Batch-job noninterference. A "batch-job" style of termination-insensitive security has been widely used to argue the correctness of Denning-style program analyses. This style ignores nonterminating runs and assumes that the attacker can observe only the final state of a computation. In particular, the batch-job notion of termination-insensitive noninterference corresponds to the correctness condition by Volpano et al. [VSI96] for Denning-style analysis:

Definition 1 (BTINI). *A deterministic program C satisfies* batch-job termination-insensitive noninterference *(BTINI) if, for any memories M and N that agree on public (low) variables, the final memories produced by running C on M and on N also agree on public variables (provided that both runs terminate successfully).*

[1] This terminology referring to insensitivity to the termination channel (for signalling information through the termination or nontermination of a computation), seems to have been coined in [SS99], although the concept arises already in discussions from e.g. [Fen74].

The above definition permits, for example:

```
if (secret = 0) then (while true do skip)
public := 0
```

The general intuition here is that such programs leak only a little – at most one bit per run.

Despite its popularity, the above definition is wholly unsuitable if attackers can observe intermediate results such as outputs. For such programs we cannot turn a blind eye when programs fail to terminate, otherwise we would deem the following program secure:

```
output secret on public_channel
while true do skip
```

In [VSI96], "a program that needs to 'write output' does so by an assignment to an explicit location". Similar issues with inappropriate use of batch-job noninterference arise elsewhere. For example, both Askarov et al. [AHS06] and Le Guernic et al. [LBJS08] consider languages with output, but their noninterference conditions ignore divergent runs. Askarov and Sabelfeld [AS07] model an attacker who observes intermediate values – but only if the program terminates (a fact also raised in [BNR08]).

A related problem is the belief that as long as the attacker "cannot observe termination" then a program leaks at most one bit. As our opening examples show, this is clearly not the case once output is possible. For example, JFlow/Jif features outputs but still appeals to the "one-bit" argument: "JFlow treats this error ⟨heap exhaustion⟩ as fatal, preventing it from communicating more than a single bit of information per program execution" [Mye99].

One solution to these problems would be to abandon the weaker notion of security that is inherent in a Denning-style analysis. But Denning-style termination-insensitive analyses are popular not because of the semantic notion of security that they enforce, but because they allow more programs. Alternative stronger security conditions would require either a difficult liveness analysis to show the absence of divergent behaviour, or a draconian restriction on the programs that can be written (e.g., no loops depending on secret guards are allowed [VS97]).

Generalising BTINI. So, what is the right definition of termination-insensitive noninterference for languages with output, and moreover what security guarantees does it provide?

In this paper we define a suitable notion of termination-insensitive noninterference (Section 2), which we believe correctly captures the security property guaranteed by Denning-style program analyses. Instead of considering only terminating runs, this notion incorporates insensitivity to divergence in intermediate states. The formulation is intuitive because it is based on a more explicit attacker model which reasons about an attacker's knowledge as it evolves during a run, rather than the more standard "two run" style presentations of noninterference properties. We substantiate our claim that this is a suitable condition for a Denning-style analysis by showing that a formalisation [VSI96] of Denning's analysis for a language with output satisfies this condition (Section 3).

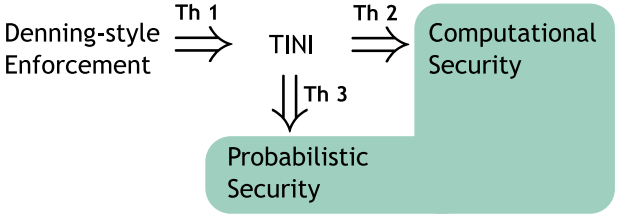


Fig. 1. Our results on termination-insensitive noninterference (TINI)

We then show that program 2 given above is the best an attacker can do – a brute force search of the space of possible secrets. We present this as two results. In Section 4, we show that it is impossible to reliably leak the secret by a program that satisfies termination-insensitive noninterference in polynomial time in the size of the secret. In Section 5, we show that if the secret is uniformly distributed, then the probability of the attacker guessing the secret after observing a polynomial number of outputs (again, in the size of the secret) gives only a negligible advantage over guessing the secret without running the program.

We discuss further examples and simple experiments with Jif, FlowCaml and SPARK Examiner in Section 6 and conclude in Section 7.

Figure 1 schematically illustrates the main contributions of the paper. The soundness, computational and probabilistic results are proved in Theorems 1, 2 and 3, respectively. The gray area corresponds to the attacker that is capable of observing divergence/abnormal termination.

2 Semantics, Attacker Model and Noninterference

In this section we define a suitable definition of termination-insensitive noninterference (TINI) which we believe suitably captures the intentions of Denning-style analyses, and generalises the batch-job definitions.

Computation model. We use a model of stateful computation represented as a labelled transition system consisting of *commands* $(C, C' \ldots)$ together with a *memory* $(M, M' \ldots)$ performing computations which produce low observable outputs. Since noninterference only constrains low outputs we simply do not model high outputs.

For simplicity we also assume that a memory is simply a pair consisting of a *low* (public) and a *high* (secret) value. We write such a memory M as a pair LH where L denotes the low part of the memory and H the high part. We also refer to the respective variables as L and H. We write $\langle C, M\rangle \xrightarrow{\ell} \langle C', M'\rangle$ to denote a computation step producing a low observable output ℓ and evolving to $\langle C', M'\rangle$. We write $\langle C, M\rangle \xrightarrow{\vec{\ell}} \langle C', M'\rangle$ in the usual way to denote the existence of a sequence of transitions $\langle C, M\rangle \xrightarrow{\ell_1} \langle C_1, M_1\rangle \xrightarrow{\ell_2} \cdots \xrightarrow{\ell_n} \langle C_n, M_n\rangle$ where $\vec{\ell} = \ell_1, \ldots, \ell_n$, and $\langle C, M\rangle \xrightarrow{\vec{\ell}}$

to mean $\exists\langle C', M'\rangle.\langle C, M\rangle \xrightarrow{\vec{\ell}} \langle C', M'\rangle$. We write $\langle C, M\rangle\Uparrow$ to mean that $\langle C, M\rangle$ has no labelled transitions. Note that we do not explicitly model normal termination, or distinguish stuck configurations from divergence. This is without loss of generality since observation of termination can be modelled easily by adding specific termination outputs at the end of each command. We write $\langle C, M\rangle \xrightarrow{\vec{\ell}\Uparrow}$ to mean $\langle C, M\rangle \xrightarrow{\vec{\ell}} \langle C', M'\rangle$ for some $\langle C', M'\rangle$ such that $\langle C', M'\rangle\Uparrow$. Let α range over either ℓ or the symbol $\Uparrow$, and let $\vec{\alpha}$ range over sequences of the form $\vec{\ell}$ or $\vec{\ell}\Uparrow$. We write $\vec{\ell}\ell$ to denote the sequence $\vec{\ell}$ followed by the single output ℓ.

We henceforth assume a *deterministic* labelled transition system, i.e., if $\langle C, M\rangle \xrightarrow{\ell} \langle C, M\rangle$ and $\langle C, M\rangle \xrightarrow{\ell'} \langle C', M'\rangle$ then $\ell = \ell'$ and $\langle C, M\rangle = \langle C', M'\rangle$.

On modelling divergence. For the purposes of this paper, we assume an attacker who can observe divergence. We take the view that there is a natural boundary between observing a program's timing behaviour and supposing that the attacker cannot even recognise divergence (what is such an attacker assumed to do: wait forever?).

We make a critical distinction between termination-(in)sensitivity in the attacker model vs. termination-(in)sensitivity in the security condition. We observe that the two are sometimes conflated in the literature. But unobservable divergence does not automatically make a security definition termination-insensitive. For example, by forcing all processes to diverge Huisman et al. [HWS06] achieve a form of termination-insensitivity in the attacker model, but their noninterference condition is not termination-insensitive in the traditional sense (despite the claims in the paper): it disallows programs like `(while (H=0)do skip); L:=1`.

Beyond batch-job noninterference. As we mentioned in the introduction, BTINI is an inappropriate notion for programs which actually produce observable outputs even though they do not terminate.

To define a more appropriate generalisation of batch-job termination-insensitive noninterference we model the *knowledge* gained by an attacker who (i) knows the initial low part of the memory, and (ii) observes some (not necessarily maximal) output trace $\vec{\ell}$, and (iii) knows the program and is able to make perfect deductions about the semantic behaviour of the program.

Definition 2 (Observations). *Given a program C and a choice of low input L, the set of possible observation of a run of the program is defined:*

$$Obs(C, L) = \{\vec{\alpha} \mid \langle C, LH\rangle \xrightarrow{\vec{\alpha}}\}$$

Definition 3 (Attacker's knowledge). *The attacker's knowledge from observing $\vec{\alpha}$ from a run of a program C with initial low memory L, written $k(C, L, \vec{\alpha})$, is defined to be the set of all possible high memories that could have lead to that observation:*

$$k(C, L, \vec{\alpha}) = \{H | \langle C, LH\rangle \xrightarrow{\vec{\alpha}}\}$$

This is based on the notion of knowledge defined in [AS07]. We include the possibility that the attacker explicitly observes divergence, and this will be used as a worst-case assumption in the following sections.

Figure 2 illustrates how attacker's knowledge changes with the observation of successive low outputs. The smaller the knowledge set, the more the attacker knows. In the extreme case a singleton set represents complete knowledge of the high memory. The empty set represents inconsistency – an impossible observation. Knowledge is also monotonic – the more you see the more you learn:

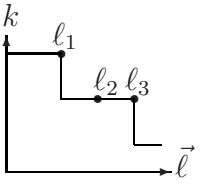


Fig. 2. Change of knowledge with low outputs

$$k(C, L, \vec{\ell}\alpha) \subseteq k(C, L, \vec{\ell})$$

From this notion of knowledge we can build various notions of noninterference. The strong termination-sensitive notion corresponds to the demand that at each step of output the attacker learns nothing new about the initial high memory. This can be formulated in the following way:

Definition 4 (Termination-sensitive noninterference). *C satisfies termination-sensitive noninterference if whenever $\vec{\ell}\alpha \in Obs(C, L)$ then $k(C, L, \vec{\ell}\alpha) = k(C, L, \vec{\ell})$.*

It perhaps looks nonstandard in this definition to include the explicit observations of divergence. In fact in this deterministic setting it turns out to make no difference to the definition if we restrict the α to $\alpha \neq \Uparrow$. In a nondeterministic setting there are subtle differences as to whether one explicitly observes divergence or not (cf. [JL00]), but this is not the concern of the present paper.

To define termination-insensitive noninterference we must relax the requirement that nothing new is learned at each step. We allow leaks that would arise from observing divergence. In the case of an output step, the idea is to permit some new knowledge when observing the next output ℓ, but only through the fact that there *is* some output. However nothing should be learned from the actual value which is output – observing one value teaches us as much as observing any other value.

Definition 5 (Termination-insensitive noninterference (TINI)). *Program C satisfies TINI if whenever $\vec{\ell}\ell \in Obs(C, L)$ then $k(C, L, \vec{\ell}\ell) = \bigcup_{\ell'} k(C, L, \vec{\ell}\ell')$.*

The term $\bigcup_{\ell'} k(C, L, \vec{\ell}\ell')$ deserves some extra attention. In terms of knowledge (as represented by sets of possible memories), union corresponds to disjunction of knowledge. More directly, this union can be defined as $\{H \mid \langle C, LH\rangle \xrightarrow{\vec{\ell}\ell'}$, for some ℓ' $\}$.

Note that, in the definition, $\ell, \ell' \neq \Uparrow$: the definition intentionally places no restrictions on what might be learned if an attacker were able to observe divergence.

The following proposition captures a number of equivalent formulations of TINI. For example, 1(2) says that TINI is equivalent to saying that what is learned from observing a specific run $\vec{\ell}$ is no more that what is learned by knowing that there *exists* a run of that length.

Proposition 1. *The following properties of a program C are equivalent to TINI:*

1. *For all L, if $\vec{\ell}\ell \in Obs(C, L)$ and $\vec{\ell}\ell' \in Obs(C, L)$ then $k(C, L, \vec{\ell}\ell) = k(C, L, \vec{\ell}\ell')$*
2. *For all L, if $\vec{\ell} \in Obs(C, L)$ then $k(C, L, \vec{\ell}) = \bigcup_{|\vec{\ell}|=|\vec{\ell}'|} k(C, L, \vec{\ell}')$*
3. *For all L, if $\langle C, LH\rangle \xrightarrow{\vec{\ell}}$ then for all H' either (i) $\langle C, LH'\rangle \xrightarrow{\vec{\ell}}$, or (ii) $\langle C, LH'\rangle \xrightarrow{\vec{\ell}'\Uparrow}$ where $\vec{\ell}'$ is a prefix of $\vec{\ell}$.*
4. *For all L, if $\vec{\ell}\ell \in Obs(C, L)$ and $\vec{\ell}\ell' \in Obs(C, L)$ then $\ell = \ell'$*
5. *For all L, the set $\{\vec{\ell} \mid \langle C, LH\rangle \xrightarrow{\vec{\ell}}\}$ forms a chain under the prefix ordering.*

The first two variants are simple consequences of the definition. The third corresponds to a more classic "two run" style definition; The last two characterisations, unlike the earlier ones, rely crucially on the assumption that computation is deterministic.

TINI subsumes BTINI. It is easy to see from this proposition that TINI generalises BTINI by considering a batch-job program to be one which performs at most one output, at the point of termination. This means that for such programs C and a given L, $\{\vec{\ell} \mid \langle C, LH\rangle \xrightarrow{\vec{\ell}}\}$ contains at most a single trace of one output, and hence for any two runs which terminate they must produce the same output.

3 Enforcement

We show that a simple Denning-style static analysis (which is at the heart of both Jif [MZZ+08] and FlowCaml [Sim03]) for a language with outputs does indeed enforce termination-insensitive noninterference.

Consider a simple imperative language with an $\texttt{output}(e)$ primitive that outputs the value of e on a low channel. The semantics of the language builds on standard small-step semantics and forms a labelled transition system, as described in Section 2. The most interesting semantic rule is the one for output:

$$\frac{e(LH) = v}{\langle \texttt{output}(e), LH\rangle \xrightarrow{v} \langle stop, LH\rangle}$$

Provided expression e evaluates to v in memory LH, the configuration $\langle \texttt{output}(e), LH\rangle$ makes a step with low-observable event v to a configuration with a halting command *stop* and unchanged memory.

Figure 3 displays the type-based enforcement rules. The rules draw on those of Volpano et al. [VSI96]. Typing environment Γ is defined as $\Gamma(L) = low$ and $\Gamma(H) = high$. Typing judgement for expressions has the form $\vdash e : \ell$. Expression e is typed as low $\vdash e : low$ only if no high variables occur in e. Typing judgement for commands has the form $pc \vdash c$, where pc is the *program counter* that keeps track of the context. Explicit flows (as in `L:=H`) are prevented by the typing rule for assignment that disallows assignments of high expressions to low variables. Implicit flows (as in `if ( H=0)then L:=0 else L:=1`) are prevented by the pc mechanism. It demands that when branching on a high expression, the branches must be typed under high pc, which

$$\vdash n : \ell \qquad \frac{\Gamma(x) = \ell \qquad \ell \sqsubseteq \ell'}{\vdash x : \ell'} \qquad \frac{\vdash e : \ell \qquad \vdash e' : \ell}{\vdash e \; op \; e' : \ell}$$

$$pc \vdash \texttt{skip} \qquad \frac{\vdash e : \ell \qquad \ell \sqcup pc \sqsubseteq \Gamma(x)}{pc \vdash x := e} \qquad \frac{pc \vdash C_1 \qquad pc \vdash C_2}{pc \vdash C_1; C_2}$$

$$\frac{\vdash e : \ell \qquad \ell \sqcup pc \vdash C_1 \qquad \ell \sqcup pc \vdash C_2}{pc \vdash \texttt{if}\ e\ \texttt{then}\ C_1\ \texttt{else}\ C_2} \qquad \frac{\vdash e : \ell \qquad \ell \sqcup pc \vdash C}{pc \vdash \texttt{while}\ e\ \texttt{do}\ C} \qquad \frac{\vdash e : low}{low \vdash \texttt{output}(e)}$$

Fig. 3. Typing rules

prevents assignments to low variables in the branches. The rule for output is a natural extension of the rules by Volpano et al. It has the same constraints on the expression and context as in the rule for assigning to a low variable.

We prove that the type system indeed guarantees termination-insensitive noninterference (TINI).

Theorem 1. *If $pc \vdash C$ then C satisfies termination-insensitive noninterference.*

According to the definition of TINI, whenever $\vec{\ell\ell} \in Obs_N(C, L)$, we need to prove $k(C, L, \vec{\ell\ell}) = \bigcup_{\ell'} k(C, L, \vec{\ell\ell'})$. The inclusion $k(C, L, \vec{\ell\ell}) \supseteq \bigcup_{\ell'} k(C, L, \vec{\ell\ell'})$ is more interesting, because $k(C, L, \vec{\ell\ell}) \subseteq \bigcup_{\ell'} k(C, L, \vec{\ell\ell'})$ is vacuous. We prove the former inclusion by induction on the length $|\vec{\ell}|$ of the sequence of low events $\vec{\ell}$ generated by C. A key property that we use in the proof is stated in the following lemma:

Lemma 1. *Suppose we have the following computation sequence starting with a configuration $\langle C_0, L_0H_0\rangle$:*

$$\langle C_i, L_iH_i\rangle \overset{\ell_{i+1}}{\rightarrow} \langle C_{i+1}, L_{i+1}H_{i+1}\rangle, \quad i \in \{0 \ldots n-1\}.$$

If C_0 is typable, and $H_0' \in k(C_0, L_0, \ell_1 \ldots \ell_n)$ then there exist $H_1', \ldots, H_n'$ such that $\langle C_i, L_iH_i'\rangle \overset{\ell_{i+1}}{\rightarrow} \langle C_{i+1}, L_{i+1}H_{i+1}'\rangle, i \in \{0 \ldots n-1\}$.

The lemma guarantees that if a typable program generates a sequence of events from some initial memory, then traces that produce the same sequence from other low-equivalent initial memories have to agree on commands in configurations that follow each low event.

4 Computational Security Implication

The type system of the previous section verifies that program 2 from the introduction is TINI. Our aim now is to show that this program is in some sense as bad as it gets – the only way for a TINI program to *reliably* leak its secret – given that the attacker can only observe a single run – is to take a non polynomial amount of time in the size of the secret.

A refined attacker model We begin by refining our attacker model. The refinement is to include a notion of time – which represents a bound on the length of the output sequences that an attacker will observe. As is usual we express results in terms of the size of the secret, N, and this is threaded through our definitions accordingly.

Definition 6 (Bounded Observations).

$$Obs_N(C, L) = \{\vec{\alpha} \mid \langle C, LH\rangle \xrightarrow{\vec{\alpha}}, 0 \leq H < 2^N\}$$

Definition 7 (Attacker knowledge (bounded version)). *The attacker's knowledge from observing $\vec{\alpha}$ by program C with initial low memory L, written $k_N(C, L, \vec{\ell})$, is defined to be the set of all high memories up to size N that could have lead to that observation:*

$$k_N(C, L, \vec{\alpha}) = \{H | H \in k(C, L, \vec{\alpha}), 0 \leq H < 2^N\}$$

The bounded version of attacker knowledge k_N differs from the knowledge k simply in that the size of the domain of H is bounded (and known to the attacker).

Definition 8 (TINI (bounded version)). *Program C satisfies TINI if for all N, whenever $\vec{\ell\ell} \in Obs_N(C, L)$ then $k_N(C, L, \vec{\ell\ell}) = \bigcup_{\ell'} k_N(C, L, \vec{\ell\ell'})$.*

The only difference from the earlier definition is that the domain of secrets is bounded and known to the attacker – but we quantify over all such bounds. Then we have

Lemma 2. *A program C satisfies TINI (bounded version) if and only if it satisfies TINI.*

Proof. We prove the left to right implication – the proof for the other direction is a simpler variant. We prove the contrapositive. Suppose C does not satisfy TINI. Then from proposition 1(1) there must exist two different observations $\vec{\ell\ell}$ and $\vec{\ell\ell'}$ in $Obs_N(C, L)$ for some L which yield different knowledge sets. Let H be a witness to this difference. Without loss of generality, assume $H \in k(C, L, \vec{\ell\ell})$ and $H \notin k(C, L, \vec{\ell\ell'})$. Now take any N such that $H < 2^N$. Clearly $H \in k_N(C, L, \vec{\ell\ell})$ and $H \notin k_N(C, L, \vec{\ell\ell'})$ and hence C is not TINI for bound N. □

Reliable leakage. We let the attacker be a pair of families $(\{L_n\}_{n\geq 0}, \{t_n\}_{n\geq 0})$, indexed over natural numbers $n \in \mathbb{N}$, where for any given natural N, L_N is a low memory that the attacker chooses based on the size of the secret N, and t_N – the attacker's running time – is the maximum time during which the attacker observes a run for secrets of that size. We will henceforth write $\{L_n, t_n\}$ as an abbreviation for $(\{L_n\}_{n\geq 0}, \{t_n\}_{n\geq 0})$.

A program leaks reliably for an attacker if he is guaranteed to learn the secret by observing a single run of the system.

Definition 9 (Reliable leakage). *Say that C leaks reliably for an attacker $\{L_n, t_n\}$ if, for each choice of N, and $H \in \{0, \ldots, 2^N - 1\}$ there is some $\vec{\alpha} \in Obs_N(C, L_N)$ such that $|\vec{\alpha}| \leq t_N$ and $k_N(C, L_N, \vec{\alpha}) = \{H\}$. Say that C leaks reliably within running time $\{t_n\}$ if there exists an attacker with that running time for which C leaks reliably.*

For example, `for i = 0 to H (output i)` leaks reliably within running time 2^n, and `output H` leaks reliably within running time $\lambda n.1$.

We can now state the main theorem of this section:

Theorem 2. *If C is TINI then C does not leak reliably within any polynomial running time.*

To prove the theorem we introduce the notion of knowledge trees.

Knowledge tree Given a program C and an attacker $\{L_n, t_n\}$ the set of possible observations that the attacker can make within the running time t_N is

$$T_N = \{\vec{\alpha} \in Obs_N(C, L_N) \mid |\vec{\alpha}| \leq t_N\}$$

This set is non-empty and prefix-closed. As is standard, such a set defines a tree. This tree is finite (finite height: $|\vec{\alpha}| \leq t_N$; finite branching: finite set of possible inputs $0 \leq H < 2^N$ plus determinism).

Definition 10 (Knowledge tree). *The* Knowledge tree *for N is the tree defined by T_N with each node $\vec{\alpha}$ labeled by its knowledge set $k_N(C, L, \vec{\alpha})$.*

We look at how knowledge trees look for N when $t_N = 2$. (These simple examples do not use L so it is not necessary to specify L_N.)

Example 1. Consider the program

```
for i = 1 to N (
    output (H mod 2) on public_channel
    H := H div 2
)
```

This program leaks the N least significant bits of H. The knowledge tree for N when $t_N = 2$ is presented in Figure 4(a). Here, annotations on the edges of the tree correspond to outputs observed by the attacker. K_0 and K_1 are knowledge sets of the form

$$K_a = \{H \mid 0 \leq H < 2^N, \text{ the least significant bit of } H \text{ is } a\} \text{ for } a \in \{0, 1\},$$

K_{00}, K_{01}, K_{10}, and K_{11} are sets of the form

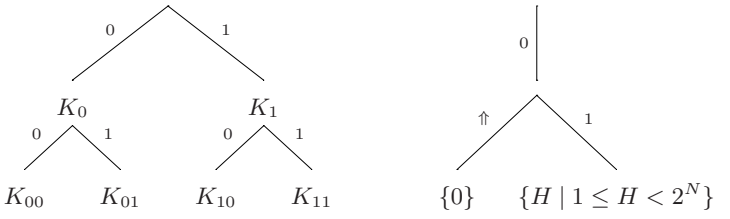
$$K_{ab} = \{H \mid 0 \leq H < 2^N, \text{ the two least significant bits of } H \text{ are } ab\} \text{ for } a, b \in \{0, 1\}.$$

Example 2. Consider now the program

```
for i=0 to 2^N-1  (
   output i on public_channel
   if (i = H) then
      (while true do skip)
)
```

The knowledge tree for this program when $t_N = 2$ is shown in Figure 4(b). As in the previous example, the annotations on the edges correspond to the attacker's observations. Thus, if the attacker observes divergence after the first output, then the knowledge about H immediately reduces to the singleton set $\{0\}$. On the other hand, observing 1 as the result of the second output only shrinks the size of the knowledge set by one.

We are now ready to formulate some properties of knowledge trees.



(a) Example 1 (b) Example 2

Fig. 4. Example knowledge trees

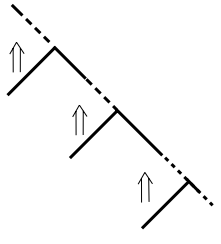
Lemma 3 (Disjointness). *Given a program C and attacker $\{L_n, t_n\}$, let $\vec{\ell} \in T_N$ be an internal node in the knowledge tree with children $\vec{\ell}\alpha_1, \ldots \vec{\ell}\alpha_n$. Let K be the knowledge set for $\vec{\ell}$ and let K_i be that for child $i, 1 \leq i \leq n$. Then the K_i are pairwise disjoint.*

Proof. Suppose $H \in K_i$ and $H \in K_j$, thus $\langle C, L_N H\rangle \xrightarrow{\vec{\ell}\alpha_i}$ and $\langle C, L_N H\rangle \xrightarrow{\vec{\ell}\alpha_j}$. Since C is deterministic, $\alpha_i = \alpha_j$.

□

The following proposition says that for programs satisfying TINI knowledge trees have a specific form.

Proposition 1 *If C satisfies TINI then for all choices of L_N, t_N, the knowledge tree has the form:*



More formally, for any $\vec{\ell} \in Obs_N(C, L_N)$, let $\{\vec{\ell}\alpha_1, \ldots, \vec{\ell}\alpha_n\}$ be the set $\{\vec{\ell}\alpha \mid \vec{\ell}\alpha \in Obs_N(C, L_N)\}$. If $n > 1$ then $n = 2$ and exactly one of α_1, α_2 is $\Uparrow$.

Proof. Suppose $\alpha_i \neq \Uparrow$ and $\alpha_j \neq \Uparrow$. Then since C satisfies TINI we have

$$k_N(C, L, \vec{\ell}\alpha_i) = \bigcup_{\ell'} \{k_N(C, L, \vec{\ell}\ell')\} = k_N(C, L, \vec{\ell}\alpha_j)$$

By the Disjointness Lemma $i = j$. Hence at most one $\alpha_i \neq \Uparrow$. □

This brings us to the proof of Theorem 2.

Proof. By definition, the height of the knowledge tree for N is t_N. If C leaks reliably then the knowledge tree contains (at least) 2^N distinct nodes, each labeled $\{H\}$ for some $0 \leq H < 2^N$. Without loss of generality we may assume that each of these singleton labels occurs on a leaf (otherwise we can prune the tree, thus choosing a shorter running time). By Proposition 1 there are at most $t_N + 1$ leaves, hence $t_N \geq 2^N - 1$. □

5 Probabilistic Security Implication

The notion of reliable leakage in the previous section is quite strong – it requires that there is never a single case when the attacker cannot deduce the exact value of the secret. To obtain a result which says something about a wider class of programs we consider the case when the attacker does not necessarily learn all the secret all the time, and hence must guess.

In this section we show that, for programs satisfying TINI, if the secrets are chosen according to a uniform distribution, then the advantage that an attacker gains by guessing the secret based on a particular observation of the computation is negligible[2] (as a function of N).

Suppose secrets $0 \leq H < 2^N$ are chosen with probability $\mu(H)$. Let C be a program and let (L_n, t_n) be an attacker. To guess a secret the attacker observes a computation and hence deduces a knowledge set. For any H, let $\vec{\alpha} \in T_N$ be the observation which the attacker uses as a basis to guess the value of H. Since knowledge is monotonic – the more an attacker observes the smaller the knowledge set – we may safely assume the attacker chooses $\vec{\alpha}$ to be the longest $\vec{\alpha} \in T_N$ such that $\langle C, L_N H\rangle \xrightarrow{\vec{\alpha}}$, i.e. $\vec{\alpha}$ is a leaf in knowledge tree (put another way, the attacker gets most information by waiting until $\vec{\alpha}$ with length t_N is produced or $\Uparrow$ is detected). Given a leaf $\vec{\alpha} \in T_N$ let the knowledge associated with $\vec{\alpha}$ be $K_{\vec{\alpha}} = k_N(C, L_N, \vec{\alpha})$. Given this, how can an attacker best guess the secret? The attacker can do no better than to choose from those elements of $k_N(C, L_N, \vec{\alpha})$ which have maximal probability according to μ. There is no disadvantage for the attacker to choose among these deterministically, so let us assume that the guess is given by a function $g_N(\vec{\alpha}) \in K_{\vec{\alpha}}$.

Now, the probability that $\vec{\alpha}$ is observed is just the sum of probabilities of all secrets H such that $\langle C, L_N, H\rangle \xrightarrow{\vec{\alpha}}$, i.e.:

$$\mu(K_{\vec{\alpha}}) = \sum_{H' \in K_{\vec{\alpha}}} \mu(H')$$

Given that $\vec{\alpha}$ is observed, the probability that the secret is $H \in K_{\vec{\alpha}}$ is $\frac{\mu(H)}{\mu(K_{\vec{\alpha}})}$. Let $Leaf \subseteq T_N$ be the set of all leaves in the knowledge tree. Then the probability that the attacker guesses the secret is

$$G_N = \sum_{\vec{\alpha} \in Leaf} \mu(K_{\vec{\alpha}}) \times \frac{\mu(g_N(\vec{\alpha}))}{\mu(K_{\vec{\alpha}})} = \sum_{\vec{\alpha} \in Leaf} \mu(g_N(\vec{\alpha}))$$

For uniformly distributed secrets, define the *attacker advantage* to be $G_N - 1/2^N$ i.e., the difference between the probability of guessing the secret based on the knowledge gained from observing a run, G_N, and a "blind" guess of the secret (which has probability $1/2^N$).

Theorem 3. *If C satisfies TINI, and secrets are chosen according to a uniform distribution, then the advantage for any polynomially-bounded attacker is negligible.*

[2] A negligible function is one that approaches zero faster than the reciprocal of any polynomial.

Proof. Since μ is uniform then regardless of g_N, $\mu(g_N(\vec{\alpha})) = 1/2^N$. Thus, in this case, the probability G_N that the attacker guesses the secret is no better than

$$\sum_{\vec{\alpha} \in Leaf} \frac{1}{2^N} = \frac{|Leaf|}{2^N}$$

We have $|Leaf| \leq t_N + 1$. Thus, $G_N \leq \frac{t_N+1}{2^N}$, and hence the attacker advantage is $\leq t_N/2^N$. From the assumption that t_N is polynomial in N, and the fact that the product of a polynomial (t_N) and a negligible function ($1/2^N$) is negligible, the attacker advantage is negligible. □

6 Practical Implications

As mentioned in Section 1 existing practical security-typed languages are based on Denning-style analysis and as a result they accept programs like program 2 from Section 1. We have encoded this program in Jif, FlowCaml and SPARK Ada to get a rough estimate on the bandwidth that such an attack creates in the worst case.

Leaking by termination in FlowCaml. Listing 1 presents encoding of program 2 from Section 1 in the FlowCaml security-typed language [Sim03].

```
flow !low < !stdout and !stdin  < !high
let maxInt : !low int  = 1000000000

let _ = let secret : !high int =
          try read_int() with _ -> 0
        in
          for i = 1 to maxInt do
             begin
               print_int i; print_newline();
               if i = secret then
                  while true do () done
             end
          done
```

Listing 1. Leaking by termination in FlowCaml

Leaking by crashing in FlowCaml. Similarly to divergence, one may also exploit program crashing. In the example above we may force a stack overflow by replacing the **if** statement with the following snippet:

```
if i = secret then
   let rec crash x = let _ = crash x in crash x
   in crash 1
```

Leaking secrets in Jif and SPARK Ada. Examples similar to the one above can be constructed for Jif security-typed language and SPARK Ada. The listings of the corresponding programs are given in the full version of this paper [AHSS08].

Channel capacity. To get a rough estimate of how practical such an attack can be we have performed a small experiment. For this, we have modified the program in Listing 1 in order to reduce the overhead related to printing on standard output. Instead, the output has been replaced by a call to a function which has the same security annotations as `print_int`, but instead of printing only saves the last provided value in a shared memory location. Observation of divergence is implemented as a separate polling process. This process periodically checks if the value in the shared location has changed since the last check. If the value is not changed this process decides that the target program has diverged.

While the time needed to reliably leak the secret is exponential in the number of secret bits, the *rate* at which this leakage happens also depends on the representation of the secret, in particular, the time needed to check two values for equivalence. Not surprisingly, the highest rate we observe is when secret is just an integer variable that fits into a computer word. In this case a 32-bit secret integer can be leaked in under 6 seconds on a machine with 3GHz CPU. Assuming this worst-case rate we may estimate time needed to leak a credit card number, typically containing 15 significant digits (50 bits), as approximately 18 1/2 days of running time. For larger secrets like encryption keys, that are usually at least 128 bits in their size, such brute-force attacks are obviously infeasible.

7 Conclusion

We have argued that in the presence of output, justifications of Denning-style analyses based on claims that they leak "just a bit" are at best misleading. We have presented the first careful analysis of termination-insensitive noninterference – the semantic condition at the heart of many information flow analysis tools and numerous research papers based on Denning's approach to analysing information flow properties of programs.

We have proposed a termination-insensitive noninterference definition that is suitable to reason about output. This definition generalizes "batch-job" style notions of termination-insensitive noninterference. The definition is tractable, in the sense that permissive Denning-style analyses enforce it. Although termination-insensitive noninterference leaks more than just a bit, we have shown that for secrets that can be made appropriately large, (i) it is not possible to leak the secret *reliably* in polynomial running time in the size of the secret; and (ii) the advantage the attacker gains when guessing the value of a uniformly distributed secret in polynomial running time is negligible in the size of the secret.

Not only is our formulation of TINI attractive for Denning-style static certification, but also for dynamic information-flow analyses. Moreover, reasoning about security based on single runs is particularly suitable for run-time monitoring approaches. Ongoing work [AS08] extends TINI with powerful declassification policies and proposes high-precision hybrid enforcement techniques that provably enforce these policies for a language with dynamic code evaluation.

Related work. The only paper of which we are aware that attempts to quantify termination leaks in Denning-style analyses is recent work of Smith and Alpízar [SA07,Smi08].

Their work takes a less general angle of attack than ours since it is (i) specific to a particular language (a probabilistic while language) and (ii) specific to a particular Denning-style program analysis. Furthermore, it uses a batch-processing model (no intermediate outputs). In their setting, the probability of divergence is shown to place a quantitative bound on the extent to which a program satisfying Denning-style conditions can deviate from probabilistic noninterference (intuitively, well-typed programs which almost always terminate are almost noninterfering). By contrast, being based on a semantic security property (TINI), our definitions and results are not language-specific, we consider deterministic systems, and the probability of divergence plays no direct role in our definitions or results. Moreover, the metric we consider is the guessing advantage afforded by termination leaks, which is not analysed in their work (we note that guessing advantage is considered in Section 2 of [Smi08] but not in the context of termination leaks).

Acknowledgements. This work was supported by EPSRC research grant EP/C009746/1 Quantitative Information Flow, the Swedish research agencies SSF, Vinnova, VR and by the Information Society Technologies programme of the European Commission under the IST-2005-015905 MOBIUS project.

References

[AHS06] Askarov, A., Hedin, D., Sabelfeld, A.: Cryptographically-masked flows. In: Proc. Symp. on Static Analysis, August 2006. LNCS, pp. 353–369. Springer, Heidelberg (2006)

[AHSS08] Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. Technical report, Chalmers University of Technology (July 2008), `http://www.cs.chalmers.se/~aaskarov/esorics08full.pdf`

[AS07] Askarov, A., Sabelfeld, A.: Gradual release: Unifying declassification, encryption and key release policies. In: Proc. IEEE Symp. on Security and Privacy, May 2007, pp. 207–221 (2007)

[AS08] Askarov, A., Sabelfeld, A.: Tight enforcement of flexible information-release policies for dynamic languages. Draft (July 2008)

[BB03] Barnes, J., Barnes, J.G.: High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)

[BNR08] Banerjee, A., Naumann, D., Rosenberg, S.: Expressive declassification policies and modular static enforcement. In: Proc. IEEE Symp. on Security and Privacy, May 2008, pp. 339–353 (2008)

[CH04] Chapman, R., Hilton, A.: Enforcing security and safety models with an information flow analysis tool. ACM SIGAda Ada Letters 24(4), 39–46 (2004)

[DD77] Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Comm. of the ACM 20(7), 504–513 (1977)

[Fen74] Fenton, J.S.: Memoryless subsystems. Computing J. 17(2), 143–147 (1974)

[HWS06] Huisman, M., Worah, P., Sunesen, K.: A temporal logic characterisation of observational determinism. In: Proc. IEEE Computer Security Foundations Workshop (July 2006)

[JL00] Joshi, R., Leino, K.R.M.: A semantic approach to secure information flow. Science of Computer Programming 37(1–3), 113–138 (2000)

[LBJS08] Le Guernic, G., Banerjee, A., Jensen, T., Schmidt, D.: Automata-based confidentiality monitoring. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 75–89. Springer, Heidelberg (2008)

[Mye99] Myers, A.C.: JFlow: Practical mostly-static information flow control. In: Proc. ACM Symp. on Principles of Programming Languages, January 1999, pp. 228–241 (1999)

[MZZ+08] Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif: Java information flow. Software release (July 2001–2008), http://www.cs.cornell.edu/jif

[SA07] Smith, G., Alpízar, R.: Fast probabilistic simulation, nontermination, and secure information flow. In: PLAS 2007: Proceedings of the 2007 workshop on Programming languages and analysis for security, pp. 67–72. ACM, New York (2007)

[Sim03] Simonet, V.: The Flow Caml system. Software release (July 2003), http://cristal.inria.fr/~simonet/soft/flowcaml/

[Smi08] Smith, G.: Adversaries and information leaks. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 383–400. Springer, Heidelberg (2008)

[SS99] Sabelfeld, A., Sands, D.: A per model of secure information flow in sequential programs. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 40–58. Springer, Heidelberg (1999)

[VS97] Volpano, D., Smith, G.: Eliminating covert flows with minimum typings. In: Proc. IEEE Computer Security Foundations Workshop, June 1997, pp. 156–168 (1997)

[VSI96] Volpano, D., Smith, G., Irvine, C.: A sound type system for secure flow analysis. J. Computer Security 4(3), 167–187 (1996)