

Object-Oriented Security Proofs

Ernie Cohen

Amazon Web Services

Abstract. We use standard program transformations to construct formal security proofs.

1 Security as Object Equivalence

A *security specification* is a functional specification that additionally bounds information flow. For example, a functional specification for a communication channel might say that messages are received in the order sent; its security specification might additionally say that sending a message can leak to an adversary only the length of the message, not its bits. Such precise specification allows the proof of useful security guarantees for practical designs under realistic environmental assumptions. Security proofs most commonly appear in the context of cryptographic protocols, but can handle a variety of security enforcement mechanisms, such as security policy checks, virtualization, and human operations.

We formalize a specification as a constructor producing fresh objects exhibiting permitted behaviors. The constructor parameters can include objects representing components of the (possibly adversarial) environment, providing to the constructed object both services to call and a way to explicitly leak information. For example, one possible ideal channel specification maintains the history of messages sent as a private field, with a send method that leaks to the adversary the length of the message, but not its bits. As demonic nondeterminism in the language would provide to an attacker an NP oracle, breaking all reasonable cryptographic assumptions, methods also delegate to the environment the resolution of nondeterminism. For example, the receive method of the channel would ask the environment to decide whether to fail the call or to return the next undelivered message from the history.

A specification $C(_)$ *refines* a specification $A(_)$ iff there is a function f such that $C(x)$ and $A(f(x))$ are equivalent. Typically, $A(x)$ will be an abstract specification of a service that describes security intent, while $C(x)$ is a more concrete implementation of the service. This reduces security reasoning to object equivalence (or, more precisely, equivalence of two constructor calls).

This approach, a flavor of universal composability, bears some similarity to methods based on process algebra or higher-order functional programming, but has some practical advantages. One advantage is that object identity makes it easy to write first-order invariants about the program state. Another is that, unlike spi calculi, there are no cryptographic security assumptions built into the logic; they are formulated directly as equivalence assumptions.

2 Reasoning About Object Equivalence

What does it mean for two objects to be equivalent? A natural definition is for every language-definable Boolean function of a single object reference to return the same value when called on the two objects. In a probabilistic language, the two function calls would have to have the same probability of returning true. This definition of equivalence, *semantic equivalence*, is suitable for constructions that do not use cryptography.

The basic tools for reasoning about semantic equivalence are mostly familiar ones used for modular reasoning about object-oriented programs. These include inlining method calls, introducing/eliminating auxiliary variables, type invariants, admissible object invariants, ownership (to justify admissibility of invariants depending on the state of owned objects), sequential code equivalence (leveraging object invariants), and simulation. To simplify such reasoning, it is useful to guarantee that a method call cannot result in a callback, so that we do not have to worry about changes to the local object state when making a method call. We achieve this by (1) requiring an object calling a method to hold a “reference” to the object (in addition to its address), (2) allowing only constructors to create new references, and (3) allowing references to be passed into constructors and returned from method calls, but not passed into method calls. In the absence of infinite loops/recursions within a single object, this discipline also guarantees that all method calls terminate.

Cryptography requires a more permissive and complex equivalence: *indistinguishability*. Indistinguishability assumes that objects are additionally parameterized by a security parameter. It requires that for every probabilistic function running in time polynomial in the security parameter, and given one of the two objects with equal probability, the advantage (probability in excess of 0.5) of the function guessing which of the two objects it is interacting with is negligible in the security parameter. (A real-valued function $f(n)$ is *negligible* iff for every polynomial $p(n)$, $\lim_{n \rightarrow \infty} (p(n) \cdot f(n)) = 0$.) Fortunately, indistinguishability is in practice not very different from semantic equivalence. It adds the side condition that the environment runs in probabilistic polynomial time. It also provides additional assumptions, e.g., that a uniformly chosen random number, of length linear in the security parameter, chooses a value not in any previously constructed set of polynomial size.

Our *simulations* are program terms that transform one state representation to another, with the usual forward simulation condition (simulation followed by a method of the first representation is equivalent to a method call of the second representation followed by simulation). This formulation is independent of semantic details of the language and the equivalence. In particular, it allows probabilistic simulations, which often eliminate explicit probability reasoning.

In a concurrent system, we want to allow internal steps, as many internal interactions with the environment might be required between successful external interactions. We do this by providing an additional method that performs an internal step. The method queries the adversary to choose which internal action to perform, performs the chosen action, and returns the result to the adversary.

Timing side channels are easily modeled by leaking the time required to perform an operation. This also allows us to reason about transition systems.

A typical cryptographic assumption is that a concrete encryption functionality, which encrypts with a randomly generated key, is indistinguishable from an ideal encryption functionality (which zeroes message bits before encryption, but remembers the original plaintext). Proofs often revolve around transforming an implementation to encapsulating a key within a concrete encryption functionality, so as to idealize it. If that ideal functionality is used to encrypt other keys, it can be transformed, through simulation, to instead map ciphertexts to concrete encryption functionalities encapsulating the encrypted keys, allowing them to be idealized also. Avoidance of key cycles thus arises naturally through the order in which encapsulations are done in the proof.

An important consideration in reasoning about key compromise using this style of proof (as opposed to direct game arguments) is the so-called *commitment problem*: when an encryption key is compromised, we can no longer justify the prior pretense that encryptions carried no information about the messages sent. Fortunately, it is still sound to use the ideal functionality to prove *safety* properties of the whole system that hold up to the point of key compromise. This allows the proof of properties like perfect forward secrecy.

3 Some Specification Examples

We write specifications in a simple, untyped, object language, where all values (including addresses) are bitstrings, fields are private, and method bodies are expressions. Calling a method that doesn't exist simply returns 0 (the empty string). We use \sim to denote indistinguishability on program terms. $Z(m)$ is the bitstring m with all bits replaced with 0's, and $\&$ is the C `&&` operator.

As a simple example, here is a possible communication channel specification:

```
ChI(n) { new (n,s=0,r=0,h=0) {
    snd(m) { m & n.snd(Z(m)), h[s++] = m }
    rcv()  { n.rcv() & r<s & h[r++] }
}}
```

This defines a function `ChI` that returns the address of a freshly constructed object, with fields `n` (initialized to the parameter `n`), `s`, `r`, and `h` (each initialized to 0). These fields give the address of the environment/adversary object, the number of messages sent, the number of messages received, and the sequence of messages sent (represented as a sparse map). Since we use 0 to represent failure, messages must be nonzero. We leak to the adversary the length of the messages sent, and allow it to fail reception. If communication was to be only authenticated, `m` itself would be leaked instead of `Z(m)`. We could allow internal steps by adding the method `step()` { `n.step()` }.

`ChI` can be viewed as turning an arbitrary (insecure) service `n` into a secure, FIFO service. Since `ChI` is idempotent (i.e., $\text{ChI}(\text{ChI}(n)) \sim \text{ChI}(n)$), we say an

expression t returns a fresh, asymptotically computationally secure channel iff $\text{ChI}(t) \sim t$.

We can implement ChI using AEAD (authenticated encryption with associated data) as follows:

```

AEAD(n) { new (n,d=0) {
    enc(a,m) { m & c = n.enc(a,Z(m)) & d[a][c] = m & c }
    dec(a,c) { d[a][c] }
}}

ChC(n,e) { new (n,e,s=0,r=0) {
    snd(m) { m & n.snd(e.enc(s++,m)), m }
    rcv() { m = e.dec(r, n.rcv()) & r++ & m }
}}

```

The correctness of this construction is given by the following theorem. For reasoning purposes, it matters little what the term inside ChI is:

Theorem: $\text{ChC}(n, \text{AEAD}(e)) \sim \text{ChI}(\text{ChC}(n, \text{AEAD}(e)))$

A typical AEAD implementation uses a uniformly chosen symmetric key k :

```

Enc(k) { new (k) {
    enc(a,m) { skEnc(k,a,m) }
    dec(a,c) { skDec(k,a,c) }
}}

```

where $\text{rnd}()$ chooses uniformly a bitstring of length given by the security parameter. (The key is a parameter so that we can replace $\text{rnd}()$ with any expression indistinguishable from $\text{rnd}()$, such as the output of a key derivation function or a pseudorandom generator.) This gives a computationally secure implementation, i.e., $\text{ChC}(n, \text{Enc}(\text{rnd}())) \sim \text{ChI}(\text{ChC}(n, \text{Enc}(\text{rnd}())))$.

4 Conclusion

We have used the methodology to formally verify several security properties of industrial protocols, including the following:

- We proved that a simple, TLS-like shared-key ciphersuite communication protocol provides a factory for secure communication channels.
- We proved that a distributed system design (using symmetric and asymmetric encryption, unauthenticated Diffie-Hellman agreement, KDFs, public-key signatures, envelope encryption, multiple encryption domains with mutually distrusting quorums, dynamically changing domain operator/server memberships, and domain key rotation) provides an ideal encryption service.

Current work includes mechanization and proofs with concrete security bounds.

We thank Supriya Anand, James Bornholdt, Matt Campagna, Byron Cook, Andres Erbsen, Rustan Leino, Andrea Nedic, Jade Philipoom, and Serdar Tasiran for their contributions.