Language Issues in Mobile Program Security[⋆]

Dennis Volpano[1] and Geoffrey Smith[2]

[1] Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943, USA, email: volpano@cs.nps.navy.mil
[2] School of Computer Science, Florida International University, Miami, FL 33199, USA, email: smithg@cs.fiu.edu

Abstract. Many programming languages have been developed and implemented for mobile code environments. They are typically quite expressive. But while security is an important aspect of any mobile code technology, it is often treated after the fundamental design is complete, in ad hoc ways. In the end, it is unclear what security guarantees can be made for the system. We argue that mobile programming languages should be designed around certain security properties that hold for all well-formed programs. This requires a better understanding of the relationship between programming language design and security. Appropriate security properties must be identified. Some of these properties and related issues are explored.

An assortment of languages and environments have been proposed for mobile code. Some have been designed for use in executable content and others for use in agents [15, 34]. Parallel efforts in extensible networks and operating systems have also focused attention on language design for mobility. These efforts include work on active networks [33, 38], the SPIN kernel [2, 17] and Exokernel [8]. What these efforts have in common is a need for security.

We can roughly separate security concerns in this setting into *code security* and *host security*. The former is concerned with protecting mobile code from untrusted hosts while the latter is concerned with protecting hosts from untrusted mobile code. This may seem a bit artificial since one might like to model security more symmetrically.[1] Nonetheless, it is a useful distinction for now. The code security problem seems quite intractable, given that mobile code is under the control of a host. For some proposals and a discussion, see [25, 26, 40]. In the remainder of this paper, we treat only the host security problem.

1 Host Security

Our view of the problem is that mobile code is executed on a host which must be protected from privacy and integrity violations. As far as privacy goes, the

[⋆] This material is based upon activities supported by DARPA and the National Science Foundation under Agreement Nos. CCR-9612176 and CCR-9612345.

[1] One can imagine a model that does not distinguish mobile code from a host, treating both as mutually suspicious parties.

host has private data that the code may need to perform some expected task. The host wants assurance that it can trust the code not to leak the private data. This is the classical view of privacy [21–23]. As for integrity, the host has information that should not be corrupted. Integrity, in general, demands total code correctness. After all, corrupt data can simply be the result of incorrect code. There are, however, weaker forms of integrity [3].

We believe that an important characteristic of the mobile code setting is that the only observable events are those that can be observed from within a mobile program using language primitives and any host utilities. There are no meta-level observers of a mobile program's behavior such as a person observing its execution behavior online. Still, depending on the language, leaks can occur in many different ways, some being much more difficult to detect than others.

1.1 Security Architectures

A common approach to host security is to monitor the execution of mobile code. You build an interpreter (or virtual machine) and slap the hands of any code that tries to touch something sensitive. The interpreter obviously needs to know whether hand slapping is in order so it might appeal to some sort of trust framework to decide. This arrangement is often called a "security architecture". Architectures are growing quite elaborate as the demand for less hand slapping rises. An example is the security architecture of the Java Developer's Kit JDK1.2 [11]. It blends some proven concepts, such as protection domains, access control, permissions, and code signing, to allow applets more room to maneuver. Netscape's "Object Signing Architecture" takes a similar approach.

One begins to wonder how much of these security architectures is really necessary. Are they a response to a need for host security given mobile programs written in a poorly-designed mobile programming language? Perhaps they can be simplified. It would seem that this is possible if mobile code is written in a language that ensures certain security properties statically.

For example, suppose that all well-typed programs have a secure flow property and that you know a certain program, needing your personal identification number (PIN), is indeed well typed. Then that program respects the privacy of your PIN and there is no need to check at runtime whether the program has permission to read it.

Our claim is not that security architectures will no longer have a role in the future. We feel their role will simply change and be more formally justified. For example, they might carry out certain static analyses or proof checking, perhaps along the lines of proof-carrying code [27]. It should be possible, for a given language, to more clearly identify the role of the security architecture. Certain desirable properties might be provable for all well-formed programs in the language, in which case some security checks can go away.

There are many different facets of mobile language design that influence security in some way. For example, access control mechanisms (encapsulation, visibility rules, etc.) are important. We will limit our attention to some of the issues that impact host privacy and integrity. On the integrity side, we look at *type safety*. Type safety is often said to be a key ingredient for security in Java and for safe kernel extensions written in Modula-3 [2]. Today, some languages like Standard ML evolve with formal treatments of the type system and semantics developed along the way. This allows one to give formal accounts of type safety that evolve as well. Other languages, like Java, lack this sort of formal treatment. Java has grown so rapidly that one quickly loses grasp of the impact of certain features on key properties like type safety.

Then we explore the relationship between privacy and language design. There are many ways mobile code can leak secrets. We start by examining information channels in a deterministic language. We look at how they are influenced by timing, synchrony, and nontermination. Then we consider channels in a simple concurrent language with shared variables. Some of these channels arise in very subtle ways. For example, they can arise from contention among processes for shared resources, like CPU cycles.

2 Type Safety

What is type safety? Consider the following description from a Java perspective:

> The Java language itself is designed to enforce security in the form of type safety. This means the compiler ensures that methods and programs do not access memory in ways that are inappropriate (i.e. dangerous). In effect, this is the most essential part of the Java security model in that it fundamentally protects the integrity of the memory map.
>
> *Secure Computing with Java: Now and the Future,*
> *1997 JavaOne Conference.*

In Java, for example, code should not somehow be able to coerce a reference of a user-defined class to one of a system class like `SecurityManager` which the run-time system (Java Virtual Machine) consults for access permissions. Obviously, this leads to trouble.

So at the heart of type safety is a guarantee against misinterpretation of data—some sequence of bits being misinterpreted by an operation. This has long been recognized as a serious computer security problem. In a well-known report published twenty-five years ago, Anderson describes a way to penetrate a time-sharing system (HIS 635/GCOS III) based on the ability to execute a user's array contents with an assigned GOTO statement in Fortran [1]. The statement can misinterpret its target, the contents of an arbitrary integer variable, as an instruction. Today we see the same sort of problem in a different context [29].

2.1 Type Preservation

An important property related to type safety is the idea of *type preservation*. Type preservation is frequently confused with type soundness in the literature. Soundness is a statement about the progress a program's execution can make if the program is well typed. Type preservation, on the other hand, merely asserts that if a well-typed program evaluates successfully, then it produces a value of the correct type. It is usually needed to prove soundness. For instance, you may know that an expression, with some type, evaluates to a value, but the value must have a specific form in order for evaluation to proceed. Type preservation gives you that the value has the same type as the expression, and with some correct forms typing lemma, you know that only values of the form needed have that type.

The following is a typical type preservation theorem. If μ is a memory, mapping locations to values, and λ is a location typing, mapping locations to types, then type preservation is stated as follows [16]:

Theorem. (Type Preservation). If $\mu \vdash e \Rightarrow v, \mu'$, $\lambda \vdash e : \tau$, and $\mu : \lambda$, then there exists λ' such that $\lambda \subseteq \lambda'$, $\mu' : \lambda'$, and $\lambda' \vdash v : \tau$.

The first hypothesis of the theorem states that under memory μ, a closed expression e evaluates to a value v and a memory μ'. Now e may contain free locations, hence e is typed with respect to a location typing λ which must be consistent with μ, that is, $\mu : \lambda$. Evaluation of e can produce new locations that wind up in v, so v is typed with respect to an extension λ' of λ.

As one can clearly see from the theorem, whether a language exhibits this property depends on the type system and the semantics. In some cases, we might expect that the type system needs to change if the property does not hold. But the semantics itself may be to blame. For instance, consider the C program in Figure 1. When compiled and executed, `*c` evaluates to a signed integer quantity,

```
char *c;
f() {char cc = 'a'; c = &cc;}
g() {int i = -99;}
main() {f(); g(); printf("%c",*c);}
```

Fig. 1. Dereferencing a dangling pointer

yet it has type `char`. If a C semantics prescribes this behavior, then we cannot prove type preservation with respect to that semantics; the C language says this program is unpredictable. This is one place where type preservation and C collide.[2] A formal C semantics should be clear about the outcome of dereferencing a dangling pointer, if this is considered "normal" execution, so that type preservation can be proved. Otherwise, it should specify that execution gets stuck in this situation, again so that type preservation holds. In the latter case, an implementation of C would be required to detect such an erroneous execution point if that implementation were *safe*. A safe (faithful) implementation guarantees that every execution is prescribed by the semantics, so that programs cannot run in

[2] Thus, perhaps, it is not surprising that the SPIN group abandoned its attempts to define a "safe subset of C", adopting Modula-3 instead [2].

ways not accounted for by the semantics. This usually requires that an implementation do some runtime type checking unless it can be proved unnecessary by a type soundness result.

Remark. Although a lack of pointer expressiveness is in general a good thing from a safety viewpoint, manifest pointers (references) are still a substantial security risk. A runtime system might accidentally provide an application a reference to a system security object that must remain invariant. This was demonstrated in Java for JDK1.1. Its entire trust framework, based on digitally-signed code, was undermined when it was discovered that applications could obtain a reference to a code signers array.[3]

2.2 Type Soundness

Type preservation theorems usually talk about successful evaluations. Their hypotheses involve an assumption about an evaluation proceeding in some number of steps to a canonical form. But this may not adequately address a type system's intentions. An objective of a system might be to guarantee termination or to ensure that programs terminate only in specified ways (e.g. no segmentation violations).[4] What is needed is a precise account of how a well-typed program can behave when executed. In other words, we want a type soundness theorem that specifies *all* the possible behaviors that a well-typed program can exhibit.

Traditional type soundness arguments based on showing that a well-typed program does not evaluate to some special untypable value are inadequate for languages like C and Java. There are many reasons why programs written in languages like Java and C may produce runtime errors. Invalid class formats in Java and invalid pointer arithmetic in C are examples. A type soundness theorem should enumerate all the errors that cause a well-typed program to get stuck (abort) according to the semantics. *These are the errors that every safe implementation must detect.* One regards the type system as *sound* if none of these errors is an error that we expect the type system to detect. This is essentially the traditional view of type soundness as a binary property. But a key point to keep in mind is that whether a given type system is sound really depends on our expectations of the type system. Though it may be clear what we expect for languages like Standard ML, it is less clear for lower-level languages like C and assembler.

For example, we give a type system for a polymorphic dialect of C in [30, 32]. The type soundness theorem basically says that executing a well-typed program either succeeds, producing a value of the appropriate type, fails to terminate, or gets stuck because of an attempt to

[3] It is interesting to consider what sort of proof would have revealed the problem. One strategy would be to try finding a P-time reduction from compromising the private key used in a digital signature to executing untrusted code. It would also establish a computational lower bound on executing untrusted code using JDK1.1.

[4] Such properties are important in situations where you need guarantees against certain faults. An example is isolating *execution* behind trust boundaries [17].

– access a dead address,
– access an address with an invalid offset,
– read an uninitialized address, or
– declare an empty or negative-sized array.

The first two errors are due to pointers in the language. Now one may expect the type system to detect the first error in which case our type system is unsound. However, if one believes it is beyond the scope of a type system for C, then our type system is sound. Clearly if the list included an error such as an attempt to apply an integer to an integer, then the type system would generally be regarded as unsound.

A better way to look at type soundness is merely as a property about the executions of programs that the type system says are acceptable. This allows us to compare type systems for a language by comparing their soundness properties. Some may be weaker than others in that they require implementations to check types at run time in order to remain safe. It is also useful for determining whether a particular language is suitable for some application. Some of the errors listed in a formulation of soundness may be among those that an application cannot tolerate. Further, and perhaps most importantly, it identifies those errors that an implementation must detect in order to safely implement the semantics. For instance, a safe implementation of C should trap any attempt to dereference a dangling pointer. Most C implementations are unsafe in this regard. One expects Java implementations to be safer, but despite all the attention to typing and bytecode verification, the current situation is unfortunately not as good as one might imagine.

Consider the Java class in Figure 2.[5] The class modifies itself by putting a `CONSTANT_Utf8` type tag for `x` in that part of the constant pool where a `CONSTANT_String` type tag for `x` is expected by the `ldc` (load from constant pool) instruction. Method `exec` gets a copy of itself in the form of a bytecode array. The class is well typed, yet it aborts with a "segmentation violation" (core dump) in JDK1.1.1, even when Java is run in "verify" mode. Verification of the modified bytecodes does fail using the verify option of the Java class disassembler `javap`. One would expect it to also fail for the bytecodes dynamically constructed in `exec`, leading to a `VerifyError` when class `SelfRef` is run. Instead we get a core dump. So the JDK1.1.1 implementation of Java is unsafe.

Perhaps making class representations available in bytecode form needs to be reconsidered. It becomes quite easy to dynamically construct classes that are difficult to analyze statically for security guarantees. Self-modifying code, in general, makes enforcement of protection constraints difficult, if not impossible. Channel command programs in the M.I.T. Compatible Time-Sharing System (CTSS), for instance, were years ago prohibited from being self-modifying for this reason [28].

[5] A bit of history. The class stems from an attempt to implement an active network in Java. Active programs migrate among servers that invoke their `exec` methods. An active program maintains state by modifying its own bytecode representation prior to being forwarded. Yes, it's a hack.

```
public class SelfRef implements ActiveProgram {

   final String x = "aaba";

   public void exec(byte[] b, MyLoader loader) throws Exception {

      if (b[13] == 0x08) {   // CONSTANT_String
         b[13] = 0x01;       // set CONSTANT_Utf8
         b[14] = 0x00;
         b[15] = 0x00;
         Class classOf = loader.defineClass(b, 0, b.length);
         ActiveProgram p = (ActiveProgram) classOf.newInstance();
         p.exec(b, loader);
      }
      else System.out.println(x);
   }
   public static void main(String[] argv) throws Exception {

      FileInputStream f = new FileInputStream("SelfRef.class");
      byte[] data = new byte[f.available()];
      int c = f.read(data);
      MyLoader loader = new MyLoader();
      new SelfRef().exec(data, loader);
   }
}
```

Fig. 2. Type mismatch leading to segmentation violation in Java

3 Privacy in a Deterministic Language

Suppose we begin by considering a very simple deterministic programming language with just variables, integer-valued expressions, assignment, conditionals, **while** loops, and sequential composition. Programs are executed relative to a memory that maps variables to integers. If a program needs I/O, then it simply reads from or writes to some specific variables of the memory. Further, suppose that some variables of the memory are considered private while others are public. Every program is free to use all variables of the memory and also knows which variables are public and which are private.

What concerns us is whether some program, in this rather anemic language, can *always* produce, in a public variable, the contents of a private variable. There are many such programs, some more complicated than others.

For instance, one can simply assign a private variable to a public one, not a terribly clever strategy for a hacker. This is an example of an *explicit channel*. Or one might try to do it more indirectly, one bit at a time, as in Figure 3 where `PIN` is private, `y` is public, and the value of `mask` is a power of two. This is an example of an *implicit channel*. It illustrates the kind of program we wish to reject because it does not respect the privacy of `PIN`. We need to formalize the security property it violates.

```
while (mask != 0) {
   if (PIN & mask != 0) {    \\ bitwise 'and'
      y := y | mask          \\ bitwise 'or'
   }
   mask := mask / 2
}
```

Fig. 3. Implicit channel

3.1 Privacy Properties

We give a more formal statement of the privacy property we want programs to have in this simple deterministic language:

Definition. (Termination Security). Suppose that c is a command and μ and ν are memories that agree on all public variables. If $\mu \vdash c \Rightarrow \mu'$ and $\nu \vdash c \Rightarrow \nu'$, then μ' and ν' agree on all public variables.

(The judgment $\mu \vdash c \Rightarrow \mu'$ asserts that executing command c in initial memory μ terminates successfully, yielding final memory μ'.) Intuitively, Termination Security says that we can change the contents of private variables without influencing the outcome of public variables. In other words, these changes cannot interfere with the final contents of public variables. The above program does not have this property. Any change in the private `PIN` will result in a different final value for the public variable `y`.

Is the Termination Security property an acceptable privacy property for programs in our simple deterministic language? That depends on what is observable. Consider the similar program in Figure 4. If one can repeatedly run this program

```
while (PIN & mask == 0) { }
y := y | mask
```

Fig. 4. Channel from nontermination

with a different `mask`, one for each bit of `PIN`, then assuming `y` is initially zero, the runs will copy `PIN` to `y`. One `PIN` bit is leaked to `y` in a single run. We assume that, after a specific period of time, if a run has not terminated then it never will, and we move on to the next bit.[6]

But although the program seems insecure, it satisfies Termination Security. Changes to `PIN` cannot influence the outcome of `y` in a single run of the program. After changing the `PIN`, the program may no longer terminate, but this does not violate Termination Security since it only applies to successful termination.

Consider another property:

[6] The task obviously becomes much easier when we enrich the language with threads and a clock. Now each bit can be examined by an asynchronously-running thread, and after some timeout we can be fairly confident that all nonzero bits have been properly set in `y`.

Definition. (Offline Security). Suppose that c is a command and μ and ν are memories that agree on all public variables. If $\mu \vdash c \Rightarrow \mu'$, then there is a ν' such that $\nu \vdash c \Rightarrow \nu'$, and ν' and μ' agree on all public variables.

Notice that we have removed one of the two successful evaluation hypotheses from Termination Security. The property basically says that changing private variables cannot interfere with the final contents of public variables, nor can it interfere with whether a program terminates. We call the property Offline Security because it addresses only what one can observe about a program's behavior if it is executed offline, or in batch mode (one either sees the results of a successful execution or is notified of some timeout that was reached). The time it takes for a program to terminate is not observed. In a deterministic language, Offline Security implies Termination Security. Actually, the formulation of Offline Security is suitable for treating nondeterminism as we shall see. The program in Figure 4 does not satisfy Offline Security.

Unfortunately, there are other sources of channels. Consider the program in Figure 5. Again suppose we can repeatedly execute it with different masks.

```
if (1 / (PIN & mask)) { }
y := y | mask
```

Fig. 5. Channel from partial operation

It always terminates, sometimes abnormally from division by zero. The effect, however, will be the same, to copy `PIN` to `y` one bit at a time.

Hence if we include partial operations like division, we have a situation where a program might either get stuck (terminate abnormally) or run forever, depending on a private variable. So we need yet a stronger offline security property. Basically it needs to extend Offline Security with the condition that if c terminates abnormally under μ, then it does so under ν as well [35].

None of the preceding properties addresses any difference in the *time* required to run a program under two memories that can disagree on private variables. These differences can be used to deduce values of private variables in timing attacks on cryptographic algorithms. For example, a private key used in RSA modular exponentiation has been deduced in this fashion [20]. Differences in timing under two memories can be ruled out by requiring that executions under the two memories proceed in lock step, a form of strong bisimilarity. This property, which might be called Online Security, is the most restrictive thus far. But is it really necessary for mobile programs? That depends on what is observable.

3.2 What is Observable?

Key to judging whether any of the preceding properties is necessary is determining what is observable in the model. Whether a privacy property is suitable depends on how it treats observable events. Notice that there is an observation being exploited in the preceding examples, even within a single run, that allows one PIN bit to be leaked to y. It arises due to the synchrony of sequential composition. Termination Security does not take this kind of observation into account, which makes Offline Security a better choice. Recall that Offline Security does not account for timing differences, but does this matter with mobile code? The key question is who observes the clock?

One can imagine examples in languages like Java where a downloaded applet begins by sending a startup message back to a server on the originating machine and then ends with a finish message. Each message is timestamped by the server which observes a clock external to the applet. We can model this sort of behavior in our simple deterministic language by adding a clock in order to record "timestamps" on values output to memory. Then, clock observation is internal to mobile code. This is how UDP/TCP ports should really be modeled because a UDP or TCP server's clock is observed by a client (applet) when it sends a TCP segment or UDP message. This brings us to our *offline* assumption:

> *In a mobile code setting, the only observable events are those that can be observed internally, that is, from within a mobile program using primitives of the language.*

So the Online Security property may be overly restrictive for mobile programs written in our simple deterministic, sequential language.

Generally, the more a program can observe, the more opportunity there is for leaking secrets. As we have seen, opportunities can arise with the most basic primitives, for instance, synchronous operations.

4 Nondeterminism and Privacy

Now suppose we introduce nondeterminism via a simple concurrent language. It is a multi-threaded imperative language based on the $\pi o\beta\lambda$ model of concurrency [19]. As before, we have commands and their sequential composition. A *thread* is a command that belongs to a thread pool (called an object pool in $\pi o\beta\lambda$). A thread pool O maps thread identifiers to threads. Threads communicate via shared variables of a global memory. A thread pool executes in one step to a new thread pool by nondeterministically selecting a thread and executing it sequentially in one step. More precisely, thread pool transitions are governed by the following two rules:

$$\frac{\begin{array}{l} O(\alpha) = c \\ (c, \mu) \xrightarrow{s} \mu' \end{array}}{(O, \mu) \xrightarrow{g} (O - \alpha, \mu')} \qquad \frac{\begin{array}{l} O(\alpha) = c \\ (c, \mu) \xrightarrow{s} (c', \mu') \end{array}}{(O, \mu) \xrightarrow{g} (O[\alpha := c'], \mu')}$$

Thread pool transitions are denoted $\xrightarrow{g}$ (global transitions) and sequential transitions $\xrightarrow{s}$. The first rule treats thread completion and the second treats thread continuation. Intuitively, the first rule says that if we can pick some thread (command) α from pool O, and execute it sequentially for one step in the shared memory μ, leaving a memory μ', then the entire thread pool O can execute in one step to a pool where α is gone and the shared memory is now μ'. The second rule treats the case where α does not complete but rather is transformed into a continuation (command) c' that represents what remains of c to be executed after it executes for only one step. Note that no thread scheduling policy is specified in these rules.

With threads come new ways to cleverly leak secrets. Programs that appear harmless can contain subtle channels for transmitting secrets, even in this very basic concurrent language. To illustrate, we consider a system introduced by Fine [10]. It is analyzed in [7] where it is concluded that the system is secure in the sense that "it is not possible for a high-level subject to pass information to a low-level subject". The system consists of two private variables, `A` and `B`, whose difference is public and stored in `Y`. As a multi-threaded program, the system is given in Figure 6.[7] Thread α corresponds to a high-level user updating the

```
Thread α :            Thread β :
B := B - A + v;       Y := B - A
A := v
```

Fig. 6. The `AB` system

system with some value `v` that can be recovered through `A`. Thread β corresponds to a low-level user reading public information from the system. The threads share variables `A` and `B`. Imagine each of these threads being executed repeatedly and that `v` is a constant input parameter. The claim is that α cannot transmit any information to β since β always sees only `B - A`. But this requires that α be atomic, for suppose `A` and `B` have initial values `a` and `b` respectively. If we execute the first assignment of α followed by the assignment in β, then `Y` becomes `B - a`, which is `b - a + v - a`. Since `Y` is initially `b - a`, we know `v - a`, the difference between two successive values input by α. So β can observe a difference controlled by α. The interleaving might be frequent in a real implementation if `v` is large.

What kind of privacy property would rule out this sort of threaded program? First, we have to rule out any analog to the Termination Security property because it applies to deterministic programs only. Instead, suppose we ask whether the outcomes of public variables can be "preserved" under changes to private variables. So in the example above, we consider an execution that leaves `Y` equal to `b - a + v - a`, say for `v > a`. Now we ask whether this outcome is *possible* when `v` is changed to a different value, say `w`. No matter how we interleave, `Y` ends up being `b - a` or `b - a + w - a`. The outcome is no longer possible. We have then the following property [31]:

[7] We ignore a third thread for low-level writing to the system.

Definition. (Possibilistic NI). Suppose μ and ν are memories that agree on all public variables and that $(O, \mu) \xrightarrow{g}^{*} (\{\}, \mu')$. Then there is a ν' such that $(O, \nu) \xrightarrow{g}^{*} (\{\}, \nu')$ and ν' and μ' agree on all public variables.

It is a kind of noninterference (NI) property that closely resembles Sutherland's notion of Nondeducibility on inputs [39]. Also notice the similarity between this property and Offline Security.

The program in Figure 6 does not satisfy Possibilistic NI. Another interesting example that does not satisfy Possibilistic NI is given in [31]. It uses a main thread and two triggered threads, each with a busy-wait loop implementing a semaphore, to copy every bit of a private `PIN` to a public variable. In fact, the program *always* produces a copy of the `PIN` in a public variable whenever thread scheduling is fair (every thread is scheduled infinitely often).

Practical extensions of our simple concurrent language make it easy to construct multi-threaded programs that violate Possibilistic NI. For example, simple programs have the property until a scheduling policy, like round-robin time slicing, is introduced [31]. Adding a clock, even without threading, leads to simple programs that fail to have the property. The same is true in the presence of thread priorities and preemption. So the outlook for guaranteeing this property in practical programs written in languages like Java appears bleak.

If a multi-threaded program satisfies Possibilistic NI then changes to private variables cannot interfere with the *possibility* of public variables having a certain outcome. But the changes may interfere with the *probability* of that outcome. If so, there is a probabilistic channel. Consider, for instance, the program in Figure 7. Suppose that `X` stores one bit and is private, `Y` is public, and all threads

Thread α :	Thread β :	Thread γ :
`Y := X`	`Y := 0`	`Y := 1`

Fig. 7. A probabilistic channel

have an equal probability of being scheduled. Is the program secure? Well, it satisfies Possibilistic NI so it cannot reveal `X` with certainty. But it is *likely* to reveal `X`. Suppose `X` is 0. Then the probability that `Y` has final value 0 is 2/3. When `X` is 1, however, the probability that `Y` has final value 0 drops to 1/3. In effect, the private variable interferes with the *probability* that `Y` has final value 0. This kind of interference Gray calls *probabilistic interference* [12]. He describes a property called P-restrictiveness that aims to rule it out in systems. The property can be viewed as a form of probabilistic noninterference [13, 14].

5 Logics and Static Analyses for Privacy

¿From the discussion thus far, it would appear that a privacy property is developed independently of any logic for reasoning about it. While this has been generally true of security properties studied for computer systems, it is usually not so for programming languages. Typically one starts with an intuitive idea of secure code and gives some sort of logic to capture the notion. The next step is to make the intuition precise so that the logic can be proved sound.[8] To illustrate, we sketch a logic for reasoning about privacy below. It is actually a type system utilizing subtypes. A complete description can be found in [37] where it is proved that every well-typed deterministic program satisfies Termination Security.

We take security classes, like L (low or public) and H (high or private), as our basic types which we denote by τ. Some typing rules treat explicit channels and others implicit channels. Below is the typing rule for an assignment $x := e$:

$$\frac{\gamma \vdash x : \tau\ acc, \quad \gamma \vdash e : \tau}{\gamma \vdash x := e : \tau\ cmd} \tag{1}$$

In order for the assignment to be well typed, it must be that

- x is a variable of type τ *acc*(eptor), meaning x is capable of storing information at security level τ, and
- expression e has type τ, meaning every variable in e has type τ.[9]

Information about x is provided by γ which maps identifiers to types. So, the rule states that in order for the assignment $x := e$ to be judged secure, x must be a variable that stores information at the same security level as e. If this is true, then the rule allows us to ascribe type $\tau\ cmd$ to the entire assignment command. The command type $\tau\ cmd$ tells us that every variable assigned to by the command (here only x) can accept information of security level τ or higher.

These command types are needed to control implicit channels like the one in Figure 3. For example, here is the typing rule for conditionals:

$$\frac{\begin{array}{l}\gamma \vdash e : \tau \\ \gamma \vdash c_1 : \tau\ cmd \\ \gamma \vdash c_2 : \tau\ cmd\end{array}}{\gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \tau\ cmd} \tag{2}$$

The idea is that c_1 and c_2 execute in a context where information about the value of e is implicitly available—when c_1 executes, the value of e was *true* and when c_2 executes, the value of e was *false*. Hence if $e : \tau$, then c_1 and c_2 must not transmit any information to variables of security level lower than τ. This is enforced by requiring c_1 and c_2 to have type $\tau\ cmd$.

Here is the typing rule for **while** loops:

$$\frac{\begin{array}{l}\gamma \vdash e : \tau \\ \gamma \vdash c : \tau\ cmd\end{array}}{\gamma \vdash \textbf{while}\ e\ \textbf{do}\ c : \tau\ cmd} \tag{3}$$

[8] Unfortunately, it is quite common to see either the logic skipped entirely, in favor of an algorithm that implements one's intuition, or soundness not treated adequately, if at all. It is important to make intuitions about security precise.

[9] Keep in mind that unlike type preservation, an expression of type τ here does not mean one that evaluates to a value of type τ. Values (in our case integers) have no intrinsic security levels.

and the typing rule for sequential composition:

$$\frac{\begin{array}{l}\gamma \vdash c_1 : \tau\ cmd \\ \gamma \vdash c_2 : \tau\ cmd\end{array}}{\gamma \vdash c_1; c_2 : \tau\ cmd} \tag{4}$$

The typing rules for expressions and commands simply require all subexpressions and subcommands to be typed at the same security level. For example, we require in rule (1) that the left and right sides of an assignment be typed at the same level. A similar requirement is imposed in rule (2). Yet we do want to allow *upward* information flows, such as from public to private. But the typing rules can remain simple because upward flows can be accommodated naturally through subtyping. For example, we would have $L \subseteq H$, but not $H \subseteq L$. The subtype relation can naturally be extended with subtype inclusions among types of the form $\tau\ cmd$ and $\tau\ acc$. The type constructors *cmd* and *acc* are *antimonotonic*, meaning that if $\tau_1 \subseteq \tau_2$, then the relation is extended with

$$\tau_2\ cmd \subseteq \tau_1\ cmd \ \text{ and } \ \tau_2\ acc \subseteq \tau_1\ acc$$

Intuitively, antimonotonicity merely reflects the fact that a reader capable of reading at one security level is capable of reading at a lower level.

Also, there are two coercions associated with variables. If $x : \tau\ var$, then $x : \tau$ and also $x : \tau\ acc$. That is, variables are both expressions and acceptors. So if

$$\gamma(x) = H\ var \ \text{ and } \ \gamma(y) = L\ var$$

then there is an explicit upward flow from y to x in $x := y$. The assignment can be typed in two ways. We can type the assignment with $x : H\ acc$ by coercing the type of y to H, or we can type the assignment with $y : L$ by coercing the type of x to $L\ acc$ through the antimonotonicity of acceptor types.

Now properties of the type system can be proved. For example, there are type-theoretic analogs of the well-known simple security property and $*$-property (Confinement) of the Bell and LaPadula model [22, 23]:

Lemma. (Type Analog of Simple Security) If $\gamma \vdash e : \tau$, then for every variable x in expression e, $\gamma \vdash x : \tau$.

Lemma. (Type Analog of Confinement) If $\gamma \vdash c : \tau\ cmd$, then for every variable x assigned to in command c, $\gamma \vdash x : \tau\ acc$.

Intuitively, Simple Security guarantees no "read up" in expressions, whereas Confinement ensures no "write down" in commands. For example, Simple Security ensures that if an expression has type L, then it contains no variables of type $H\ var$. Likewise, Confinement guarantees that if a command has type $H\ cmd$, then it contains no assignments to variables of type $L\ var$.

With these two properties, one can now prove that every well-typed program in our simple deterministic language satisfies Termination Security [37]. The type system is not limited to privacy. One can also introduce integrity classes T

(trusted) and U (untrusted), such that $T \subseteq U$. Now if a program satisfies Termination Security, then no trusted variable can be "contaminated" by untrusted variables [3].

To achieve stronger security properties, such as Offline Security it is necessary to restrict the typing of **while** loops. Intuitively, a **while** loop can transmit information not only by assigning to variables, but also by terminating or failing to terminate. This idea was exploited by the program in Figure 4. To prevent such flows, one can restrict rule (3) to the following:

$$\frac{\begin{array}{l}\gamma \vdash e : L \\ \gamma \vdash c : L\ cmd\end{array}}{\gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ c : L\ cmd} \tag{5}$$

With this stricter rule, one can show that every well-typed program satisfies Offline Security [35]. By restricting the typing of partial operations like division, it is also shown in [35] that well-typed programs satisfy a stronger offline security property that addresses aborted executions as well as nontermination. Finally, under rule (5) it can be shown that every well-typed concurrent program satisfies Possibilistic NI [31].

An advantage of the type system is that it affords type inference. Procedures are polymorphic with respect to security classes. Principal types are constrained type schemes that convey how code can be used without violating privacy [36]. Notice that type checking here is not merely an optimization in that it replaces run-time checks, as in traditional type checking. Denning's early work on program certification and the lattice model over 20 years ago [4–6] showed that one cannot rely only on run-time mechanisms to enforce secure information flow, a direction that had been pursued by Fenton [9]. Static analysis is needed to reveal implicit channels like the one in Figure 3.

There is still some question about how the type system should be deployed in a mobile code setting. Currently we are exploring its use in a code certification pipeline aimed at certifying the security of e-commerce applications written in Java. But we can also imagine the need for analyzing some lower-level intermediate language like Java virtual machine instructions. The loss of program structure at this level would likely make it more difficult to specify a simple type system for privacy.

5.1 Decidability

The type system above is decidable. A type inference algorithm is given for it in [36]. A desirable property of any logic for reasoning about privacy is that it be decidable. However, there is often tension between decidability, soundness and completeness in such logics. One is naturally unwilling to compromise soundness so that can mean having to give up completeness for decidability.

For instance, the problem of deciding whether a program, written in our simple deterministic language of Section 3, has the Termination Security property is not recursively enumerable. This means that any sound and recursively enumerable logic for reasoning about Termination Security must be incomplete. Now the question is how much have we lost by conceding incompleteness? There must be examples of code that have some desired security property, but which cannot be proved in the logic. For example, here is a snippet of code in our sequential language that satisfies Termination Security, yet is untypable in the system above if `X` is private and `Y` is public:

```
X := Y;
Y := X
```

Further, thread β in the program of Figure 6 is also untypable. The question of how much has been lost often depends on whether such examples arise frequently in practice. If they do, then the logic may yield too many "false positives" for it to be practical. This has been a primary criticism of information-flow checkers for some time [7].

6 Conclusion

We have explored the relationship between some aspects of language design and security issues. The issues we considered in this paper, namely type safety and privacy, are really independent of code mobility. Nonetheless, the prospect of migrating code that executes financial transactions, or extends the functionality of a network switch, makes them relevant.

So what sort of advice can we offer designers of secure languages? First, security should not be viewed as a programming language graft. The literature is filled with attempts that treat security this way. Languages have a fundamental role in secure computation and should be designed with this in mind. A designer might begin by establishing the security properties of interest for a language and then attempt to introduce functionality while preserving them. This seems more promising than treating security afterward. Also, one cannot overemphasize the need for a formal semantics. It is essential for proving soundness and basic safety properties like those in [24].

We strongly believe that secure languages should have simple, compositional logics for reasoning about the security properties of interest. Compilers should be able to incorporate decision procedures for these logics as static analyses that programmers can easily understand. For instance, the type system of Section 5 is simple and has an efficient type inference algorithm for inferring type schemes that convey how programs can be used securely.

As far as privacy properties go, one has to know what is observable, and how it can be observed. There are some known pitfalls. In a concurrent setting, beware of any ability to modulate one thread with another, for instance, through a semaphore [31]. Time-sliced thread scheduling is also problematic. It does not preserve the Possibilistic NI security property in languages like Java. Java threading and its many features make it easy to build covert timing channels. This suggests that it is unsuitable for secure e-commerce applications. The subset, Java Card 2.0, proposed for smartcards, may be better since it has no threading and supports the notion of a transaction [18]. Designing a secure concurrent language that is flexible and admits simple and accurate static analyses is the subject of current research.

References

1. James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Electronic Systems Division, Hanscom Field, Bedford, MA, 1972.
2. Brian Bershad, et al. Extensibility, safety and performance in the SPIN operating system. In *Proc. 15th Symposium on Operating Systems Principles*, pages 267–284, December 1995.
3. K. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, MITRE Corp., 1977.
4. Dorothy Denning. *Secure Information Flow in Computer Systems.* PhD thesis, Purdue University, West Lafayette, IN, May 1975.
5. Dorothy Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.
6. Dorothy Denning and Peter Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
7. Steven T. Eckmann. Eliminating formal flows in automated information flow analysis. In *Proceedings 1994 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1994.
8. D.R. Engler, et al. Exokernel: An operating system architecture for application-level resource management. In *Proc. 15th Symposium on Operating Systems Principles*, December 1995.
9. J. Fenton. *Information Protection Systems.* PhD thesis, University of Cambridge, 1973.
10. Todd Fine. A foundation for covert channel analysis. In *Proc. 15th National Computer Security Conference*, Baltimore, MD, October 1992.
11. Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997.
12. James W. Gray, III. Probabilistic interference. In *Proceedings 1990 IEEE Symposium on Security and Privacy*, pages 170–179, Oakland, CA, May 1990.
13. James W. Gray, III. Toward a mathematical foundation for information flow security. In *Proceedings 1991 IEEE Symposium on Security and Privacy*, pages 21–34, Oakland, CA, May 1991.
14. James W. Gray, III and Paul F. Syverson. A logical approach to multilevel security of probabilistic systems. In *Proceedings 1992 IEEE Symposium on Security and Privacy*, pages 164–176, Oakland, CA, May 1992.
15. David Halls, John Bates, and Jean Bacon. Flexible distributed programming using mobile code. In *Proc. 7th ACM SIGOPS European Workshop, Systems Support for Worldwide Applications*, Connemara, Ireland, September 1996.
16. Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51:201–206, 1994.
17. Wilson C. Hsieh, et al. Language support for extensible operating systems. Unpublished manuscript. Available at `www-spin.cs.washington.edu.`, 1996.

18. *Java Card 2.0 Language Subset and Virtual Machine Specification*. Sun Microsystems, October 1997.
19. Cliff B. Jones. Some practical problems and their influence on semantics. In *Proceedings of the 6th European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 1–17, April 1996.
20. Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. In *Proceedings 16th Annual Crypto Conference*, August 1996.
21. Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.
22. Carl E. Landwehr. Formal models for computer security. *Computing Surveys*, 13(3):247–278, 1981.
23. Leonard J. LaPadula and D. Elliot Bell. MITRE Technical Report 2547, Volume II. *Journal of Computer Security*, 4(2,3):239–263, 1996.
24. X. Leroy and F. Rouaix. Security properties of typed applets. In *Proceedings 25th Symposium on Principles of Programming Languages*, pages 391–403, San Diego, CA, January 1998.
25. Catherine Meadows. Detecting attacks on mobile agents. In *Proc. 1997 Foundations for Secure Mobile Code Workshop*, pages 64–65, Monterey, CA, March 1997.
26. Yaron Minsky, Robbert van Renesse, Fred B. Schneider, and Scott Stoller. Cryptographic support for fault-tolerant distributed computing. In *Proc. 7th ACM SIGOPS European Workshop, Systems Support for Worldwide Applications*, Connemara, Ireland, September 1996.
27. George Necula and Peter Lee. Proof-carrying code. In *Proceedings 24th Symposium on Principles of Programming Languages*, Paris, France, 1997.
28. J.H. Saltzer. Case studies of protection system failures. Appendix 6-A, unpublished course notes on The Protection of Information in Computer Systems., 1975.
29. Vijay Saraswat. Java is not type-safe. Unpublished manuscript. Available at `www.research.att.com/~vj/bug.html.`, 1997.
30. Geoffrey Smith and Dennis Volpano. Towards an ML-style polymorphic type system for C. In *Proceedings of the 6th European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 341–355, April 1996.
31. 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.
32. Geoffrey Smith and Dennis Volpano. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, 32(2–3), 1998. To appear.
33. D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden. A survey of active network research. *IEEE Communications*, 35(1):80–86, January 1997.
34. Tommy Thorn. Programming languages for mobile code. *Computing Surveys*, 29(3):213–239, 1997.
35. Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proc. 10th IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.
36. Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proc. Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621, April 1997.
37. 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.

38. David J. Wetherall and David L. Tennenhouse. The ACTIVE IP option. In *Proc. 7th ACM SIGOPS European Workshop, Systems Support for Worldwide Applications*, Connemara, Ireland, September 1996.
39. J. Todd Wittbold and Dale M. Johnson. Information flow in nondeterministic systems. In *Proceedings 1990 IEEE Symposium on Security and Privacy*, pages 144–161, Oakland, CA, May 1990.
40. Bennet S. Yee. A sanctuary for mobile agents. In *Proc. 1997 Foundations for Secure Mobile Code Workshop*, pages 21–27, Monterey, CA, March 1997.