Reliable Evidence: Auditability by Typing

Nataliya Guts[1], Cédric Fournet[2,1], and Francesco Zappa Nardelli[3,1]

[1] MSR-INRIA Joint Centre
[2] Microsoft Research
[3] INRIA

Abstract. Many protocols rely on audit trails to allow an impartial judge to verify *a posteriori* some property of a protocol run. However, in current practice the choice of what data to log is left to the programmer's intuition, and there is no guarantee that it constitutes enough evidence. We give a precise definition of auditability and we show how typechecking can be used to statically verify that a protocol always logs enough evidence. We apply our approach to several examples, including a full-scale auction-like protocol programmed in ML.

1 A Language-Based Approach to Auditing

Consider a simple protocol where a client $\mathcal{A}$ sends an authenticated mail to a server $\mathcal{B}$. To prove her identity, $\mathcal{A}$ signs the message using her secret signing key and appends the signature to the message:

$$\mathcal{A} \longrightarrow \mathcal{B} : text, sign(secret\_key(\mathcal{A}), text)$$

Intuitively, this protocol guarantees the authenticity of the message sent by $\mathcal{A}$. The server $\mathcal{B}$ can verify the signature using $\mathcal{A}$'s public key and, if the test succeeds, $\mathcal{B}$ can be sure of the authenticity of the message. But, in case of dispute between $\mathcal{A}$ and $\mathcal{B}$, does $\mathcal{B}$ possess enough evidence to prove authenticity to a third party?

We say that a protocol is *auditable* with respect to a property if it logs enough evidence to convince an *impartial* third party, called a *judge*, of that property.

In our example, $\mathcal{A}$'s text and signature, if securely stored by $\mathcal{B}$, constitute sufficient evidence for auditing. Later, a judge can take a decision upon verifying the signature and, inasmuch as all principals agree on the public key infrastructure for signing, they also agree that this judge is impartial. Note that the signature alone may not constitute sufficient evidence: a careless server that discards or alters the received text would not be able to convince the judge.

Suppose now that, instead of signing the text, $\mathcal{A}$ signs a fresh key k, encrypts it under $\mathcal{B}$'s public key, and encrypts the text under k using non-malleable encryption. In this case, $\mathcal{B}$ can decrypt and authenticate the key k, then decrypt the message, and infer the authenticity of $text$. However, an impartial judge cannot attribute the message to $\mathcal{A}$, since both $\mathcal{B}$ and $\mathcal{A}$ are able to encrypt data using the key k; the authenticity of $text$ for $\mathcal{A}$ is not auditable. (For mail, this feature is often called *deniability* [Roe97].)

The concept of auditability is entangled with the figure of the judge. A judge is an entity that evaluates if some evidence enforces a given property, in an impartial and

M. Backes and P. Ning (Eds.): ESORICS 2009, LNCS 5789, pp. 168–183, 2009.

transparent manner. Thus, its decision procedure must be relatively simple, and it must be known and accepted *a priori* by all principals concerned by the auditing.

Similarly, fair non-repudiation protocols rely on trusted third parties (TTPs): for each message, evidence of its origin and receipt of its dispatch is collected by the participants and this evidence can be passed to the TTP to resolve disputes [KMZ02]. Judges are similar to offline TTPs: they are invoked *a posteriori*, only when necessary. Nevertheless judges never issue their own signatures (unlike transparent TTPs), nor participate in the protocol.

Auditing is an essential component of secure distributed applications, and the use of audit logs is strongly recommended by security standards [Pub96, ISO04]. In practice, most applications selectively store information of why authorisations were granted or why services were provided in *audit logs*, with the hope that this data can be later used for regular maintenance, such as debugging of security policies, as well as conflict resolution. However, deciding which evidence should be logged to enable reliable and efficient auditing is left to the programmer's intuition. As shown above, it is not the case that all properties that can be verified by a principal at run-time can be audited by an external judge. Even considering only properties that can be audited, it is unclear if some given evidence enforces them. Besides, extensive logging may conflict with other security goals, such as confidentiality and privacy.

The first contribution of this paper is a formal definition of auditable properties (Sections 2 and 3). We aim to verify concrete protocol implementations, rather than their abstract models, so we represent protocols as programs written in F# (a dialect of ML) and we specify their properties using logical formulas. Judges are represented as trusted F# functions; a new language primitive marks data proposed as evidence for some property.

The second contribution is a method for verifying that some collected evidence suffices to prove a property to a given judge (Section 4). Our method relies on refinement types, and uses F7 (an extended type-checker for F# [BBF$^+$08]) to statically verify that a property is auditable. Our approach is tested against several sample protocols, including a realistic multiparty partial-information game programmed in F# (Section 5). A companion paper, the source code for all our protocols, and additional examples are available from http://www.msr-inria.inria.fr/projects/sec/logs.

2 Modelling Security Protocols in F7

We build on F7 [BFG08], an existing tool for verifying safety properties in F# programs, and on RCF [BBF$^+$08], its formal core language. RCF is a typed concurrent call-by-value lambda calculus with an F# syntax. We only recall its syntax and informal semantics, and refer to earlier work for a complete definition.

Values, denoted by M, include names, variables, functions, pairs, constructed values and unit. Expressions, denoted by A and B, include a standard functional core: values, application, syntactic equality, let binding, pattern matching; and some concurrent constructs: name restriction, $(\nu a)A$, parallel composition of threads, $A \Rsh B$, asynchronous send of a message N over a channel M, *send* M N, and reception of a message over a channel M, *rcv* M. In addition, expressions include logical annotations. We let C

range over formulas in a first-order logic that includes predicates over values. Formulas can be assumed (denoted **assume** C) or asserted (denoted **assert** C) by programs. (Free variables of a term M are denoted $fv(M)$.)

An expression represents a concurrent, message-passing computation, which may return a value. The state of the computation can be represented as an expression in normal form that includes (1) a multiset of assumed formulas; (2) a multiset of pending messages; and (3) a multiset of expressions being evaluated in parallel. We use F to range over the multiset of assumed formulas.

The reduction semantics is defined in terms of a small-step relation over configurations. It contains the usual β-reduction, pattern-matching reduction and communication reduction, closed under evaluation contexts E, defined as

$$E ::= [\_] \mid \textbf{let } x = E \textbf{ in } A \mid (\nu a : T)E \mid E \Rsh B \mid B \Rsh E]$$

The evaluation of **assume** C extends the current multiset of assumed formulas with C. Informally, **assume**s are privileged expressions, recording for instance that a principal intends to send a message. Conversely, **assert** records that a principal believes that some logical property holds at this point. We say that **assert** C *succeeds* if, when it is evaluated, the formula C is deducible from the assumed formulas F, denoted $F \vdash C$. For example, the assert in the expression **assume** C; **assert** C always succeeds. The **assume** and **assert** expressions always reduce to unit: their role is to specify, rather than to enforce, run-time properties of a program.

Protocols and roles as programs. A protocol can be written in F# as a collection of functions that represent compliant code for the different roles, possibly sharing some variables (such as cryptographic keys). This collection of functions and variables can be structured into modules; the module interfaces are then made available to the environment, which can run, and interact with, the roles. The environment models an active attacker; it is *a priori* untrusted and should not access some of the shared variables (such as private keys). In F# the visibility of variables is specified in typed interfaces (as done in Section 4), but, for clarity, in this section we do not use types and rely instead on some syntactic sugar.

A *protocol*, denoted $\mathcal{L}$, is a context that defines *public* and *private* let bindings:

$$\mathcal{L} = \textbf{let } a = A \textbf{ in } \mathcal{L} \mid \textbf{private let } a = A \textbf{ in } \mathcal{L} \mid [\_]$$

Let $private(\mathcal{L})$ (resp. $public(\mathcal{L})$) be the set of variables declared in $\mathcal{L}$ with (resp. without) the **private** prefix. An *opponent*, denoted O, is an expression that does not contain any **assert** (and **audit**, defined later) and whose free variables cannot be bound to variables declared as **private**. A *program* is a closed expression of the form $\mathcal{L}[O]$ where $\mathcal{L}$ defines the global variables and roles and O is an opponent (as such, it holds that $private(\mathcal{L}) \cap fv(O) = \emptyset$).

To illustrate our setup, we program the authenticated mail of Section 1 relying on RSA public-key signatures. In the code below, we omit the trusted libraries *Crypto* and *Net* that define functions such as *sigkey*, *verifkey*, *rsasha1*, and *verify_sig*. (Following the ML syntax, we omit **in** between top-level definitions.)

```
private let seed = rsaKeyGen ()
private let ska = sigkey seed
let pka = verifkey (rsaPub seed)

let princA () =
  let text = "Hey" in
  assume (Send("A",text));
  send c (text, rsasha1 ska text)

let princB () =
  let text,sign = recv c in
  if verify_sig pka text sign
  then assert (Send("A",text));
  text
```

The code first defines the secret key *ska* and the verification key *pka* for principal $\mathcal{A}$. The secret key is declared **private**, to prevent the environment to sign messages. The principal $\mathcal{A}$, implemented by *princA*, creates a signed message and sends it over the channel c. The principal $\mathcal{B}$, implemented by *princB*, receives the message and its signature, and verifies if the signature is valid for the message issued by $\mathcal{A}$. We call the protocol above $\mathcal{L}_{mail}$ (we omit the standard library modules $\mathcal{L}_{Crypto}$ and $\mathcal{L}_{Net}$ it depends on). The predicate *Send*(*a*,*x*) encodes at the logical level that the principal *a* sent the message *x*. Since the principal $\mathcal{A}$ is compliant, all the other participants trust her to add *Send*("A","Hey") to the set of valid formulas using the **assume** primitive. If the signature verification succeeds, then the server can expect this property to hold, which is specified by asserting it.

Pinpointed expressions. To formalise auditability we need to track precisely the substitutions that are applied to some sub-expressions of a program. Technically, we extend the syntax of expressions with *pinpointed expressions*, denoted $\underline{A}\sigma$, where σ is a finite substitution of values for variables. The definition of substitution used for evaluation is then modified to extend σ rather than propagate through A:

$$(\underline{A}\sigma)\{M/x\} = \underline{A}(\sigma;\{M/x\}) \ .$$

Once a pinpointed expression gets in head position inside an evaluation context, the deferred substitution σ is applied to A, resuming the computation via the rule $\underline{A}\sigma \rightarrow A\sigma$. Just before this reduction, σ contains exactly the substitutions applied by the context to the sub-expression A. It is easy to see that the expression A and the expression obtained by replacing a sub-expression A' of A with $\underline{A'}$ (a pinpointed expression with an empty substitution) reduce to the same value.

Safety and Robust Safety. A program is *safe* if, in all evaluations all its assertions succeed [BBF+08]. We recast this definition using pinpointed assertions:

Definition 1. *The formula C is* safe *in the program $A[\textbf{assert}\ C]$ when, for all reductions*

$$A[\underline{\textbf{assert}\ C}] \rightarrow^* E[\underline{\textbf{assert}\ C\sigma}]$$

where E is an evaluation context with assumed formulas F, we have $F \vdash C\sigma$. A program is safe *when all its assertions are safe. A protocol $\mathcal{L}$ is* robustly safe *if, for all opponents O, the program $\mathcal{L}[O]$ is safe.*

Note that when **assert** is evaluated the substitution σ records the actual values for the free variables of the formula C.

For example, using the protocol $\mathcal{L}_{mail}$, the program $\mathcal{L}_{mail}[\mathit{princA}\ () \Rsh \mathit{princB}\ ()]$ is safe. The only occurrence of **assert** is in the code of *princB*, and it is evaluated after reception on channel c. Only *princA* sends a message on c, with content `"Hey"`, and only after assuming $\mathit{Send}(\texttt{"A"},\texttt{"Hey"})\{\}$. These reductions lead to a configuration

$$E\left[\textbf{assert}\ (\mathit{Send}(\texttt{"A"},\mathit{text}))\{^{\texttt{"Hey"}}/_{\mathit{text}}\}\right]$$

with multiset of assumed formulas $F = \{\mathit{Send}(\texttt{"A"},\texttt{"Hey"})\}$, so we trivially have $F \vdash \mathit{Send}(\texttt{"A"},\texttt{"Hey"})$. More interestingly, $\mathcal{L}_{mail}$ is also robustly safe. Since robust safety quantifies over all environments that interact with the protocol, we might imagine a malicious opponent that after launching *princB* sends the message `("A","Hey")` over the public channel c. However, the signature verification performed by $\mathcal{B}$ guarantees that the message received on the public channel c has been sent by $\mathcal{A}$, and in turn that the formula $\mathit{Send}(\texttt{"A"},\texttt{"Hey"})$ has been previously assumed.

3 A Definition of Auditability

Informally, a program is *auditable* if, at any audit point, an impartial judge is satisfied with the evidence produced by the program.

We extend RCF with the primitive **audit** $C\ L$. This allows the programmer to specify the program points that require auditing for property C, using the value L as evidence. In practice, although we do not enforce it, the evidence produced by L should be safely logged by the program. Similarly to **assert**, this primitive plays a role only in the specification of properties: **audit** $C\ L$ always reduces to unit.

To simplify the presentation, we focus on programs with a single audited property, a single judge, and a single audit request point. Let C be this property, and suppose that $\mathit{fv}(C) = \tilde{x}$. Our definitions generalise easily to several distinct properties and audit requests, possibly sharing the same judge.

We represent the judge as a function, named *judge*, taking as arguments the actual values of the free variables of C and the evidence, and evaluating a boolean expression J that computes the judge's decision. The judge function in a protocol should be defined by a public binding of the form **let** $\mathit{judge}\ \tilde{x}\ e = J$. For sanity, we require that J does not assume any property or access any private binding of the protocol.

Auditability for the authenticated mail. In the introduction we suggested that in the authenticated mail example the property $\mathit{Send}(\texttt{"A"},\mathit{text})$ is not only safe but also auditable. For a given text sent by the client (e.g. `"Hey"`), the associated signature constitutes the evidence to enforce the property $\mathit{Send}(\texttt{"A"},\mathit{text})\{\texttt{"Hey"}/\mathit{text}\}$. We can then replace the **assert** $(\mathit{Send}(\texttt{"A"},\mathit{text}))$ executed by *prinB* with the audit request **audit** $(\mathit{Send}(\texttt{"A"},\mathit{text}))\ \mathit{sign}$. In this example the PKI is trusted by all participants: a judge that, given a text and a signature, returns **true** if and only if the signature is valid can be deemed impartial (or *correct*). Observe that the signature always suffices to convince the judge: we say that it constitutes *complete* evidence.

The key property that distinguishes auditing from asserting properties, is that the judge can be called in any context where the public key of the client is known: for instance, a third party can invoke the judge to confirm the outcome of the transaction.

We can update the code of the authenticated mail protocol and add the definition of the judge.

```
let judge text e =
  verify_sig pka text e
```

```
let princB () =
  let text,sign = recv c in
  if verify_sig pka text sign then
    audit (Send("A",text)) sign;
    send d (text,sign); text
```

The judge function just validates the signature passed in as evidence. As discussed above, it is correct for the property *Send*("A",*text*). The **audit** (*Send*("A",*text*)) *sign* statement executed by *princB succeeds* if the evidence *sign* suffices to convince the judge, as is the case here. Thus, the property *Send*(*a*,*x*) is *auditable* in this example. The principal *princB* then publishes the evidence on the channel *d*.

Auditability, formally. Given a program $\mathcal{L}[O]$, we rewrite it as a two-hole context applied to the body J of the judge (**let** *judge* $\tilde{x}$ e = J) and to the evidence L provided in the audit statement. With a slight abuse of notation we denote it as $A[J, L]$. Our definition says that a (well-formed) program is *auditable for a property* C if it defines an impartial judge for C (correctness), and if the evidence provided in the audit call suffices to convince the correct judge of the validity of the property (completeness).

Definition 2. *Let* $\mathcal{L}$ *be a protocol with a (public) declaration* **let** *judge* $\tilde{x}$ e = J *and a statement* **audit** C L *in its scope. Let* O *be an opponent. Let* A *be a two hole context such that* $A[J, L] = \mathcal{L}[O]$. *The program* $\mathcal{L}[O]$ *is* auditable *when*

(Well-formedness) *(a) the declared variables of* $\mathcal{L}$ *are not rebound; (b)* J *and* L *do not contain* **assume***s. (c)* $fv(J) \cap private(\mathcal{L}) = \emptyset$*;*

(Correctness) *if* $A[\underline{J}, L] \rightarrow^* E[\underline{J}\sigma]$ *for some evaluation context* E *with assumed formulas* F*, and* $J\sigma \rightarrow^*$ **true***, then we have* $F \vdash C\sigma$*; and*

(Completeness) *if* $A[J, \underline{L}] \rightarrow^* E[\underline{L}\sigma]$ *for some evaluation context* E*, then we have (***let** e = L **in** $J)\sigma \rightarrow^*$ **true**.

The protocol $\mathcal{L}$ *is* auditable *when the program* $\mathcal{L}[O]$ *is auditable for all opponents* O.

Let us illustrate the definition above for the authenticated mail protocol, with some opponent code that receives the audit evidence on channel *d* then invokes the judge:

$$\mathcal{L}_{mail}[\mathit{princA}\ ()\ \Rsh\ \mathit{princB}\ ()\ \Rsh\ (\textbf{let}\ \mathit{text},\mathit{e} = \mathit{recv}\ d\ \textbf{in}\ \textbf{if}\ \mathit{not}\ (\mathit{judge}\ \mathit{text}\ e)\ \textbf{then}\ \texttt{"bad"})]$$

With this particular opponent, the judge is called after the server successfully completes, and thus after the client's **assume**, so the judge is *correct* when it returns **true**. The evidence is also *complete*: at the audit point, if we pass the actual evidence to the judge we get

$$(\textbf{let}\ e = \mathit{sign}\ \textbf{in}\ \mathit{verify\_sig}\ \mathit{pka}\ \mathit{text}\ e)\sigma$$

for some substitution σ that substitutes `"Hey"` for *text*, the result of *rsasha1 ska text* for *sign*, a cryptographic function for *verify_sig*, and a matching keypair for *ska* and *pka*. This expression reduces to **true** by the definition (and the F# implementation) of the verification of asymmetric signatures.

In some cases the conditions required for correctness can be trivially satisfied. A judge that always returns false is correct; however in this case no evidence can satisfy the judge, and thus the protocol cannot be complete. Also, if the judge is not called, then correctness is vacuously satisfied. Correctness and completeness are complementary properties: giving evidence to an unreliable judge makes no sense, nor does conducting a trial with insufficient evidence. Note that a judge is correct if and only if it is safe to assert the audited property whenever the judge returns true.

Opponents and partial compromise. The environment O models a potentially hostile attacker, which can access all public values and roles of the protocol, and control public communications. In addition, an attacker may corrupt a subset of the principals to gain access to their private resources (like signing keys). Interestingly, in this case the remaining compliant principals may remain auditable: a signature by a principal, compliant or not, constitutes audit evidence.

Compromised participants can be represented in our setting by extending the protocol with definitions that export their private resources. Suppose that, in the authenticated mail example, $\mathcal{A}$ is compromised. Its secret key becomes public, and the code below is added to the end of the protocol:

let *leaked_key* = **assume** ($\forall x$. *Send*("A",*x*)); *ska*

The attacker can now choose any message and sign it with $\mathcal{A}$'s signature:

send c ("Bleah", *rsasha1 leaked_key* "Bleah")) ↱ *princB* ()

The compromise of $\mathcal{A}$must be reflected in the logical world. The meaning of the formula *Send*("A",*x*) was that "principal $\mathcal{A}$ sent message *x*", and it was possible to certify this action by verifying the relevant signature. However, now arbitrary messages sent by the attacker can be signed with $\mathcal{A}$'s key. The **assume** ($\forall x$. *Send* ("A",*x*)) evaluated just before exporting the private key of $\mathcal{A}$ captures this fact. In general, before exporting the private resources of a compromised participant, it is necessary to "saturate" all the properties related to the compromised participant, as done here (in a modal logic, this would be equivalent to assuming the formula $\mathcal{A}$ **says false** [FGM07]). Observe that, in a protocol run where $\mathcal{A}$ was compromised and the environment issued the attack above, if an audit for the property *Send*("A","Bleah") is requested, then the server can still provide enough evidence to the judge: the protocol is still auditable.

An attacker might also invoke directly a judge and provide some bogus evidence to accuse a compliant principal. However, Definition 2 states that a judge is correct only if it always takes the right decision, independently of the origin of the evidence. So, this attack is deemed to fail.

Auditable properties. Even if typical evidence includes some collection of signed data, the judge does not necessarily rely on cryptography. To audit the arithmetic property "$2^n - 1$ is not prime", with two integers as evidence, a correct judge simply checks that these integers are greater than 1 and their product is equal to $2^n - 1$. Similarly, if an access control database is trusted by the judge and by all principals, then the compliance of granted or denied accesses can be verified against the corresponding database entries, and no evidence must be provided.

Some properties, like deniable authentication in the second example of Section 1, cannot be audited. In general, all deniable properties are not auditable, and all auditable properties are undeniable (luckily properties enforced by most of the protocols are neither deniable nor undeniable). Privacy constraints might also prevent auditing: if x is secret, then a property C where x appears as cleartext in the evidence cannot be audited.

Datatypes that guarantee audit properties. We previously showed that it is possible to audit cheating (write-after-commit attacks) on an implementation of write-once cells [FGZN08]. We can prove that the described distributed protocol that globally compares log entries behaves as a correct judge, and that the information stored in the distributed log constitutes complete evidence.

4 Static Analysis of Auditability

In the previous sections we relied on **assume**, **assert**, and **audit** statements to relate the states of a program to logic formulas. In this section we describe how the refinement types of RCF and the associated typechecker for ML, called F7 [BFG08], can be used to statically verify the correctness of the judge and the completeness of the evidence.

Review of refinement types. Refinement types associate logical formulas with program expressions: the type of an expression A is of the form x: T { C } where x binds the value of A, T is a type being refined (e.g. an ordinary ML type), and C is a formula that holds when A returns (e.g. a property of x).

In the mail example, the type *x*: *string* {*Send*("A",x)} is inhabited by all strings M such that the property *Send*("A",M) follows from the assumed formulas. So, the string "Hey" sent by $\mathcal{A}$ has this type, since the property follows from the preceding **assume**. The string returned by $\mathcal{B}$ also has this type: the signature verification ensures that the property follows from the preceding **assume**.

Refinements that appear in the arguments of a function specify preconditions that must hold when the function is invoked, while the refinement of the return type specifies a postcondition that will hold when the function returns. Hence, the role *princB* is a function that can be typed as *unit* $\rightarrow$ $text$:*string*{*Send*("A",$text$)}: no preconditions are required to run *princB*, and the returned string *text* will satisfy *Send*("A",*text*). Although all formulas in our examples so far are just facts (representing protocol events), in general formulas also include policy rules. Consider, for instance, a variant of our example where $\mathcal{B}$ also enforces an authorisation policy after authenticating the message: a message may be forwarded to a mailing list only if the sender is a member of that list:

assume ($\forall x,t,l.$ (*Send*(x, t) $\wedge$ *CanPost*(x,l)) $\rightarrow$ *Post*(l,t)

and we may type *princB* as

unit {*CanPost*("A","comp.risk")} $\rightarrow$ *text*:*string* {*Post*("comp.risk",*text*)}

Now *princB* can only be invoked in a context where *CanPost*("A","comp.risk") holds.

An **assert** *C* statement is well-typed in a typing environment where *C* logically follows from the formulas of the environment. Conversely, an **assume** *C* statement is always well typed, with *C* as a postcondition.

Typing cryptography. The library *Crypto* in the F7 distribution provides a refinement-typed symbolic implementation for standard cryptographic functions. In particular, the types for public-key signature operations let us specify matching logical conditions between signers and verifiers that exchange messages over some untrusted channel, used as preconditions before signing, and as postconditions after signature verification. Let *payload* be a plain ML type (without refinement formula) that expresses the structure of a message. Let *signed* abbreviate the refinement type p:*payload* $\{C\}$ for some formula C. The functions for signing and verifying payloads can be typed as:

val *rsasha1*: *signed sigkey* → *signed* → *dsig*
val *verify_sig*: *signed verifkey* → *p*:*payload* → *dsig* → *b*:*bool* { *b*=**true** ⇒ C }

The type *dsig* is the type of signatures. The types *signed sigkey* and *signed verifkey* are the type of keys used to compute and verify signatures for values of type *signed*. The function *rsasha1* computes a signature of a *payload* value that satisfies the precondition C. The function *verify_sig* takes as parameter a verification key, a *payload* value, and a signature; it dynamically checks whether this is a valid signature for that value and returns the Boolean outcome. The postcondition states that, if the verification succeeds, then property C holds for p, hence that p can be given the more precise refinement type *signed*. Informally, this postcondition is correct if all signers are also well-typed and the signature scheme is cryptographically secure.

Typing opponents. The opponents that interact with a protocol do not have to be well-typed: they are untrusted and we should not (artificially) limit their power.

To this end, the type system has a universal type [Aba99], written *Un*, to represent data that may flow to and from the opponent. We recall below the main type safety theorem.

Theorem 1 ([BBF+08]). *If* $\emptyset \vdash \mathcal{L}[public(\mathcal{L})]$: *Un, then* $\mathcal{L}$ *is robustly safe.*

The typing judgment $\Gamma \vdash A$ states that expression A is well-typed in the typing environment Γ. The intuition is that Γ safely approximates the set of formulas that hold whenever A is evaluated. Thus, if A contains well-typed asserts, then these asserts will succeed in all executions of A. In the theorem, since $\mathcal{L}[public(\mathcal{L})]$ is typed as *Un*, each of its publicly declared expressions must also have type *Un*, and for any opponent O we also have $\emptyset \vdash \mathcal{L}[O]$: *Un* guaranteeing that $\mathcal{L}[O]$ is safe.

With the theorem above, we can show that our authenticated email protocol is robustly safe: (1) we type it, in particular for cryptography by instantiating *payload* to *string* (the type of *text*) and *signed* to *text*:*string* {*Send*("A",*text*)}; and (2) we check that all its variables exported to the environment have a public type. Both checks are automatically performed by the F7 typechecker.

Auditability via typechecking. We show that types can also be used to statically verify the auditability of a property in a well-formed protocol (Definition 2). This relies on being able to assign (and verify) precise types to the judge function and to the functions it uses. We first discuss correctness for the judge, then completeness for the evidence.

Correctness. A judge is a public function that returns a boolean value. The untrusted environment should be able to call it, so its arguments should have type *Un*. (In particular, the evidence values themselves are not trusted until they are verified by the judge.)

The correctness condition says that the judge returns **true** only when the target audited property holds; this can be expressed as a post-condition on its result. We obtain the following type declaration for the judge:

val *judge*: $\tilde{x}$: $\tilde{Un} \rightarrow e$:*Un* $\rightarrow b$:*bool* { b=**true** $\Rightarrow C$}

and every expression that can be given this type is a correct judge function.

Completeness. Definition 2 states that some evidence is complete for an audit request if a call to the judge in the same context and with the same evidence returns **true**. This requires that: (1) the judge terminates, and (2) if the judge terminates, it returns **true**.

Termination of the judge function must be proved manually. Termination is hard to prove in general, but pragmatically we limit ourselves to judges that are sequences of calls to deterministic functions that terminate unconditionally: either non-recursive functions, or recursive functions of linear-time complexity (e.g. cryptographic functions) in their inputs. So termination is not a real issue.

We must then show that the context of every **audit** provides enough guarantees on the gathered evidence to ensure that the judge returns **true**. This amounts to writing a *success condition* for the judge; typechecking is then used to verify that the condition holds at every audit point. We emphasise that these annotations need not be trusted, as their correctness is checked by typing.

Typically a judge is a sequence of verifications: its success condition is the conjunction of the success condition for each of them. We need some additional refinements for public-key signatures, so that typechecking guarantees the success of future signature verifications once a signature has been verified. We introduce a predicate *IsDsig* $(vkey, p, sg)$ where $vkey$, p, and sg are of type *signed verifkey*, *payload*, and *dsig* respectively. This predicate records key-data-signature triples for which the cryptographic primitive *verify_sig* is guaranteed to succeed. The postcondition of *verify_sig* is now a conjunction that captures the two uses of the function: either we do not know whether it will succeed and if it returns **true** we learn one *IsDsig* fact; or we know the relevant *IsDsig* fact and we deduce that it will return **true**.

val *verify_sig* : $vkey$:*signed verifkey* $\rightarrow p$:*payload* $\rightarrow sg$:*dsig* $\rightarrow b$:*bool*
{ b=**true** $\Rightarrow$ ($C \wedge$ *IsDsig*$(vkey, p, sg)$) $\wedge$ (*IsDsig*$(vkey, p, sg) \Rightarrow b$=**true**) }

(We modified the symbolic implementation of *verify_sig* in the *Crypto* library by inserting an **assume** (*IsDsig*$(vkey, p, sg)$) just before returning **true**, so that it can be typechecked with the new refinement. Since the verification is deterministic, this is justified by our interpretation of the predicate *IsDsig*.)

For example, our judge for authenticated mail calls *verify_sig* once, and it can now be re-typed with a success clause:

val *judge* : $text$:*string* $\rightarrow e$:*dsig* $\rightarrow b$:*bool*
{ (b=**true** $\Rightarrow$ (*Send*("A",$text$) $\wedge$ *IsDsig*(*pka*,$text$,e)) $\wedge$ (*IsDsig*(*pka*,$text$,e) $\Rightarrow b$=**true**) }

In general, once we have identified a success condition D for the judge, with $\tilde{x}$ and e as free variables for D, the judge should be typechecked against the refined type

val *judge* : $\tilde{x}$: $\tilde{Un} \rightarrow e$:*Un* $\rightarrow b$:*bool* { (b=**true** $\Rightarrow$ ($C \wedge D$)) $\wedge$ ($D \Rightarrow b$=**true**) }

Typechecking must then guarantee that D holds for the evidence used in the actual audit request. To enforce it in F7 code that includes the audit primitive **audit** C L, we declare **audit** as a function typed with precondition D:

private val audit : $\tilde{x}$: $\tilde{Un} \rightarrow e$:*Un* { D }$\rightarrow$ *unit*

(This function is trivially implemented as **let audit** $\tilde{x}$ *e* = ()). With these type annotations, typechecking plus unconditional termination of the judge guarantee auditability:

Theorem 2. *Let $\mathcal{L}$ be a well-formed protocol with a judge function that always terminates and an audit statement* **audit** C L *in its scope (with fv*$(C) = \{\tilde{x}\}$*).*

Let Γ be the typing environment **audit** *: $\tilde{x}$: $\tilde{Un} \rightarrow e$:Un { D }$\rightarrow$ unit for some formula D (with fv*$(D) \subseteq \{\tilde{x}, e\}$*). The protocol $\mathcal{L}$ is auditable for C if we have*

1. $\Gamma \vdash \mathcal{L}$[*public*($\mathcal{L}$)] *: Un and*
2. $\Gamma \vdash \mathcal{L}$[*judge*] *:* $\tilde{x} : \tilde{Un} \rightarrow e$*:Un* $\rightarrow b$*:bool*{ (*b*=**true** $\Rightarrow$ (*C* $\wedge$ *D*)) $\wedge$ (*D* $\Rightarrow$ *b*=**true**) }

These annotations tend to be verbose but easy to write. For example, in the authenticated mail, we have **val audit**: $text$:*string* $\rightarrow e$:*dsig* {*IsDsig*(*pka*,$text$,*e*)} $\rightarrow$ *unit* and, since $\mathcal{B}$verifies the signature just before the audit request, *IsDsig*(*pka*,*text*,*sign*) holds when the audit command is typed.

5 Application: A Protocol for n-Player Games

We design, implement, and verify a multiparty protocol with non-trivial auditable properties. Our protocol supports distributed games between n players and a server. The game may be instantiated to rock-paper-scissors, online auctions (as programmed in our code), leader elections, and similar partial-information protocols. For simplicity, we assume that the game is symmetric between all players and that it can be played in one round. The protocol participants are willing to cooperate but they want to reveal as little information as possible; in particular they do not reveal their moves until everyone has played (as e.g. in the Lockstep protocol [BL01]).

At the end of the game, depending on the moves for all players, one player wins, and expects to be recognised as the winner—this is our main target auditable property.

Informal description of the protocol. The protocol has two roles, the player and the server; each run involves $n+1$ principals, n players plus one server. The same principal may be involved multiple times in the same run, as several players plus possibly the server. The protocol assumes a basic public-key infrastructure, with a public-signature keypair for each principal.

The protocol has three rounds, each with a message from every player to the server, followed by a message from the server multicast to every player:

$A_i \longrightarrow S \ : A_i$	Hello
$S \longrightarrow A_i : id, \widetilde{A}, \{id, \widetilde{A}\}_S$	Start the game
$A_i \longrightarrow S \ : H_i, \{id, H_i\}_{A_i}$	Commit move, where $H_i = hash(A_i, id, M_i, K_i)$
$S \longrightarrow A_i : \widetilde{H}, \{id, H\}_A, \{id, \widetilde{H}\}_S$	Commit list
$A_i \longrightarrow S \ : M_i, K_i$	Reveal move
$S \longrightarrow A_i : \widetilde{M}, \widetilde{K}$	Reveal list

Each player first contacts the game server. Once a party of n players is ready, the server informs the players that the game starts: it generates a fresh game identifier id and signs it together with the list of players $\widetilde{A_i}$ for the game.

After accepting the server message, each player selects a move M_i and commits to it: he computes and signs the hash of his move together with the game identifier (to prevent replays), his own name (to prevent reflexion attacks), and a fresh confounder K_i (to prevent dictionary attacks on his move). The server countersigns and forwards all commitments to all players.

After accepting the server message and checking all commitments, each player unveils his move (and his confounder) to the server. The server finally publishes all moves, hence the outcome of the game.

Protocol implementation. The complete, verified source code for the protocol implementation appears online. It consists of 280 lines of F7 declarations and 420 lines of F# definitions, excluding the standard F7/F# libraries for networking and cryptography. The code is reasonably complex, partly because of the tension between confidentiality and authentication/auditability, partly because it supports any number of players. Automated verification for n-ary group protocols and their implementations is still largely an open problem, even for confidentiality and authentication [BL01].

We tested our implementation on a local network, running games that involve between 2 and 60 participants. A game involving 60 players ends in about 11 seconds on an Intel Core Duo 2GHz with 1GB RAM, running virtualised Windows XP with .NET cryptography over local HTTP communications.

Security goals (informally). Our protocol offers several properties.

- *Integrity:* the messages (Start), (Commit) and (Commit list) are authenticated.
- *Secrecy:* each player's move remains secret until successful completion of the commitment round, hence the other players' moves for this game cannot depend on it.
- *Auditability:* once a player wins a game id, it can reliably convince all other principals of his victory (according to a "judge" procedure, defined below).

To prove his wins, each player collects the verified commitments from the other players, as well as the second server signature. We now explain what constitutes evidence for this property, first operationally, by defining our judge function, then from a specification viewpoint, by defining formulas that relate the actions of the participants.

Judge and evidence. Our target property is *Wins*(*server*,*id*,*players*,*winner*,*move*), a predicate parameterized by the server principal, the game identifier, the list of players, the winner principal, and the winning move. We list below the judge, as defined in our ML implementation: a function that takes the same parameters plus some evidence (*ssig2*,*evl*):

```
let judge server id players winner move e =
  let (ssig2,evl) = e in
  let vk = get_publickey server in
  let players',hashes,moves,keys,sigs = unzip5 evl in
  if verify_sig server vk (CommitList_data(id,players,hashes)) ssig2 then (* (1) *)
  if players = players' then (* (2) *)
```

```
  if forall_1 verify_hash id evl then (* (3) *)
  if forall_2 verify_move id evl then (* (4) *)
  if winning_move move moves then (* (5) *)
  if exists winner move evl then true (* (6) *)
  else false
```

and that calls the two auxiliary functions

```
let verify_hash id x = let (player,hash,move,key,sg) = x in
  let vk = get_publickey player in verify_sig player vk (Commit_data(id,hash)) sg
let verify_move id x = let (player,hash,move,key,sg) = x in
  ishash player id key hash move
```

The evidence should consist of the server's signature on the committed hashes (*ssig2*) and a list (*evl*) of 5-tuples $A_i, H_i, M_i, K_i, \{N, H_i\}_{A_i}$ (one for each player). This evidence is checked as follows: split the tuple list into 5 lists of the respective tuple components, using a variant of the ML library function *List.unzip*; then check that (1) the server's signature on the hashes is valid; (2) the two lists of players are the same; (3) for each 5-tuple, the hash is well-signed; (4) for each 5-tuple, the hash is correctly computed from the move; (5) *move* meets some game-specific victory condition; and (6) *winner* actually played this *move*. Finally, return **true** if all those checks succeed, **false** otherwise. The code uses monomorphic variants of a ML library function *List.forall* that calls a boolean function on each element of a list and returns **true** if all those calls return **true**; we omit their definitions, which are needed only for typechecking with different refinement types.

Logical Properties. To convince ourselves (and the players) that our judge is indeed correct, and that our player is auditable for *Wins*(*s*,*id*,*pls*,*w*,*m*), we now associate logical properties with each message, at each point of the protocol. This association is enforced by typechecking our code against refinement types that embed these properties. Thus, these properties form the basis for our security verification. We refer to the code for their complete, formal definition. By convention, when a property can be attributed to a principal, the corresponding predicate records that principal as its first argument. We first specify the events assumed by the principals before signing. To sign a message, the corresponding predicate must be assumed.

Message	Assumption	Meaning
Start	*Start*$(s, players, id)$	server s started game id with $players$
Commit	*Commits*$(p, id, hash)$	player p committed to $hash$ in game id
Commit list	*CommitList*$(s, id, hashes)$	server s collected $hashes$ in game id

We also define auxiliary predicates for verifying our code, for instance recursive predicates on lists. Predicate *Mem* defines list membership. Predicate *Ishash*(p, id, h, m) is the verified post-condition of a function *ishash* that tests whether a value is the hash of a move m by principal p in game id. Predicate *Zip3*(l,l_1,l_2,l_3) is the verified post-condition of a function *unzip3* that splits a list l of triples into three lists l_1, l_2, and l_3. Predicate *Winning*(m, ms) holds when the function *winning_move*(m, ms) returns true.

The main rule of the game puts all these pieces together, formalising when the players and the server concede victory, as an assumption that defines the *Wins* predicate:

```
assume (∀server,id,winner,move.
  ∀players,moves,evl,r1,hashes,sigs,keys,hash,key,sg.
    (Start(server,players,id) ∧ CommitList(server,id,hashes)
    ∧ Zip5(evl,players,hashes,moves,keys,sigs)
    ∧ (∀p,h,m,s,k. Mem((p,h,m,k,s),evl) ⇒ (Ishash(p,id,h,m) ∧ Commits(p,id,h) ))
    ∧ Winning(move,moves) ∧ Mem((winner,hash,move,key,sg),evl))
  ⇒ ( Wins(server,id,players,winner,move)))
```

Victory is inferred when *server* started a game *id* for some *players* (*Start*) and collected some commitments *hashes* (*CommitList*), and when there are *moves*, *keys* and *sigs* that form a list of evidence *evl* (*Zip5*), such that (i) in each tuple, the hash is obtained from the move and the key(*IsHash*), and the principal signed his hash (*Plays*); (ii) *move* is the best move among all *moves* (*Winning*), and *winner* did play *move* (*Mem*).

Our model finally accounts for compromised players and servers; to this end, we provide a public interface for creating both good and bad (compromised) principals. All signing keys for all principals are kept in a database; before releasing a signing key to the opponent, we formally assume *Leak*(*p*), which collects any assumption that the compromised principal p may ever make.

```
assume (∀p. Leak(p) ⇒ (∀id,x. Start(p,x,id) ∧ CommitList(p,id,x) ∧ Commits(p,id,x) ) )
let create_bad_principal p =
  create_good_principal p; assume (Leak(p)); get_secretkey p
```

Player (with an Audit statement). In contrast with the code for the judge, the players need not agree on the code for the server and the other players. Still, a player willing to use our client code may wish to review when this code performs actions on his behalf (relying on the **assert**s statements), and when this code has enough evidence to prove his wins (relying on the **audit** statement). The code for the player is available online. The **audit** statement appears after successfully processing all three messages from the server. The gathered evidence consists of the server's signature for the list of commitments, and the list of 5-tuples representing all moves.

Security (formally). We can now precisely state and prove our security goals. The most interesting result is that, for any number of games between any number of players, for any assignment of the server and these players to principals, any player's win is auditable, even if all other participants are corrupted and collude against this player.

Theorem 3 (Security of the n-players game). *Let $\mathcal{L}$ match the protocol obtained by composing the Crypto and Net libraries and the source code of our protocol.*

1. *integrity: $\mathcal{L}$ is robustly safe;*
2. *auditability: $\mathcal{L}$ is auditable;*
3. *secrecy: $\mathcal{L}$ preserves secrecy of the moves until all players commit.*

Proof. Typechecking `game4n-dsec.fs` takes 18s and generates 105 queries to Z3 (checking secrecy requires 4 extra queries).

1. By typing the code and Theorem 1, all assertions are always satisfied.
2. By typing, Theorem 2, and a termination argument for the *judge*: its code is a sequence of let bindings on expressions that terminate in linear time in the size of their list parameters, so by construction the *judge* function terminates on all inputs.
3. By typechecking a variant of the code. We then model a move as a function with the *ReleaseMove*(*player*,*id*) precondition (defined below) so that one cannot actually apply the function without satisfying the precondition.

   assume $(\forall p,id.\ ReleaseMove(p,id) \Leftrightarrow (\exists server,players,hashes,sigs,l.$
   $Start(server,players,id) \wedge Mem(p,players) \wedge CommitList(server,id,hashes)$
   $\wedge\ Zip3(l,players,hashes,sigs) \wedge (\forall p,h,sg.\ (Mem((p,h,sg),l) \Rightarrow Commits(p,id,h)))))$

   Player p may release his move in game id, if a server committed to a list of valid sealed bids for all players including p.

Fair non-repudiation (another application). To validate our approach, we also implemented and verified a fair non-repudiation protocol with an offline TTP [KMZ02]. Using types, we proved it auditable for two properties: non-repudiation of receipt and non-repudiation of origin. (See the online version of this paper.)

6 Related Work and Research Directions

Aura [JVM+08, VJMZ08] is a programming language that embeds an authorisation logic. Compared to the F7 typechecker, which uses formulas only for typechecking then erases them, Aura's logic constructs and proofs are first-class citizens, computed and manipulated at runtime. Aura has no specific support for cryptography: generic signatures of propositions rather than of data terms are allowed, and, relying on signed proof terms, Aura can log these as evidence of any past run. Since, in their design, all authorisations decisions are implicitly auditable, at run-time Aura must carry all generated proof terms (at least before compiler optimisations). In our approach, the programmer exports the terms that will constitute the evidence, as an important, explicit part of the protocol design. The typechecker statically guarantees completeness of the evidence and, at run-time, the judge validates the associated proofs only on demand.

The use of logs for optimistic security enforcement has been advocated in earlier work [CCD+07, EW07]. The work closest to our is by Cederquist et al. [CCD+07]; they develop an audit-based logical framework for user accountability, specialised for discretionary access control. In their framework, all auditors (judges) are based on a sound and complete proof checker, and are correct in our sense. However, principals must rely on a tamper resistant logging device to prevent a malicious agent from forging a log entry. In comparison, we delegate the integrity and authorisation checks to the code of the judge. Their framework defines whether an agent is *accountable* for a given run, and hints that if an agent logs all relevant evidence before each action then all of its run will be accountable. They do not provide a static analysis method to verify accountability.

In related work on secure provenance [HSW07], the provenance certificate is a standalone set of records that includes cryptographically encrypted or signed data and keying material, and provides integrity and selective secrecy for the data. Both audit trails

in our approach and provenance certificates in theirs can be seen as proof that can be verified out-of-context.

Given a (terminating) judge it should be possible to infer automatically a success condition by computing its weakest preconditions. (This would avoid the easy but tedious task of annotating the code.) It is more challenging to design a tool that compiles audit requirements of the form **audit** C to the minimal complete evidence for a given correct judge. We conjecture that, at least in some cases, the type specification of the judge function carries enough information to enable this synthesis, and will explore this in future work.

Acknowledgments. Thanks to Karthik Bhargavan for his help with F7 and Jean-Jacques Lévy for his comments.

References

[Aba99] Abadi, M.: Secrecy by typing in security protocols. JACM 46(5), 749–786 (1999)

[BBF+08] Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. In: IEEE Computer Security Foundations Symposium, pp. 17–32 (2008)

[BFG08] Bhargavan, K., Fournet, C., Gordon, A.D.: F7: Refinement types for F#. version 1.0 (2008), `http://research.microsoft.com/en-us/projects/F7/`

[BL01] Baughman, N.E., Levine, B.N.: Cheat-proof playout for centralized and distributed online games. In: 20th Annual Joint Conference of the IEEE Computer and Communications Societies, vol.1 (2001)

[CCD+07] Cederquist, J.G., Corin, R., Dekker, M.A.C., Etalle, S., den Hartog, J.I., Lenzini, G.: Audit-based compliance control. International Journal of Information Security 6(2), 133–151 (2007)

[EW07] Etalle, S., Winsborough, W.H.: A posteriori compliance control. In: SACMAT, pp. 11–20. ACM Press, New York (2007)

[FGM07] Fournet, C., Gordon, A., Maffeis, S.: A Type Discipline for Authorization in Distributed Systems. In: IEEE Computer Security Foundations Symposium, pp. 31–48 (2007)

[FGZN08] Fournet, C., Guts, N., Zappa Nardelli, F.: A formal implementation of value commitment. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 383–397. Springer, Heidelberg (2008)

[HSW07] Hasan, R., Sion, R., Winslett, M.: Introducing secure provenance: problems and challenges. StorageSS (2007)

[ISO04] ISO/IEC. Common criteria for information technology security evaluation (2004)

[JVM+08] Jia, L., Vaughan, J.A., Mazurak, K., Zhao, J., Zarko, L., Schorr, J., Zdancewic, S.: AURA: a programming language for authorization and audit. In: ICFP, pp. 27–38 (2008)

[KMZ02] Kremer, S., Markowitch, O., Zhou, J.: An intensive survey of fair non-repudiation protocols. Computer Communications 25(17), 1606–1621 (2002)

[Pub96] NIST Special Publications. Generally accepted principles and practices for securing information technology systems (September 1996)

[Roe97] Roe, M.: Cryptography and evidence. PhD thesis, University of Cambridge (1997)

[VJMZ08] Vaughan, J.A., Jia, L., Mazurak, K., Zdancewic, S.: Evidence-based audit. In: IEEE Computer Security Foundations Symposium, pp. 177–191 (2008)