Verifying Secrets and Relative Secrecy

Dennis Volpano* Geoffrey Smith†

Abstract

Systems that authenticate a user based on a shared secret (such as a password or PIN) normally allow anyone to query whether the secret is a given value. For example, an ATM machine allows one to ask whether a string is the secret PIN of a (lost or stolen) ATM card. Yet such queries are prohibited in any model whose programs satisfy an information-flow property like Noninterference. But there is complexity-based justification for allowing these queries. A type system is given that provides the access control needed to prove that no well-typed program can leak secrets in polynomial time, or even leak them with nonnegligible probability if secrets are of sufficient length and randomly chosen. However, there are well-typed deterministic programs in a synchronous concurrent model capable of leaking secrets in linear time.

1 Introduction

A common approach to authenticating a user is based on a shared secret. Normally, anyone can ask whether the secret is a particular value. For instance, password-based authentication allows anyone to ask whether a certain string is the password of a given user. An ATM permits anyone to ask, for a given card, whether a string is the hidden PIN (Personal ID Number) of that card, and so on. Formalizing the secrecy guaranteed by systems that use this form of authentication is beyond the scope of information-flow techniques because their goal is absolute secrecy.

For instance, suppose h is a constant (read-only variable) whose value is a k-bit integer secret. If we represent a query as $match(e)$, which is true iff $h = e$, then a brute-force attack on h can be represented by a simple loop:

$$\begin{array}{l} l := 0; \\ \textit{while } \neg match(l) \textit{ do} \\ \quad l := l + 1 \end{array}$$

An information-flow property like Noninterference [5, 15, 12, 14] rejects the program. Given that h has security class H (high) and l has security class L (low), Noninterference requires that the final value of l be independent of the initial value of h—this plainly fails here.

But the loop is a $\Theta(2^k)$ attack, assuming the value of h is chosen at random. So as long as k is large enough and the value of h is randomly chosen, we do not consider the attack a threat and would like a secrecy criterion that reflects this view. Our approach is to adopt a relative criterion for secrecy that depends upon certain parameters, specifically, the number of bits in a secret and randomness. The idea is to admit only those programs for which it can be proved that sufficiently-long secrets cannot be learned in polynomial time or even learned with nonnegligible probability in polynomial time if randomly chosen.

The query expression $match(e)$ also represents the most basic tool any attacker has for compromising public-key cryptography. Given ciphertext c that is the result of encrypting plaintext h with a public key, an attacker can test efficiently whether h is a particular string e simply by encrypting e with the public key and seeing whether the same ciphertext c is produced. Of course the nonuniform distribution of plaintext makes cracking a public-key system with *match* queries different from a brute-force attack on a random secret. Good dictionary attacks exploit the distribution. In this case, *match* queries may not be justified from a complexity standpoint and another cryptographic system may be required.

In this paper, we consider the problem of trying to learn the k-bit integer value of a constant h of type H using well-typed programs written in a deterministic programming language with *match* queries. We require that programs be well typed in the system of [15], except that we allow the type of *match* queries to be L, which destroys traditional Noninterference properties.

Instead, we argue for the security of our language by proving that no well-typed program that is capable of deducing h runs in time bounded by a polynomial in k (the size of h). Notice that such a result appears to separate $\mathcal{P}$ and $\mathcal{NP}$ (see pg. 373 of [11]), as a well-typed nondeterministic program can guess h and verify its guess with *match* in time linear in k. But our result actually separates relativized forms of $\mathcal{P}$ and $\mathcal{NP}$ because of our use of the *match* oracle. This sort of separation was shown long ago [1].

The preceding guarantee is the best one can do in the absence of any probability distribution for the values of h. In the case where there is a distribution, we can talk about the probability of successfully learning h. We show that if the value of h is chosen with respect to a uniform probability distribution, then for any well-typed polynomial-time program c, the probability that c successfully learns h goes to zero as k increases.

One might conclude that equality testing is safe in general, and that we could simply give $e_1 = e_2$ type L, regard-

*Computer Science Department, Naval Postgraduate School, Monterey, California 93943. Email: volpano@cs.nps.navy.mil.

†School of Computer Science, Florida International University, Miami, Florida 33199. Email: smithg@cs.fiu.edu.

less of the types of e_1 and e_2.[2] However, this is not so; it then becomes possible to leak a secret in linear time.

Finally, we consider the effect of adding concurrency to the language. Unlike our previous work [12, 14], we consider a synchronous form of concurrency, so that programs remain deterministic. We show that synchronous concurrency, even without *match* queries, allows well-typed programs to leak secrets in linear time.

2 The deterministic language

In this paper, we consider a deterministic imperative programming language with a query primitive *match*:

$$\begin{array}{lrcl} (expr) & e & ::= & x \mid n \mid h \mid match(e) \mid e_1 + e_2 \mid \\ & & & e_1 < e_2 \mid e_1 = e_2 \mid e_1 \neq e_2 \\ (cmds) & c & ::= & \textbf{skip} \mid x := e \mid c_1; c_2 \mid \\ & & & \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \\ & & & \textbf{while } e \textbf{ do } c \end{array}$$

Metavariable x ranges over identifiers that are mapped by memories to integers; n ranges over integer literals. Integers are the only values; we use 0 for false and nonzero for true. The special identifier h is a read-only variable whose binding we assume is secret.

A standard transition semantics for the language is given in Figure 1. A *memory* μ is a mapping from identifiers to integers. We assume that expressions are evaluated atomically. Thus we simply extend a memory in the obvious way to map expressions to integers, writing $\mu(e)$ to denote the value of expression e in memory μ. We say that $\mu(match(e)) = 1$ iff $\mu(h) = \mu(e)$; otherwise $\mu(match(e)) = 0$. Note that expressions do not have side effects, nor do they contain partial operations like division. Thus $\mu(e)$ is defined for all e, so long as every identifier in e is in $dom(\mu)$.

These rules define a transition relation $\longrightarrow$ on configurations. A configuration is either a pair (c, μ) or simply a memory μ. In the first case, c is the command yet to be executed; in the second case, the command has terminated, yielding final memory μ. As usual, we define $\kappa \longrightarrow^0 \kappa$, for any configuration κ, and $\kappa \longrightarrow^k \kappa''$, for $k > 0$, if there is a configuration κ' such that $\kappa \longrightarrow^{k-1} \kappa'$ and $\kappa' \longrightarrow \kappa''$.

3 Relative secrecy

We begin by looking at relative secrecy properties that can be proved for programs that access h via *match* queries only. Then, in Section 5, we give a reduction that extends these results to programs that may have free occurrences of h. The reduction requires that these programs be well typed under the type system of Section 4.

The first property is an intractability guarantee:

Theorem 3.1 *There is no deterministic command capable of copying the k-bit integer value of h into a variable l in time polynomial in k, for all k, if h is accessed via match queries only.*

Proof. Suppose c runs in polynomial time $p(k)$ where k is the number of bits needed to encode the value of h. Choose k large enough so that $2^k > p(k) + 1$. Since c can make at most $p(k)$ queries and $2^k > p(k)+1$, there are k-bit integers i and j such that $i \neq j$ and neither i nor j is queried by c. Now c must copy the value i into l when the value of h is i. But if it does, then it also copies i into l when the value of h is j, since it is deterministic and does not query j. □

Notice that if commands were nondeterministic, then a command could nondeterministically choose an integer n and then issue a query $match(n)$. If the query succeeds, it copies n into l. So we can always copy the value of h in nondeterministic polynomial time.

The preceding guarantee is the best one can do in the absence of any probability distribution on the values of h, or in other words, when its values are chosen nondeterministically. When there is a probability distribution for h, we can talk about the probability of copying h. Indeed, in this case, a command might succeed often, perhaps even most of the time, in copying h. However, one can show that if access to h is limited to *match* queries, then any deterministic, polynomial-time command for copying h to a variable l will succeed with only negligible probability for uniformly-distributed and sufficiently-large values of h.

Suppose c is a deterministic command that runs in polynomial time $p(k)$ and accesses a constant h via *match* queries only. And suppose the value of h is a k-bit integer, chosen with respect to some probability distribution d on k-bit integers. Finally, assume that k is large enough so that $2^k > p(k)$. Now, as c runs, it makes *match* queries in an attempt to learn h. If any such query gets result 1, then c can put the correct value into l and halt. But in $p(k)$ time, c can query at most $p(k)$ out of the 2^k possible values of h. Since $2^k > p(k)$, all of c's queries might get result 0. In such a run, c must eventually halt and put some value v into l. But note that since c is deterministic and accesses h only through *match* queries, c's behavior will be exactly the same if the value of h is *any* of the unqueried integers—in any such run, c will make exactly the same sequence of queries and will finally write the same default value v into l. That is, for any such c we can identify a set S of at most $p(k)$ integers that c will query, and a default value v that c will write into l if all of the queries get result 0. It follows that c will be successful in copying the value of h into l if and only if the value of h is in $S \cup \{v\}$.

Given distribution d, how can c maximize its probability of success? Clearly, the best strategy is to choose S and v so that $d(S \cup \{v\})$ is as large as possible. This can be done by choosing the $p(k)+1$ most likely values under d, putting $p(k)$ of them into S, and using the remaining value as v.

The probability of success may be high if either k is too small or the probability distribution d is severely skew. But if d is a uniform distribution, so that all possible values of h are equally likely, then for any choice of $p(k)+1$ elements of $S \cup \{v\}$ we have

$$d(S \cup \{v\}) = \frac{p(k)+1}{2^k}.$$

(Indeed, we can see that a uniform distribution minimizes c's probability of success.) Hence we have proved

Theorem 3.2 *If the value of h is a uniformly distributed k-bit integer and c is a deterministic command that runs in polynomial time $p(k)$ and accesses h via match queries only, then the probability that c successfully copies the value of h into a variable l is at most $(p(k)+1)/2^k$.*

[2] In contrast, $\leq$ is clearly dangerous, as it allows a secret to be computed by binary search.

$$\text{(NO-OP)} \qquad (\mathbf{skip}, \mu) \longrightarrow \mu$$

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

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

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

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

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

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

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

Figure 1: Transition semantics

4 The type system

Now we wish to allow programs to access h directly, rather than just via *match* queries. Of course, a program could then simply do $l := h$, so we need another mechanism to ensure that h is not leaked. For this purpose, we impose a type system.

The types are stratified into data and phrase types:

$$\begin{array}{lll} (\textit{data types}) & \tau & ::= L \mid H \\ (\textit{phrase types}) & \rho & ::= \tau \mid \tau\ \textit{var} \mid \tau\ \textit{cmd} \end{array}$$

The data types are just the security levels low and high. The rules of the type system are given in Figure 2. They just extend the system of [15] with a rule for *match*. The rules allow us to prove typing judgments of the form $\gamma \vdash p : \rho$ as well as subtyping judgments of the form $\rho_1 \subseteq \rho_2$. Here γ is a typing that maps identifiers to types of the form $\tau\ \textit{var}$, except for h which is mapped by every typing to H. The typing rules for the remaining binary operators are similar to that for EQ. Intuitively, γ classifies variables as either high or low, and the typing rules prevent information from flowing from high variables to low variables.

Notice that rule QUERY allows one to assign type L to a query even though that query is against a high constant. It is this rule that breaks Noninterference [15], for it allows some information about a secret to become public. Also note that the rule allows restricted access to a high constant in the guard of a **while** loop or conditional without requiring the body or branches to be typed as $H\ \textit{cmd}$. This provides more flexibility in programming, and indeed it now becomes possible to write a conditional that, through an indirect flow, copies a one-bit secret to a low variable. But *match* is clearly unsafe with a secret of only one bit!

5 The reduction

We reduce the problem of learning the value of h, using a deterministic program whose access to h is limited to *match* queries, to that of learning the value of h using a well-typed deterministic program (whose access to h is not limited to queries). More precisely, we show that every well-typed command's computation with respect to low variables can be simulated, with no increase in time complexity, by a command whose access to h is restricted to *match* queries. So learning a secret with any well-typed program is as hard as learning it with only *match* queries at your disposal. Hence the relative secrecy properties apply to well-typed commands. If a program is not well typed then all bets are off, as it could well be the case that a secret is easily leaked via a direct or indirect flow.

We begin with some definitions:

Definition 5.1 *Memories μ and ν are equivalent with respect to a typing γ, written $\mu \sim_\gamma \nu$, if $dom(\mu) = dom(\nu) = dom(\gamma)$ and $\mu(x) = \nu(x)$ for all x such that $\gamma(x) = L\ \textit{var}$.*

This definition requires two memories to agree on the contents of all low variables.

Definition 5.2 *We say that a command c is a low command with respect to γ if $\gamma(x) = L\ \textit{var}$ for every free identifier x in c.*[3]

[3]Note that this is not the same as saying that the command has type $L\ \textit{cmd}$, as *every* well-typed command has type $L\ \textit{cmd}$.

(INT) $\gamma \vdash n : L$

(SECRET) $\dfrac{\gamma(h) = H}{\gamma \vdash h : H}$

(R-VAL) $\dfrac{\gamma(x) = \tau\ var}{\gamma \vdash x : \tau}$

(SKIP) $\gamma \vdash \mathbf{skip} : H\ cmd$

(EQ) $\dfrac{\gamma \vdash e_1 : \tau,\ \ \gamma \vdash e_2 : \tau}{\gamma \vdash e_1 = e_2 : \tau}$

(QUERY) $\dfrac{\gamma \vdash e : L}{\gamma \vdash match(e) : L}$

(ASSIGN) $\dfrac{\gamma(x) = \tau\ var,\ \ \gamma \vdash e : \tau}{\gamma \vdash x := e : \tau\ cmd}$

(COMPOSE) $\dfrac{\gamma \vdash c_1 : \tau\ cmd,\ \ \gamma \vdash c_2 : \tau\ cmd}{\gamma \vdash c_1; c_2 : \tau\ cmd}$

(IF) $\dfrac{\gamma \vdash e : \tau,\ \ \gamma \vdash c_1 : \tau\ cmd,\ \ \gamma \vdash c_2 : \tau\ cmd}{\gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : \tau\ cmd}$

(WHILE) $\dfrac{\gamma \vdash e : \tau,\ \ \gamma \vdash c : \tau\ cmd}{\gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ c : \tau\ cmd}$

(BASE) $L \subseteq H$

(REFLEX) $\rho \subseteq \rho$

(CMD$^-$) $\dfrac{\tau_1 \subseteq \tau_2}{\tau_2\ cmd \subseteq \tau_1\ cmd}$

(SUBTYPE) $\dfrac{\gamma \vdash p : \rho_1,\ \ \rho_1 \subseteq \rho_2}{\gamma \vdash p : \rho_2}$

Figure 2: Typing and subtyping rules

Definition 5.3 *We say that a command c' is a low simulation of a command c with respect to γ if c' is a low command relative to γ and for all μ where $dom(\mu) = dom(\gamma)$, whenever $(c, \mu) \longrightarrow^n \mu'$, there is a μ'' and m such that $(c', \mu) \longrightarrow^m \mu''$, $\mu' \sim_\gamma \mu''$ and $m \leq n$.*

The reduction is given by the following theorem.

Theorem 5.1 *If c is a well-typed command with respect to γ, then there is a low simulation of c with respect to γ.*

A proof of the theorem is given in the Appendix. The essence of the proof is that commands of type H *cmd* cannot affect any low variable. Therefore, they can be replaced by the **skip** command. Commands that cannot be given type H *cmd* are retained. The result is a command that preserves the computation on low variables and runs in time bounded by that of the original command.

Since a low command is limited to accessing h via *match* queries, we have the following corollaries:

Corollary 5.2 *There is no well-typed deterministic command capable of copying the k-bit integer value of h to a low variable in time polynomial in k, for all k.*

Corollary 5.3 *If the value of h is a uniformly distributed k-bit integer and c is a well-typed deterministic command that runs in polynomial time $p(k)$, then the probability that c successfully copies the value of h into a low variable is at most $(p(k) + 1)/2^k$.*

Thus we have shown that the secrecy of h is (practically speaking) preserved even if programs are allowed to make *match* queries freely. One might conclude from this that equality testing is harmless in general. However, this is not so. If we allow general equality tests among high expressions to have type L, then we can leak h in linear time:

$$
\begin{array}{l}
l := 0; \\
mask := 2^{k-1}; \\
\textbf{while } mask \neq 0 \textbf{ do} \\
\quad b := h \mathbin{\&} mask; \\
\quad \textbf{if } b = mask \textbf{ then} \\
\quad\quad l := l \mid mask; \\
\quad mask := mask \gg 1
\end{array}
$$

6 Synchronous concurrency

Now consider a deterministic, multi-threaded semantics where all threads execute simultaneously and synchronously. Threads are commands that communicate via a shared global memory. Every thread of a multi-threaded program can make one transition, according to the semantics of Figure 1, in a given clock cycle. At each step, parallel reads are allowed but not parallel writes. The latter situation causes an evaluation to get stuck.

The synchronous behavior of the model provides well-typed commands with a timing channel that can be exploited to leak a secret perfectly in time linear in its size. Examples of multi-threaded programs capable of doing this can be found in timing attacks on implementations of cryptography [7]. Here there are bit-wise operations on a secret key and one exploits the fact that the implementation is optimized according to certain bits of the key. An example is the C implementation of the block cipher algorithm IDEA in [10] (see pg. 640). The default is to avoid multiplication when certain bits of a key are zero.

The essence of this sort of attack can be formulated within the synchronous concurrent model. To formulate it concretely, let us introduce the bitwise operators `~` (ones complement), `&` (bitwise and), `|` (bitwise or) and `>>` (right shift). Typing rules for these operators are similar to that of rule EQ. We also take the liberty to introduce a **while** loop with empty body. Now suppose h stores a k-bit secret that is being inspected bitwise by a legitimate thread that simply loops through all bits, doing some computation if a bit is nonzero and nothing otherwise. Think of the thread as an implementation of an encryption algorithm where less time is required if a bit of the k-bit secret key is zero. The inspection thread is given by the following command:

```
while mask != 0
   if h & mask then
      skip;
      skip;
      skip;
      skip
   fi
   mask := mask >> 1
```

We set up $2k$ attack threads, one pair for each bit of the secret. Each is responsible for setting a particular bit of an integer stored in low variable l which is where the secret is leaked. If $k = 8$, for instance, then for the most significant bit, we have the threads:

```
                          while mask != 128
while mask != 128            ;
   ;                      skip;
while mask == 128         skip;
   ;                      skip;
l := l | 128              skip;
                          l := l & ~128
```

There would be another identical pair for the next bit, except they would spin while waiting for *mask* to become 64, and so on. The inspection and attack threads share *mask* which is a low variable.[4] Further, all threads are well typed and every attack thread is a low command which we would expect since an attack would not have direct access to variables storing secrets anyway. We have set up a race where the thread that sets a bit of l to b wins the race iff the corresponding bit of h is $\neg b$.

To illustrate the behavior, suppose the first two bits of h are 01 and that *mask* is initially 128. The trace of the inspection thread on this input is given in the first column of Figures 3 and 4. Figure 3 shows the trace of a thread pair attacking the first bit of h while Figure 4 shows the trace of a thread pair attacking the second bit. The type system is not equipped to deal with this kind of model where timing observations can be made. Tricks like atomicity [14] do not help here because threads in effect share a real-time clock!

If one views the preceding multi-threaded program as a probabilistic algorithm then

$$\Pr[h = n \mid l = n] = 1$$

[4] Making *mask* a high variable is not a good idea here because then no assignments to low variables would be allowed in the body of the loop. This in turn would hamper an implementation of encryption.

```
while mask != 0        while mask != 128       while mask != 128
if h & mask then       while mask == 128       skip
mask := mask >> 1      while mask == 128       skip
while mask != 0        while mask == 128       skip
if h & mask then       l := l | 128            skip
skip                                           l := l & ~128
```

Figure 3: Timing attack on first bit of h

```
while mask != 0        while mask != 64        while mask != 64
if h & mask then       while mask != 64        while mask != 64
mask := mask >> 1      while mask != 64        while mask != 64
while mask != 0        while mask != 64        while mask != 64
if h & mask then       while mask == 64        skip
skip                   while mask == 64        skip
skip                   while mask == 64        skip
skip                   while mask == 64        skip
skip                   while mask == 64        l := l & ~64
mask := mask >> 1      while mask == 64
while mask != 0        while mask == 64
if h & mask then       l := l | 64
```

Figure 4: Timing attack on second bit of h

for all n.[5] In other words, one can be certain that h stores a particular secret if the algorithm says so. This is an ideal probabilistic algorithm.

But consider an interleaving semantics where at each step of executing a multi-threaded program, exactly one thread is chosen to execute for a single transition. Now multi-threaded programs are nondeterministic. If at every step, every thread has a nonzero probability of being selected by a scheduler, then $\Pr[h = n \mid l = n] < 1$, for all n, unless $\Pr[h = n] = 1$. That's because the final value of l can be any value with nonzero probability, regardless of h. Each value of *mask* will be considered, allowing the thread pair for the bit in question to execute, and these two threads can complete in either order with nonzero probability. So $\Pr[l = n \mid h \neq n] > 0$.

The example shows that there are attacks that are stronger in a synchronous concurrent model than in a probabilistic interleaving one. For each new model, one needs to investigate the feasibility of revealing secrets.

7 Discussion

It has long been recognized that practical information flow control must allow for declassifying information. An example is a simple password checker. It must indicate whether a given string matches a user's password in a password file. Information-flow control would prohibit the result of such an attempt from being observed by an arbitrary user. One way around this restriction is to allow privileged users to explicitly declassify the result, allowing some information to leak [9]. If a password checker merely returns the result of a *match* query, then the declassification could be justified knowing that any attempt to exploit the checker in order to learn a password is subject to the limitations of Theorems 3.1 and 3.2.

Note we have said nothing about the role of the type system at this point. Type checking would be performed on any program that updates the password file, since this is where new passwords are input. As long as the password checker merely returns the result of a *match* query, password updating is well typed, and passwords are stored in a secret (high) file, the password system as a whole is secure in the sense that it is subject to the limitations of Corollaries 5.2 and 5.3.

But what can we say about the security of a system in which there is a *public* file containing the images of the passwords under some one-way function? The reduction of Theorem 5.1 can also be used to argue for security in this case. Suppose we add to our deterministic language a function f that we believe is a one-way function [11] and a constant fh bound to the image of h under f. That is, $\mu(fh) = \mu(f(h))$, for any memory μ. Further, suppose they are typed as $\gamma \vdash fh : L$, and for any expression e,

$$\frac{\gamma \vdash e : \tau}{\gamma \vdash f(e) : \tau}$$

One cannot allow $f(e)$ to be typed low unless e can be typed as a low expression. Otherwise, we can copy h into a low variable l in linear time using a well-typed program similar to the one in Section 5:

$l := 0;$
$mask := 2^{k-1};$
while $mask \neq 0$ **do**
  if $f(mask) = f(h \;\&\; mask)$ **then**
    $l := l \mid mask;$
  $mask := mask \gg 1$

Notice that this program might fail to copy every bit of h due to a collision caused by f. But this would be unlikely if f is indeed a one-way function. Also note that fh makes *match* unnecessary: if $\gamma \vdash e : L$ then the expression $fh = f(e)$ can, practically speaking, be used in place of $match(e)$.

[5] When we run the algorithm and it terminates with $l = n$, we want to be able to conclude that $h = n$ with some probability. The conditional probability $\Pr[l = n \mid h = n]$ is wrong for this purpose. If $\Pr[l = n \mid h = n] = .99$, say, and we run the algorithm for some value of h and it terminates with $l = n$, then this does not imply $h = n$ with probability .99. The same argument applies to probabilistic primality testing algorithms [2].

We wish to argue for the security of any well-typed program c that uses f. Now c may have free occurrences of h and can call f according to the typing rules above. Suppose that c can copy h into a low variable, for all k-bit values of h, in time $t(k)$. By Theorem 5.1, we can construct a low simulation c' of c that does the same thing as efficiently. That is, given only fh and all calls to f that c makes involving only low variables, c' can also copy h in $t(k)$ time. In effect, c' is an algorithm for deducing h from fh using some number (as a function of k) of calls to f. So any lower bound on this problem applies to copying h with a well-typed program.

Although we suggest above that *match* could be replaced, the preceding argument does not subsume our results about the security of programs that access h using *match* only. For instance, the preceding argument about f does not prove an intractability result for programs limited to accessing h via fh, as Theorem 3.1 does for *match*. We have only argued that the hardness of copying h using a well-typed program with references to fh and calls to f rests squarely upon the hardness of inverting f.

Turing reductions have also been used in proving the security of RSA-based signature schemes [3, 4] and cryptographic protocols [6, 8]. Again the basic idea is to prove that the security of a protocol rests on the strength of its cryptographic primitives. Our synchronous concurrency example shows that if an intruder can observe the timing behavior of an implementation of a cryptographic operation, then the specification of a protocol that uses the operation cannot realistically treat it as primitive. A reduction may not exist because of timing observations, as our example shows.

8 Conclusion

Our research is aimed at characterizing secrecy in systems with *inherent* leaks, for example, those that use authentication based on shared secrets. Noninterference is too strong for such systems. Instead, we propose a relative secrecy proof that takes practical security parameters (e.g. key size and randomness) into consideration.

Acknowledgments

This material is based upon activities supported by the National Science Foundation under Agreement Nos. CCR-9612176 and CCR-9612345 [sic]. We are grateful to the anonymous referees for helpful comments.

References

[1] T. Baker, J. Gill, and R. Solovay. Relativizations of the P =? NP question. *SIAM J. Computing*, 4(4):431–442, 1975.

[2] P. Beauchemin, G. Brassard, C. Crépeau, C. Goutier, and C. Pomerance. The generation of random numbers that are provably prime. *Journal of Cryptology*, 1(1):53–64, 1988.

[3] M. Bellare and P. Rogaway. The exact security of digital signatures—how to sign with RSA and Rabin. In *Proc. Eurocrypt 96*. Lecture Notes in Computer Science 1070, 1996.

[4] M. Bellare and P. Rogaway. Practice-oriented provable security. In *Proc. of First International Workshop on Information Security*. Lecture Notes in Computer Science 1396, 1998.

[5] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings 1982 IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, 1982.

[6] J. Gray, K. Ip, and K. Lui. Provable security for cryptographic protocols—exact analysis and engineering applications. *Journal of Computer Security*, 6(1,2):23–52, 1998.

[7] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. In *Proceedings 16th Annual Crypto Conference*, August 1996.

[8] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *Proceedings 5th ACM Conference on Computer and Communications Security*, San Francisco, CA, November 1998.

[9] A. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings 26th Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, TX, January 1999.

[10] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 1996. Second Edition.

[11] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[12] G. Smith and D. 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.

[13] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Proceedings 10th IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.

[14] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, 1999.

[15] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.

9 Appendix

The proofs of Theorem 5.1 and the Simple Security and Confinement lemmas are complicated a bit by subtyping. Assume, without loss of generality, that all typing derivations end with a single (perhaps trivial) application of rule SUBTYPE.

Theorem 5.1 *If c is a well-typed command with respect to γ, then there is a low simulation of c with respect to γ.*

Proof. The proof is by induction on the structure of c.

We first consider the case when $\gamma \vdash c : H\ cmd$. In this case, we can simply let the low simulation of c be **skip**. For by the Confinement lemma below, c does not assign to any variable x for which $\gamma(x) = L\ var$. Hence if $(c, \mu) \longrightarrow^n \mu'$, then $\mu' \sim_\gamma \mu$. And we have $(\mathbf{skip}, \mu) \longrightarrow^1 \mu$, by rule NO-OP.

Finally, we note that $n \geq 1$ and that **skip** is a low command with respect to γ.

Next we consider the case when $\gamma \not\vdash c : H\ cmd$. In this case we must have $\gamma \vdash c : L\ cmd$ (since c is well typed under γ) and the derivation must end with a trivial application of rule SUBTYPE (since otherwise we would have $\gamma \vdash c : H\ cmd$). We now consider the possible forms of c:

Case **skip**. Since $\gamma \vdash \mathbf{skip} : H\ cmd$, this case has already been handled.

Case $x := e$. By the discussion above, there is a derivation of $\gamma \vdash x := e : L\ cmd$ that ends with an application of rule ASSIGN. This implies that $\gamma(x) = L\ var$ and $\gamma \vdash e : L$. So, by the Simple Security lemma below, $\gamma(y) = L\ var$ for every identifier y free in e. Therefore $x := e$ is itself a low command relative to γ, and $x := e$ is thus (trivially) a low simulation of itself.

Case **if** e **then** c_1 **else** c_2. Again by the discussion above, there is a derivation of $\gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : L\ cmd$ that ends with an application of rule IF. Hence $\gamma \vdash e : L$ and both c_1 and c_2 are well typed with respect to γ. Then by induction there exist commands c_1' and c_2' which are low simulations of c_1 and c_2, respectively, with respect to γ. We claim that **if** e **then** c_1' **else** c_2' is a low simulation of **if** e **then** c_1 **else** c_2 with respect to γ. First, note that it is a low command under γ, since (by the Simple Security lemma) $\gamma(x) = L\ var$ for every free identifier x in e. Next, suppose that μ is a memory such that $dom(\mu) = dom(\gamma)$ and $(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \mu) \longrightarrow^n \mu'$. Then if $\mu(e) \neq 0$, the evaluation has the form

$$(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \mu) \longrightarrow (c_1, \mu) \longrightarrow^{n-1} \mu'.$$

Therefore, since c_1' is a low simulation of c_1, there exists a memory μ'' and an integer m such that $(c_1', \mu) \longrightarrow^m \mu''$, $\mu' \sim_\gamma \mu''$ and $m \leq n-1$. Therefore, by rule BRANCH,

$$(\mathbf{if}\ e\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2', \mu) \longrightarrow (c_1', \mu) \longrightarrow^m \mu''.$$

And $m + 1 \leq n$. The argument is similar in the case when $\mu(e) = 0$.

Case $c_1; c_2$. As above, there is a derivation of $\gamma \vdash c_1; c_2 : L\ cmd$ that ends with an application of rule COMPOSE, implying that c_1 and c_2 are well typed under γ. So by induction there exist commands c_1' and c_2' which are low simulations of c_1 and c_2, respectively, with respect to γ. We claim that $c_1'; c_2'$ is a low simulation of $c_1; c_2$ under γ. For suppose that μ is a memory with $dom(\mu) = dom(\gamma)$ and $(c_1; c_2, \mu) \longrightarrow^n \mu'$. By Lemma 9.4, there exist k and μ'' such that $0 < k < n$, $(c_1, \mu) \longrightarrow^k \mu''$, and $(c_2, \mu'') \longrightarrow^{n-k} \mu'$. Since c_1' is a low simulation of c_1, there exist ν'' and m such that $(c_1', \mu) \longrightarrow^m \nu''$, $\mu'' \sim_\gamma \nu''$, and $m \leq k$. And since c_2' is a low simulation of c_2, there exist ν' and j such that $(c_2', \mu'') \longrightarrow^j \nu'$, $\mu' \sim_\gamma \nu'$, and $j \leq n - k$.

Now c_1 is well typed under γ and therefore h is not assigned to in c_1 because $\gamma(h) = H$ (i.e. h is not a variable). So $\mu''(h) = \mu(h)$. Further, c_1' is a low command, so $\nu''(h) = \mu(h)$. Hence $\mu''(h) = \nu''(h)$. Since $\mu'' \sim_\gamma \nu''$, $\mu''(h) = \nu''(h)$, and c_2' is a low command, we have

$$(c_2', \nu'') \longrightarrow^j \nu'.^6$$

Therefore, by Lemma 9.5, $(c_1'; c_2', \mu) \longrightarrow^{m+j} \nu'$. And $m + j \leq k + n - k = n$.

[6] Note that this conclusion does *not* follow simply from the facts that $(c_2', \mu'') \longrightarrow^j \nu'$ and $\mu'' \sim_\gamma \nu''$, because c_2' may contain *match* queries.

Case **while** e **do** c_1. As above, there is a derivation of $\gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ c_1 : L\ cmd$ ending with rule WHILE, which implies that $\gamma \vdash e : L$ and c_1 is well typed with respect to γ. By induction, there is a low simulation c_1' of c_1 with respect to γ. We claim that **while** e **do** c_1' is a low simulation of **while** e **do** c_1 with respect to γ. First, it is a low command under γ, since (by the Simple Security lemma) $\gamma(x) = L\ var$ for every free identifier x in e. Next, suppose μ is a memory with $dom(\mu) = dom(\gamma)$ and $(\mathbf{while}\ e\ \mathbf{do}\ c_1, \mu) \longrightarrow^n \mu'$. Then by Lemma 9.1, there exist ν' and m such that $(\mathbf{while}\ e\ \mathbf{do}\ c_1', \mu) \longrightarrow^m \nu'$, $\mu' \sim_\gamma \nu'$, and $m \leq n$. □

Lemma 9.1 *Suppose that c' is a low simulation of c with respect to γ, c is well typed under γ, $dom(\mu) = dom(\gamma)$, and $(\mathbf{while}\ e\ \mathbf{do}\ c, \mu) \longrightarrow^n \mu'$. Then there exist ν' and m such that $(\mathbf{while}\ e\ \mathbf{do}\ c', \mu) \longrightarrow^m \nu'$, $\mu' \sim_\gamma \nu'$, and $m \leq n$.*

Proof. By induction on n.

If $n = 1$ then, by the first LOOP rule, $\mu(e) = 0$ and $\mu' = \mu$. So $(\mathbf{while}\ e\ \mathbf{do}\ c', \mu) \longrightarrow \mu$ by the first LOOP rule.

Suppose $n > 1$. Then by the second LOOP rule, we have

$$(\mathbf{while}\ e\ \mathbf{do}\ c, \mu) \longrightarrow (c; \mathbf{while}\ e\ \mathbf{do}\ c, \mu) \longrightarrow^{n-1} \mu'.$$

By Lemma 9.4, there is a μ'' and k such that $(c, \mu) \longrightarrow^k \mu''$, $(\mathbf{while}\ e\ \mathbf{do}\ c, \mu'') \longrightarrow^{n-1-k} \mu'$, and $0 < k < n - 1$. Since c' is a low simulation of c, there is a memory ν'' and integer i such that $(c', \mu) \longrightarrow^i \nu''$, $\mu'' \sim_\gamma \nu''$ and $i \leq k$. And by induction, there is a ν' and j such that $(\mathbf{while}\ e\ \mathbf{do}\ c', \mu'') \longrightarrow^j \nu'$, $\mu' \sim_\gamma \nu'$ and $j \leq n - 1 - k$.

Now c is well typed under γ and therefore h is not assigned to in c because $\gamma(h) = H$. So $\mu''(h) = \mu(h)$. Further, c' is a low command, so $\nu''(h) = \mu(h)$. Hence $\mu''(h) = \nu''(h)$. Since $\mu'' \sim_\gamma \nu''$, $\mu''(h) = \nu''(h)$, and **while** e **do** c' is a low command, we have $(\mathbf{while}\ e\ \mathbf{do}\ c', \nu'') \longrightarrow^j \nu'$. Therefore by Lemma 9.5,

$$(c'; \mathbf{while}\ e\ \mathbf{do}\ c', \mu) \longrightarrow^{i+j} \nu'.$$

And by the second LOOP rule

$$(\mathbf{while}\ e\ \mathbf{do}\ c', \mu) \longrightarrow (c'; \mathbf{while}\ e\ \mathbf{do}\ c', \mu).$$

Therefore, $(\mathbf{while}\ e\ \mathbf{do}\ c', \mu) \longrightarrow^{i+j+1} \nu'$. And $i + j + 1 \leq k + (n - 1 - k) + 1 = n$. □

Lemma 9.2 (Simple Security) *If $\gamma \vdash e : L$ then $\gamma(x) = L\ var$, for every free identifier x in e.*

Proof. By induction on the structure of e. Since $H \not\sqsubseteq L$, the derivation of $\gamma \vdash e : L$ ends with a trivial application of rule SUBTYPE.

1. Case n. The lemma holds vacuously.
2. Case x. By rule rule R-VAL, $\gamma(x) = L\ var$.
3. Case $match(e)$. By rule QUERY, we have $\gamma \vdash e : L$. By induction, $\gamma(x) = L\ var$ for every free identifier x in e.
4. Case $e_1 = e_2$. By rule EQ, $\gamma \vdash e_1 : L$ and $\gamma \vdash e_2 : L$. By induction, $\gamma(x) = L\ var$ for every free identifier x in e_1 and e_2. The other binary operators are handled similarly.

□

Notice here a departure from our earlier work. We can have $\gamma \vdash e : L$ and $\mu \sim_\gamma \mu'$ yet $\mu(e) \neq \mu'(e)$, if μ and μ' disagree on the value of h. This fact breaks some forms of Noninterference introduced in our earlier work, specifically [12, 13, 14]. These forms are essentially bisimulations which are sensitive to any difference in the value of h.

Lemma 9.3 (Confinement) *If $\gamma \vdash c : H\ cmd$, then we have $\gamma(x) = H\ var$, for every variable x assigned to in c.*

Proof. By induction on the structure of c. Since $L\ cmd \not\subseteq H\ cmd$, the derivation of $\gamma \vdash c : H\ cmd$ ends with a trivial application of rule SUBTYPE.

1. Case **skip**. The lemma holds vacuously.
2. Case $x := e$. By rule ASSIGN, $\gamma(x) = H\ var$.
3. Case $c_1; c_2$. By rule COMPOSE, we have $\gamma \vdash c_1 : H\ cmd$ and $\gamma \vdash c_2 : H\ cmd$. By induction, we have $\gamma(x) = H\ var$ for every variable x assigned to in c_1 and in c_2.
4. Case **if** e **then** c_1 **else** c_2. By rule IF, we have $\gamma \vdash c_1 : H\ cmd$ and $\gamma \vdash c_2 : H\ cmd$. By induction, we have $\gamma(x) = H\ var$ for every variable x assigned to in c_1 and in c_2.
5. Case **while** e **do** c. By rule WHILE, we have $\gamma \vdash c : H\ cmd$. By induction, we have $\gamma(x) = H\ var$ for every variable x assigned to in c.

□

The next two lemmas treat the behavior of sequential composition:

Lemma 9.4 *If $(c_1; c_2, \mu) \longrightarrow^j \mu'$, then there exist k and μ'' such that $0 < k < j$, $(c_1, \mu) \longrightarrow^k \mu''$ and $(c_2, \mu'') \longrightarrow^{j-k} \mu'$.*

Proof. By induction on j. If the derivation begins with an application of the first SEQUENCE rule, then there exists μ'' such that $(c_1, \mu) \longrightarrow \mu''$ and $(c_1; c_2, \mu) \longrightarrow (c_2, \mu'') \longrightarrow^{j-1} \mu'$. So we can let $k = 1$. And, since $j - 1 \geq 1$, we have $k < j$.

If the derivation begins with an application of the second SEQUENCE rule, then there exists c_1' and μ_1 such that $(c_1, \mu) \longrightarrow (c_1', \mu_1)$ and $(c_1; c_2, \mu) \longrightarrow (c_1'; c_2, \mu_1) \longrightarrow^{j-1} \mu'$. By induction, there exists k and μ'' such that $0 < k < j - 1$, $(c_1', \mu_1) \longrightarrow^k \mu''$, and $(c_2, \mu'') \longrightarrow^{j-1-k} \mu'$. Hence $(c_1, \mu) \longrightarrow^{k+1} \mu''$ and $(c_2, \mu'') \longrightarrow^{j-(k+1)} \mu'$. And $0 < k + 1 < j$. □

Lemma 9.5 *If $(c_1, \mu) \longrightarrow^j \mu'$ and $(c_2, \mu') \longrightarrow^k \mu''$, then $(c_1; c_2, \mu) \longrightarrow^{j+k} \mu''$.*

Proof. By induction on j. If $j = 1$ then by the first SEQUENCE rule, $(c_1; c_2, \mu) \longrightarrow (c_2, \mu') \longrightarrow^k \mu''$. Hence $(c_1; c_2, \mu) \longrightarrow^{1+k} \mu''$.

If $j > 1$ then there exist c_1' and μ_1 such that $(c_1, \mu) \longrightarrow (c_1', \mu_1) \longrightarrow^{j-1} \mu'$. By induction, $(c_1'; c_2, \mu_1) \longrightarrow^{j-1+k} \mu''$. And, by the second SEQUENCE rule, $(c_1; c_2, \mu) \longrightarrow (c_1'; c_2, \mu_1)$. Hence $(c_1; c_2, \mu) \longrightarrow^{j+k} \mu''$. □