April 17, 1998

SRC Research Report 154

Protection in Programming-Language Translations

Martín Abadi

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.digital.com/SRC/

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Protection in
Programming-Language Translations

Martín Abadi

April 17, 1998

This paper will appear in the Proceedings of the 25th International Colloquium on Automata, Languages and Programming, to be published by Springer-Verlag in July 1998.

Abstract

We discuss abstractions for protection and the correctness of their implementations. Relying on the concept of full abstraction, we consider two examples: (1) the translation of Java classes to an intermediate bytecode language, and (2) in the setting of the pi calculus, the implementation of private channels in terms of cryptographic operations.

Contents

1 Introduction

Tangible crimes and measures against those crimes are sometimes explained through abstract models—with mixed results, as the detective Erik Lönnrot discovered [Bor74]. Protection in computer systems relies on abstractions too. For example, an access matrix is a high-level specification that describes the allowed accesses of subjects to objects in a computer system; the system may rely on mechanisms such as access lists and capabilities for implementing an access matrix [Lam71].

Abstractions are often embodied in programming-language constructs. Recent work on Java [GJS96] has popularized the idea that languages are relevant to security, but the relation between languages and security is much older. In particular, objects and types have long been used for protection against incompetence and malice, at least since the 1970s [Mor73, LS76, JL78]. In the realm of distributed systems, programming languages (or their libraries) have sometimes provided abstractions for communication on secure channels of the kind implemented with cryptography [Bir85, WABL94, vDABW96, WRW96, Sun97b].

Security depends not only on the design of clear and expressive abstractions but also on the correctness of their implementations. Unfortunately, the criteria for correctness are rarely stated precisely—and presumably they are rarely met. These criteria seem particularly delicate when a principal relies on those abstractions but interacts with other principals at a lower level. For example, the principal may express its programs and policies in terms of objects and remote method invocations, but may send and receive bit strings. Moreover, the bit strings that it receives may not have been the output of software trusted to respect the abstractions. Such situations seem to be more common now than in the 1970s.

One of the difficulties in the correct implementation of secure systems is that the standard notion of refinement (e.g., [Hoa72, Lam89]) does not preserve security properties. Ordinarily, the non-determinism of a specification may be intended to allow a variety of implementations. In security, the non-determinism may also serve for hiding sensitive data. As an example, let us consider a specification that describes a computer that displays an arbitrary but fixed string in a corner of a screen. A proposed implementation might always display a user's password as that string. Although this implementation may be functionally correct, we may consider it incorrect for security purposes, because it leaks more information than the specification seems to allow. Security properties are thus different from other common properties; in fact, it has been argued that security properties do not conform to the

Alpern-Schneider definition of properties [AS85, McL96].

Reexamining this example, let us write P for the user's password, $I(P)$ for the proposed implementation, and $S(P)$ for the specification. Since the set of behaviors allowed by the specification does not depend on P, clearly $S(P)$ is equivalent to $S(P')$ for any other password P'. On the other hand, $I(P)$ and $I(P')$ are not equivalent, since an observer can distinguish them. Since the mapping from specification to implementation does not preserve equivalence, we may say that it is not fully abstract [Plo77]. We may explain the perceived weakness of the proposed implementation by this failure of full abstraction.

This paper suggests that, more generally, the concept of full abstraction is a useful tool for understanding the problem of implementing secure systems. Full abstraction seems particularly pertinent in systems that rely on translations between languages—for example, higher-level languages with objects and secure channels, lower-level languages with memory addresses and cryptographic keys.

We consider two examples of rather different natures and review some standard security concerns, relating these concerns to the pursuit of full abstraction. The first example arises in the context of Java (section 2). The second one concerns the implementation of secure channels, and relies on the pi calculus as formal framework (section 3). The thesis of this paper about full abstraction is in part a device for discussing these two examples.

This paper is rather informal and partly tutorial; its contributions are a perspective on some security problems and some examples, not new theorems. Related results appear in more technical papers [SA98, AFG98].

Full abstraction, revisited

We say that two expressions are equivalent in a given language if they yield the same observable results in all contexts of the language. A translation from a language L_1 to a language L_2 is equationally fully abstract if (1) it maps equivalent L_1 expressions to equivalent L_2 expressions, and (2) conversely, it maps nonequivalent L_1 expressions to nonequivalent L_2 expressions [Plo77, Sha91, Mit93]. We may think of the context of an expression as an attacker that interacts with the expression, perhaps trying to learn some sensitive information (e.g., [AG97a]). With this view, condition (1) means that the translation does not introduce information leaks. Since equations may express not only secrecy properties but also some integrity properties, the translation must preserve those properties as well. Because of these consequences of condition (1), we focus on it; we mostly ignore condition (2), although it can be useful too, in particular for excluding trivial translations.

Closely related to equational full abstraction is logical full abstraction. A translation from a language L_1 to a language L_2 is logically fully abstract if it preserves logical properties of the expressions being translated [LP98]. Longley and Plotkin have identified conditions under which equational and logical full abstraction are equivalent. Since we use the concept of full abstraction loosely, we do not distinguish its nuances.

An expression of the source language L_1 may be written in a silly, incompetent, or even malicious way. For example, the expression may be a program that broadcasts some sensitive information—so this expression is insecure on its own, even before any translation to L_2. Thus, full abstraction is clearly not sufficient for security; however, as we discuss in this paper, it is often relevant.

2 Objects and Mobile Code

The Java programming language is typically compiled to an intermediate language, which we call JVML and which is implemented by the Java Virtual Machine [GJS96, LY96]. JVML programs are often communicated across networks, for example from Web servers to their clients. A client may run a JVML program in a Java Virtual Machine embedded in a Web browser. The Java Virtual Machine helps protect local resources from mobile JVML programs while allowing those programs to interact with local class libraries. Some of these local class libraries perform essential functions (for example, input and output), so they are often viewed as part of the Java Virtual Machine.

2.1 Translating Java to JVML

As a first example we consider the following trivial Java class:

```
class C {
    private int x;
    public void set_x(int v) {
      this.x = v;
    };
}
```

This class describes objects with a field `x` and a method `set_x`. The method `set_x` takes an integer argument `v` and updates the field `x` to `v`. The keyword

this represents the self of an object; the keyword **public** indicates that any client or subclass can access `set_x` directly; the keyword **private** disallows a similar direct access to `x` from outside the class. Therefore, the field `x` can be written but never read.

The result of compiling this class to JVML may be expressed roughly as follows. (Here we do not use the official, concrete syntax of JVML, which is not designed for human understanding.)

```
class C {
    private int x;
    public void set_x(int) {
      .framelimits locals = 2, stack = 2;
      aload_0;       // load this
      iload_1;       // load v
      putfield x;    // set x
    };
}
```

As this example indicates, JVML is a fairly high-level language, and in particular it features object-oriented constructs such as classes, methods, and self. It differs from Java in that methods manipulate local variables, a stack, and a heap using low-level load and store operations. The details of those operations are not important for our purposes. Each method body declares how many local variables and stack slots its activation may require. The Java Virtual Machine includes a bytecode verifier, which checks that those declarations are conservative (for instance, that the stack will not overflow). If undetected, dynamic errors such as stack overflow could lead to unpredictable behavior and to security breaches.

The writer of a Java program may have some security-related expectations about the program. In our simple example, the field `x` cannot be read from outside the class, so it may be used for storing sensitive information. Our example is so trivial that this information cannot be exploited in any way, but there are more substantial and interesting examples that permit controlled access to fields with the qualifier **private** and similar qualifiers. For instance, a Java class for random-number generation (like `java.util.Random`) may store seeds in private fields. In these examples, a security property of a Java class may be deduced—or presumed—by considering all possible Java contexts in which the class can be used. Because those contexts must obey the type rules of Java, they cannot access private fields of the class.

When a Java class is translated to JVML, one would like the resulting JVML code to have the security properties that were expected at the Java level. However, the JVML code interacts with a JVML context, not with a Java context. If the translation from Java to JVML is fully abstract, then matters are considerably simplified—in that case, JVML contexts have no more power than Java contexts. Unfortunately, as we point out below, the current translation is not fully abstract (at least not in a straightforward sense). Nevertheless, the translation approximates full abstraction:

- In our example, the translation retains the qualifier **private** for `x`. The occurrence of this qualifier at the JVML level may not be surprising, but it cannot be taken for granted. (At the JVML level, the qualifier does not have the benefit of helping programmers adhere to sound software-engineering practices, since programmers hardly ever write JVML, so the qualifier might have been omitted.)

- Furthermore, the bytecode verifier can perform standard typechecking, guaranteeing in particular that a JVML class does not refer to a private field of another JVML class.

- The bytecode verifier can also check that dynamic errors such as stack overflow will not occur. Therefore, the behavior of JVML classes should conform to the intended JVML semantics; JVML code cannot get around the JVML type system for accessing a private field inappropriately.

Thus, the bytecode verifier restricts the set of JVML contexts, and in effect makes them resemble Java contexts (cf. [GJS96, p. 220]). As the set of JVML contexts decreases, the set of equivalences satisfied by JVML programs increases, so the translation from Java to JVML gets closer to full abstraction. Therefore, we might even view full abstraction as the goal of bytecode verification.

Recently, there have been several rigorous studies of the Java Virtual Machine, and in particular of the bytecode verifier [Coh97, SA98, Qia97, FM98]. These studies focus on the type-safety of the JVML programs accepted by the bytecode verifier. As has long been believed, and as Leroy and Rouaix have recently proved in a somewhat different context [LR98], strong typing yields some basic but important security guarantees. However, those guarantees do not concern language translations. By themselves, those guarantees do not imply that libraries written in a high-level language have expected security properties when they interact with lower-level mobile code.

2.2 Obstacles to full abstraction

As noted, the current translation of Java to JVML is not fully abstract. The following variant of our first example illustrates the failure of full abstraction. We have no reason to believe that it illustrates the only reason for the failure of full abstraction, or the most worrisome one; Dean, Felten, Wallach, and Balfanz have discovered several significant discrepancies between the semantics of Java and that of JVML [DFWB98].

```
class D {
    class E {
        private int y = x;
    };
    private int x;
    public void set_x(int v) {
      this.x = v;
    };
}
```

The class `E` is an inner class [Sun97a]. To each instance of an inner class such as `E` corresponds an instance of its outer class, `D` in this example. The inner class may legally refer to the private fields of the outer class.

Unlike Java, JVML does not include an inner-class construct. Therefore, compilers "flatten" inner classes while adding accessor methods. Basically, as far as compilation is concerned, we may as well have written the following classes instead of `D`:

```
class D {
    private int x;
    public void set_x(int v) {
      this.x = v;
    };
    static int get_x(D d) {
      return d.x;
    };
}

class E {
    ... get_x ...
}
```

Here `E` is moved to the top level. A method `get_x` is added to `D` and used in `E` for reading `x`; the details of `E` do not matter for our purposes. The method `get_x` can be used not just in `E`, however—any other class within the same package may refer to `get_x`.

When the classes `D` and `E` are compiled to JVML, therefore, a JVML context may be able to read `x` in a way that was not possible at the Java level. This possibility results in the loss of full abstraction, since there is a JVML context that distinguishes objects that could not be distinguished by any Java context. More precisely, a JVML context that runs `get_x` and returns the result distinguishes instances of `D` with different values for `x`.

This loss of full abstraction may result in the leak of some sensitive information, if any was stored in the field `x`. The leak of the contents of a private component of an object can be a concern when the object is part of the Java Virtual Machine, or when it is trusted by the Java Virtual Machine (for example, because a trusted principal digitally signed the object's class). On the other hand, when the object is part of an applet, this leak should not be surprising: applets cannot usually be protected from their execution environments.

For better or for worse, the Java security story is more complicated and dynamic than the discussion above might suggest. In addition to protection by the qualifier **private**, Java has a default mode of protection that protects classes in one package against classes in other packages. At the language level, this mode of protection is void—any class can claim to belong to any package. However, Java class loaders can treat certain packages in special ways, guaranteeing that only trusted classes belong to them. Our example with inner classes does not pose a security problem as long as `D` and `E` are in one of those packages.

In hindsight, it is not clear whether one should base any security expectations on qualifiers like **private**, and more generally on other Java constructs. As Dean et al. have argued [DFWB98], the definition of Java is weaker than it should be from a security viewpoint. Although it would be prudent to strengthen that definition, a full-blown requirement of full abstraction may not be a necessary addition. More modest additions may suffice. Section 4 discusses this subject further.

3 Channels for Distributed Communication

In this section, we consider the problem of implementing secure channels in distributed systems. As mentioned in the introduction, some systems for distributed programming offer abstractions for creating and using secure channels. The implementations of those channels typically rely on cryptography for ensuring the privacy and the integrity of network communication. The relation between the abstractions and their implementations is usually explained only informally. Moreover, the abstractions are seldom explained in a self-contained manner that would permit reasoning about them without considering their implementations at least occasionally.

The concept of full abstraction can serve as a guide in understanding secure channels. When trying to approximate full abstraction, we rediscover common attacks and countermeasures. Most importantly, the pursuit of full abstraction entails a healthy attention to the connections between an implementation and higher-level programs that use the implementation, beyond the intrinsic properties of the implementation.

3.1 Translating the pi calculus to the spi calculus

The formal setting for this section is the pi calculus [Mil92, MPW92, Mil93], which serves as a core calculus with primitives for creating and using channels. By applying the pi calculus restriction operator, these channels can be made private. We discuss the problem of mapping the pi calculus to a lower-level calculus, the spi calculus [AG97b, AG97c, AG97a], implementing communication on private channels by encrypted communication on public channels. Several low-level attacks can be cast as counterexamples to the full abstraction of this mapping. Some of the attacks can be thwarted through techniques common in the literature on protocol design [MvOV96]. Some other attacks suggest fundamental difficulties in achieving full abstraction for the pi calculus.

First we briefly review the spi calculus. In the variant that we consider here, the syntax of this calculus assumes an infinite set of names and an infinite set of variables. We let c, d, m, n, and p range over names, and let w, x, y, and z range over variables. We usually assume that all these names and variables are different (for example, that m and n are different names). The set of terms of the spi calculus is defined by the following grammar:

$L, M, N ::=$	terms
$\quad n$	name
$\quad x$	variable
$\quad \{M_1, \ldots, M_k\}_N$	encryption ($k \geq 0$)

Intuitively, $\{M_1, \ldots, M_k\}_N$ represents the ciphertext obtained by encrypting the terms $M_1, \ldots, M_k$ under the key N (using a symmetric cryptosystem such as DES or RC5 [MvOV96]). The set of processes of the spi calculus is defined by the following grammar:

$P, Q ::=$	processes
$\quad \overline{M}\langle N_1, \ldots, N_k \rangle$	$\quad$ output ($k \geq 0$)
$\quad M(x_1, \ldots, x_k).P$	$\quad$ input ($k \geq 0$)
$\quad \mathbf{0}$	$\quad$ nil
$\quad P \mid Q$	$\quad$ composition
$\quad !P$	$\quad$ replication
$\quad (\nu n)P$	$\quad$ restriction
$\quad [M \textit{ is } N]\ P$	$\quad$ match
$\quad \textit{case } L \textit{ of } \{x_1, \ldots, x_k\}_N \textit{ in } P$	$\quad$ decryption ($k \geq 0$)

An output process $\overline{M}\langle N_1, \ldots, N_k \rangle$ sends the tuple $N_1, \ldots, N_k$ on M. An input process $M(x_1, \ldots, x_k).Q$ is ready to input k terms $N_1, \ldots, N_k$ on M, and then to behave as $Q[N_1/x_1, \ldots, N_k/x_k]$. Here we write $Q[N_1/x_1, \ldots, N_k/x_k]$ for the result of replacing each free occurrence of x_i in Q with N_i, for $i \in 1..k$. Both $M(x_1, \ldots, x_k).Q$ and $\textit{case } L \textit{ of } \{x_1, \ldots, x_k\}_N \textit{ in } P$ (explained below) bind the variables $x_1, \ldots, x_k$. The nil process $\mathbf{0}$ does nothing. A composition $P \mid Q$ behaves as P and Q running in parallel. A replication $!P$ behaves as infinitely many copies of P running in parallel. A restriction $(\nu n)P$ makes a new name n and then behaves as P; it binds the name n. A match process $[M \textit{ is } N]\ P$ behaves as P if M and N are equal; otherwise it does nothing. A decryption process $\textit{case } L \textit{ of } \{x_1, \ldots, x_k\}_N \textit{ in } P$ attempts to decrypt L with the key N; if L has the form $\{M_1, \ldots, M_k\}_N$, then the process behaves as $P[M_1/x_1, \ldots, M_k/x_k]$; otherwise it does nothing.

By omitting the constructs $\{M_1, \ldots, M_k\}_N$ and $\textit{case } L \textit{ of } \{x_1, \ldots, x_k\}_N \textit{ in } P$ from these grammars, we obtain the syntax of the pi calculus (more precisely, of a polyadic, asynchronous version of the pi calculus).

As a first example, we consider the trivial pi calculus process:

$$(\nu n)(\overline{n}\langle m \rangle \mid n(x).\mathbf{0})$$

This is a process that creates a channel n, then uses it for transmitting the name m, with no further consequence. Communication on n is secure in the sense that no context can discover m by interacting with this process, and no context can cause a different message to be sent on n; these are typical secrecy and integrity properties. Such properties can be expressed as equivalences (in particular, as testing equivalences [DH84, BN95, AG97a]). For example, we may express the secrecy of m as the equivalence between $(\nu n)(\overline{n}\langle m \rangle \mid n(x).\mathbf{0})$ and $(\nu n)(\overline{n}\langle m' \rangle \mid n(x).\mathbf{0})$, for any names m and m'.

Intuitively, the subprocesses $\overline{n}\langle m \rangle$ and $n(x).\mathbf{0}$ may execute on different machines; the network between these machines may not be physically secure. Therefore, we would like to explicate a channel like n in lower-level terms, mapping it to some sort of encrypted connection multiplexed on a public channel. For example, we might translate our first process, $(\nu n)(\overline{n}\langle m\rangle \mid n(x).\mathbf{0})$, into the following spi calculus process:

$$(\nu n)(\overline{c}\langle\{m\}_n\rangle \mid c(y).case\ y\ of\ \{x\}_n\ in\ \mathbf{0})$$

Here c is a distinguished, free name, intuitively the name of a well-known public channel. The name n still appears, with a restriction, but it is used for a key rather than for a channel. The sender encrypts m using n; the recipient tries to decrypt a ciphertext y that it receives on c using n; if the decryption succeeds, the recipient obtains a cleartext x (hopefully m).

This translation strategy may seem promising. However, it has numerous weaknesses; we describe several of those weaknesses in what follows. The weaknesses represent obstacles to full abstraction and are also significant in practical terms.

3.2 Obstacles to full abstraction

Leak of traffic patterns

In the pi calculus, $(\nu n)(\overline{n}\langle m\rangle \mid n(x).\mathbf{0})$ is simply equivalent to $\mathbf{0}$, because the internal communication on n cannot be observed. On the other hand, in the spi calculus, $(\nu n)(\overline{c}\langle\{m\}_n\rangle \mid c(y).case\ y\ of\ \{x\}_n\ in\ \mathbf{0})$ is not equivalent to the obvious implementation of $\mathbf{0}$, namely $\mathbf{0}$. A spi calculus process that interacts with $(\nu n)(\overline{c}\langle\{m\}_n\rangle \mid c(y).case\ y\ of\ \{x\}_n\ in\ \mathbf{0})$ can detect traffic on c, even if it cannot decrypt that traffic.

The obvious way to protect against this leak is to add noise to communication lines. In the context of the spi calculus, we may for example compose all our implementations with the noise process $!(\nu p)\overline{c}\langle\{\}_p\rangle$. This process continually generates keys and uses those keys for producing encrypted traffic on the public channel c.

In practice, since noise is rather wasteful of communication resources, and since a certain amount of noise might be assumed to exist on communication lines as a matter of course, noise is not always added in implementations. Without noise, full abstraction fails.

Trivial denial-of-service vulnerability

Consider the pi calculus process

$$(\nu n)(\overline{n}\langle m\rangle \mid n(x).\overline{x}\langle\rangle)$$

which is a small variant of the first example where, after its receipt, the message m is used for sending an empty message. This process preserves the integrity of m, in the sense that no other name can be received and used instead of m; therefore, this process is equivalent to $\overline{m}\langle\rangle$.

The obvious spi calculus implementations of $(\nu n)(\overline{n}\langle m\rangle \mid n(x).\overline{x}\langle\rangle)$ and $\overline{m}\langle\rangle$ are respectively

$$(\nu n)(\overline{c}\langle\{m\}_n\rangle \mid c(y).case\ y\ of\ \{x\}_n\ in\ \overline{c}\langle\{\}_x\rangle)$$

and $\overline{c}\langle\{\}_m\rangle$. These implementations can be distinguished not only by traffic analysis but also in other trivial ways. For example, the former implementation may become stuck when it interacts with $\overline{c}\langle p\rangle$, because the decryption $case\ y\ of\ \{x\}_n\ in\ \overline{c}\langle\{\}_x\rangle$ fails when y is p rather than a ciphertext. In contrast, the latter implementation does not suffer from this problem.

Informally, we may say that the process $\overline{c}\langle p\rangle$ mounts a denial-of-service attack. Formally, such attacks can sometimes be ignored by focusing on process equivalences that capture only safety properties, and not liveness properties. In addition, the implementations may be strengthened, as is commonly done in practical systems. For example, as a first improvement, we may add some replication to $(\nu n)(\overline{c}\langle\{m\}_n\rangle \mid c(y).case\ y\ of\ \{x\}_n\ in\ \overline{c}\langle\{\}_x\rangle)$, obtaining:

$$(\nu n)(\overline{c}\langle\{m\}_n\rangle \mid !c(y).case\ y\ of\ \{x\}_n\ in\ \overline{c}\langle\{\}_x\rangle)$$

This use of replication protects against $\overline{c}\langle p\rangle$.

Exposure to replay attacks

Another shortcoming of our implementation strategy is exposure to replay attacks. As an example, we consider the pi calculus process:

$$(\nu n)(\overline{n}\langle m_1\rangle \mid \overline{n}\langle m_2\rangle \mid n(x).\overline{x}\langle\rangle \mid n(x).\overline{x}\langle\rangle)$$

which differs from the previous example only in that two names m_1 and m_2 are transmitted on n, asynchronously. In the pi calculus, this process is equivalent to $\overline{m_1}\langle\rangle \mid \overline{m_2}\langle\rangle$: it is guaranteed that both m_1 and m_2 go from sender to receiver exactly once.

This guarantee is not shared by the spi calculus implementation

$$(\nu n)\left(\begin{array}{l}\overline{c}\langle\{m_1\}_n\rangle \mid \overline{c}\langle\{m_2\}_n\rangle \mid \\ c(y).case\ y\ of\ \{x\}_n\ in\ \overline{c}\langle\{\}_x\rangle \mid \\ c(y).case\ y\ of\ \{x\}_n\ in\ \overline{c}\langle\{\}_x\rangle\end{array}\right)$$

independently of any denial-of-service attacks. When this implementation is combined with the spi calculus process $c(y).(\overline{c}\langle y\rangle \mid \overline{c}\langle y\rangle)$, which duplicates a message on c, two identical messages may result, either $\overline{c}\langle\{\}_{m_1}\rangle \mid \overline{c}\langle\{\}_{m_1}\rangle$ or $\overline{c}\langle\{\}_{m_2}\rangle \mid \overline{c}\langle\{\}_{m_2}\rangle$.

Informally, we may say that the process $c(y).(\overline{c}\langle y\rangle \mid \overline{c}\langle y\rangle)$ mounts a replay attack. Standard countermeasures apply: timestamps, sequence numbers, and challenge-response protocols. In this example, the addition of a minimal challenge-response protocol leads to the following spi calculus process:

$$(\nu n)\left(\begin{array}{l} c(z_1).\overline{c}\langle\{m_1, z_1\}_n\rangle \mid c(z_2).\overline{c}\langle\{m_2, z_2\}_n\rangle \mid \\ (\nu p_1)(\overline{c}\langle p_1\rangle \mid c(y).case\ y\ of\ \{x, z_1\}_n\ in\ [z_1\ is\ p_1]\,\overline{c}\langle\{\}_x\rangle) \mid \\ (\nu p_2)(\overline{c}\langle p_2\rangle \mid c(y).case\ y\ of\ \{x, z_2\}_n\ in\ [z_2\ is\ p_2]\,\overline{c}\langle\{\}_x\rangle) \end{array}\right)$$

The names p_1 and p_2 serve as challenges; they are sent by the subprocesses that are meant to receive m_1 and m_2, received by the subprocesses that send m_1 and m_2, and included along with m_1 and m_2 under n. This challenge-response protocol is rather simplistic in that the challenges may get "crossed" and then neither m_1 nor m_2 would be transmitted successfully; it is a simple matter of programming to protect against this confusion. In any case, for each challenge, at most one message is accepted under n. This use of challenges thwarts replay attacks.

Leak of message equalities

In the pi calculus, the identity of messages sent on private channels is concealed. For example, an observer of the process

$$(\nu n)(\overline{n}\langle m_1\rangle \mid \overline{n}\langle m_2\rangle \mid n(x).\mathbf{0} \mid n(x).\mathbf{0})$$

will not even discover whether $m_1 = m_2$. (For this example, we drop the implicit assumption that m_1 and m_2 are different names.) On the other hand, suppose that we translate this process to:

$$(\nu n)\left(\begin{array}{l} \overline{c}\langle\{m_1\}_n\rangle \mid \overline{c}\langle\{m_2\}_n\rangle \mid \\ c(y).case\ y\ of\ \{x\}_n\ in\ \mathbf{0} \mid \\ c(y).case\ y\ of\ \{x\}_n\ in\ \mathbf{0} \end{array}\right)$$

An observer of this process can tell whether $m_1 = m_2$, even without knowing m_1 or m_2 (or n). In particular, the observer may execute:

$$c(x).c(y).([x\ is\ y]\,\overline{d}\langle\rangle \mid \overline{c}\langle x\rangle \mid \overline{c}\langle y\rangle)$$

This process reads and relays two messages on the channel c, and emits a message on the channel d if the two messages are equal. It therefore distinguishes whether $m_1 = m_2$. The importance of this sort of leak depends on circumstances. In an extreme case, one cleartext may have been guessed (for example, the cleartext "attack at dawn"); knowing that another message contains the same cleartext may then be significant.

A simple countermeasure consists in including a different confounder component in each encrypted message. In this example, the implementation would become:

$$(\nu n)\left(\begin{array}{l} (\nu p_1)\overline{c}\langle\{m_1, p_1\}_n\rangle \mid (\nu p_2)\overline{c}\langle\{m_2, p_2\}_n\rangle \mid \\ c(y).case\ y\ of\ \{x, z_1\}_n\ in\ \mathbf{0} \mid \\ c(y).case\ y\ of\ \{x, z_2\}_n\ in\ \mathbf{0} \end{array}\right)$$

The names p_1 and p_2 are used only to differentiate the two messages being transmitted. Their inclusion in those messages ensures that a comparison on ciphertexts does not reveal an equality of cleartexts.

Lack of forward secrecy

As a final example, we consider the pi calculus process:

$$(\nu n)(\overline{n}\langle m\rangle \mid n(x).\overline{p}\langle n\rangle)$$

This process transmits the name m on the channel n, which is private until this point. Then it releases n by sending it on the public channel p. Other processes may use n afterwards, but cannot recover the contents of the first message sent on n. Therefore, this process is equivalent to

$$(\nu n)(\overline{n}\langle m'\rangle \mid n(x).\overline{p}\langle n\rangle)$$

for any m'. Interestingly, this example relies crucially on scope extrusion, a feature of the pi calculus not present in simpler calculi such as CCS [Mil89].

A spi calculus implementation of $(\nu n)(\overline{n}\langle m\rangle \mid n(x).\overline{p}\langle n\rangle)$ might be:

$$(\nu n)(\overline{c}\langle\{m\}_n\rangle \mid c(y).case\ y\ of\ \{x\}_n\ in\ \overline{c}\langle\{n\}_p\rangle)$$

However, this implementation lacks the forward-secrecy property [DvOW92]: the disclosure of the key n compromises all data previously sent under n. More precisely, a process may read messages on c and remember them, obtain n by decrypting $\{n\}_p$, then use n for decrypting older messages on c. In particular, the spi calculus process

$$c(x).(\overline{c}\langle x\rangle \mid c(y).case\ y\ of\ \{z\}_p\ in\ case\ x\ of\ \{w\}_z\ in\ \overline{d}\langle w\rangle)$$

may read and relay $\{m\}_n$, read and decrypt $\{n\}_p$, then go back to obtain m from $\{m\}_n$, and finally release m on the public channel d.

Full abstraction is lost, as with the other attacks; in this case, however, it seems much harder to recover. Several solutions may be considered.

- We may restrict the pi calculus somehow, ruling out troublesome cases of scope extrusion. It is not immediately clear whether enough expressiveness for practical programming can be retained.

- We may add some constructs to the pi calculus, for example a construct that given the name n of a channel will yield all previous messages sent on the channel n. The addition of this construct will destroy the source-language equivalence that was not preserved by the translation. On the other hand, this construct seems fairly artificial.

- We may somehow indicate that source-language equivalences should not be taken too seriously. In particular, we may reveal some aspects of the implementation, warning that forward secrecy may not hold. We may also specify which source-language properties are maintained in the implementation. This solution is perhaps the most realistic one, although we do not yet know how to write the necessary specifications in a precise and manageable form.

- Finally, we may try to strengthen the implementation. For example, we may vary the key that corresponds to a pi calculus channel by, at each instant, computing a new key by hashing the previous one. This approach is fairly elaborate and expensive.

The problem of forward secrecy may be neatly avoided by shifting from the pi calculus to the join calculus [FG96]. The join calculus separates the capabilities for sending and receiving on a channel, and forbids the communication of the latter capability. Because of this asymmetry, the join calculus is somewhat easier to map to a lower-level calculus with cryptographic constructs. This mapping is the subject of current work [AFG98]; although still impractical, the translation obtained is fully abstract.

4 Full Abstraction in Context

With progress on security infrastructures and techniques, it may become less important for translations to approximate full abstraction. Instead, we may rely on the intrinsic security properties of target-language code and on digital signatures on this code. We may also rely on the security properties of source-language code, but only when a precise specification asserts that translation preserves those properties. Unfortunately, several caveats apply.

- The intrinsic security properties of target-language code may be extremely hard to discover a posteriori. Languages such as JVML are not designed for ease of reading. Furthermore, the proof of those properties may require the analysis of delicate and complex cryptographic protocols. Certifying compilers [NL97, MWCG98] may alleviate these problems but may not fully solve them.

- Digital signatures complement static analyses but do not obviate them. In particular, digital signatures cannot protect against incompetence or against misplaced trust. Moreover, digital signatures do not seem applicable in all settings. For example, digital signatures on spi calculus processes would be of little use, since these processes never migrate from one machine to another.

- Finally, we still have only a limited understanding of how to specify and prove that a translation preserves particular security properties. This question deserves further attention. It may be worthwhile to address it first in special cases, for example for information-flow properties [Den82] as captured in type systems [VIS96, Aba97, ML97, HR98].

The judicious use of abstractions can contribute to simplicity, and thus to security. On the other hand, abstractions and their translations can give rise to complications, subtleties, and ultimately to security flaws. As Lampson wrote [Lam83], "neither abstraction nor simplicity is a substitute for getting it right". Concepts such as full abstraction should help in getting it right.

Acknowledgements

Most of the observations of this paper were made during joint work with Cédric Fournet, Georges Gonthier, Andy Gordon, and Raymie Stata. Drew Dean, Mark Lillibridge, and Dan Wallach helped by explaining various Java subtleties. Mike Burrows, Cédric Fournet, Mark Lillibridge, John Mitchell, and Dan Wallach suggested improvements to a draft. The title is derived from that of a paper by Jim Morris [Mor73].

References

[Aba97] Martín Abadi. Secrecy by typing in security protocols. In *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 611–638. Springer-Verlag, 1997.

[AFG98] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. In *Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science*, June 1998. To appear.

[AG97a] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. Technical Report 414, University of Cambridge Computer Laboratory, January 1997. Extended version of both [AG97b] and [AG97c]. A revised version appeared as Digital Equipment Corporation Systems Research Center report No. 149, January 1998, and an abridged version will appear in *Information and Computation*.

[AG97b] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, pages 36–47, 1997.

[AG97c] Martín Abadi and Andrew D. Gordon. Reasoning about cryptographic protocols in the spi calculus. In *Proceedings of the 8th International Conference on Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 59–73. Springer-Verlag, July 1997.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[Bir85] Andrew D. Birrell. Secure communication using remote procedure calls. *ACM Transactions on Computer Systems*, 3(1):1–14, February 1985.

[BN95] Michele Boreale and Rocco De Nicola. Testing equivalence for mobile processes. *Information and Computation*, 120(2):279–303, August 1995.

[Bor74] Jorge Luis Borges. La muerte y la brújula. In *Obras completas 1923–1972*, pages 499–507. Emecé Editores, Buenos Aires, 1974. Titled "Death and the compass" in English translations.

[Coh97] Richard M. Cohen. Defensive Java Virtual Machine version 0.5 alpha release. Web pages at `http://www.cli.com/`, May 13, 1997.

[Den82] Dorothy E. Denning. *Cryptography and Data Security.* Addison-Wesley, Reading, Mass., 1982.

[DFWB98] Drew Dean, Edward W. Felten, Dan S. Wallach, and Dirk Balfanz. Java security: Web browsers and beyond. In Dorothy E. Denning and Peter J. Denning, editors, *Internet beseiged: countering cyberspace scofflaws*, pages 241–269. ACM Press, 1998.

[DH84] Rocco De Nicola and Matthew C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[DvOW92] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2:107–125, 1992.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, January 1996.

[FM98] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. On the Web at `http://theory.stanford.edu/~freunds/`, 1998.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, 1998.

[JL78] Anita K. Jones and Barbara H. Liskov. A language extension for expressing constraints on data access. *Communications of the ACM*, 21(5):358–367, May 1978.

[Lam71] Butler W. Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, pages 437–443, 1971.

[Lam83] Butler W. Lampson. Hints for computer system design. *Operating Systems Review*, 17(5):33–48, October 1983. Proceedings of the Ninth ACM Symposium on Operating System Principles.

[Lam89] Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[LP98] John Longley and Gordon Plotkin. Logical full abstraction and PCF. In Jonathan Ginzburg, Zurab Khasidashvili, Carl Vogel, Jean-Jacques Lévy, and Enric Vallduví, editors, *The Tbilisi Symposium on Logic, Language and Computation: Selected Papers*, pages 333–352. CSLI Publications and FoLLI, 1998.

[LR98] Xavier Leroy and François Rouaix. Security properties of typed applets. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 391–403, 1998.

[LS76] Butler W. Lampson and Howard E. Sturgis. Reflections on an operating system design. *Communications of the ACM*, 19(5):251–265, May 1976.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[McL96] John McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Transactions on Software Engineering*, 22(1):53–66, January 1996.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[Mil92] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2:119–141, 1992.

[Mil93] Robin Milner. The polyadic π-calculus: a tutorial. In Bauer, Brauer, and Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer-Verlag, 1993.

[Mit93] John C. Mitchell. On abstraction and the expressive power of programming languages. *Science of Computer Programming*, 21(2):141–163, October 1993.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 129–142, 1997.

[Mor73] James H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, January 1973.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, September 1992.

[MvOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 85–97, 1998.

[NL97] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. To appear in the proceedings of PLDI'98, 1997.

[Plo77] Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–256, 1977.

[Qia97] Zhenyu Qian. A formal specification of Java(tm) Virtual Machine instructions (draft). Web page at `http://www.informatik.uni-bremen.de/~qian/abs-fsjvm.html`, 1997.

[SA98] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 149–160, January 1998.

[Sha91] Ehud Shapiro. Separating concurrent languages with categories of language embeddings. In *Proceedings of the Twenty Third Annual ACM Symposium on the Theory of Computing*, pages 198–208, 1991.

[Sun97a] Sun Microsystems, Inc. Inner classes specification. Web pages at `http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/`, 1997.

[Sun97b] Sun Microsystems, Inc. RMI enhancements. Web pages at `http://java.sun.com/products/jdk/1.2/docs/guide/rmi/index.html`, 1997.

[vDABW96] Leendert van Doorn, Martín Abadi, Mike Burrows, and Edward Wobber. Secure network objects. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 211–221, May 1996.

[VIS96] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4:167–187, 1996.

[WABL94] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, February 1994.

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. *Computing Systems*, 9(4):265–290, Fall 1996.