Injective agreement properties are useful trace properties in security protocols, as they ensure replay protection. Traditionally injective agreement is ensured with challenge-response mechanisms—where a fresh nonce is sent to be signed along with the response and later checked—however there are increasingly more situations where this pattern is not applicable, hence replay protection must be handled with internal state.

Formal verification of stateful protocols is particularly challenging, as it is often a source of imprecision and non-termination using state-of-the-art protocols verifiers [2]. Here we prove the desired property using refinement types—types with attached logic formulas that express properties on data—directly on the protocol implementation.

As an example we showcase a real-time authentication protocol used in automotive, where signals are transmitted in a low-bandwidth network (the CAN Bus) for reliability, and replay protection is achieved by including counters in the message signature. In this case, the combination of real-time, safety and cost constraints does not allow replay protection using challenge-response.

This pattern is also present in mobile GSM networks [1] to avoid unnecessary roundtrips during authentication, and was also proposed as an optional feature of emerging Internet protocols like QUIC [3,5] that support a zero round-trip mode. There a so-called “strike-register” could be used to ensure freshness of short-lived nonce received by the server until a time-stamp invalidates the nonce.

Contribution: we propose an approach to prove injective agreement using refinement types that is applicable to the above-mentioned scenarios. We extend on previous work on verifying weak authentication using event logs [4].

Our approach differs from previous work using refinement types and affine logic [6] in that we use classical logic formulas on the event traces to prove the strong agreement property. This allows us to use F*—an extension of the F# functional language with refinement types, supported by an SMT-based type checker—to obtain a provably correct implementation of the protocol.

1 A Protocol with Counters

We present here a simple protocol that uses state in order to avoid replay attacks. This is a simplified version of a security protocol in automotive, where signals are sent between
control units in a car:

\[ A \rightarrow B : \text{sign}(\text{Signal} || \text{Counter}, K_{AB}) \]

When A sends a signal to B, it increases a counter and signs the signal with the current value, using the shared key \( K_{AB} \). Upon receiving the message, B compares the counter with its own local copy; if the received counter is greater than B’s local counter, then the message is accepted, otherwise it is ignored.

Here follows the code for the main protocol, with omitted type signatures:

```haskell
let k = keygen sig_prop

let sender s =
  let c = next_cnt () in
  assume (Signal s c);
  let t = Format.signal s c in
  let m = mac k t in
  send (append t m);
  None

let receiver () =
  let msg = recv () in
  if length msg <> signal_size + macsize then Some "Wrong_length"
  else
    let (t, m) = split msg signal_size in
    match Format.signal_split t with
    | Some (s, c) →
      if not (fresh_cnt c) then Some "Counter already used"
      else if not (verify k t m) then Some "MAC failed"
      else (* Signal accepted *)
        (
        assert (Signal s c);
        max_lemma s c !log_p;
        log_recv s c;
        update_cnt c;
        None
        )
    | None → Some "Bad tag"
```

Both principals sender and receiver share a session key \( k \), which is generated once and for all in the protocol with an attached property \( \text{sig_prop} \), which specifies the security property, as we will see later. The sender encodes the signal into a tagged bytestring using the function \( \text{Format.signal} \), produces its signature, and finally sends the assembled message. The receiver disassembles it, checking all the conditions that violate the property. Such conditions are signaled by the use of the option return type.

The function \( \text{next_cnt} \) increments and returns a counter kept by the sender, the function \( \text{fresh_cnt} \) returns true if the counter has not been observed by the receiver, and \( \text{update_cnt} \) updates the counter for the receiver to the newer version. Communication between the two parties is done using the \( \text{send} \) and \( \text{recv} \) functions, while the cryptographic component is handled by the \( \text{MAC} \) module.
2 Verification Approach

We break down the problem of verifying injective agreement in two steps. First we prove weak agreement as shown, for example, in the RPC protocol [4]: a custom predicate on the signatures ensures that whenever the receiver accepts a message, the message has previously been produced and sent by an honest principal. Then we use the log of events to prove strong agreement: whenever we insert an event in the log that corresponds to the acceptance of the message by the receiver, we ensure by typing that the event does not appear in the log.

We use refinement types to express these properties on data. Refinement types are types of the form $x : t \{ \varphi(x) \}$, where $x$ is a variable name, $t$ is a type, and $\varphi(x)$ is a first order logic formula on $x$. The combined use of assume and assert with the predicate $\text{Signal } s \ c$ proves weak agreement by typing: the protocol types if, whenever $\text{Signal } s \ c$ is asserted, it was previously assumed as an hypothesis. Similarly the correct typing of $\text{log} \ \text{recv } s \ c$ proves strong agreement, as its type requires the event $\text{Recv } s \ c$ not to be in the log. We use refinement types to attach these properties on data, and to prove invariants on the structure of the log.

2.1 Weak Authentication

We define a logic type $\text{key_prop}$ that is used to attach a property to the key and the text. An Entry constructor requires $\text{key_prop}$ to hold on $k$ and $t$ before the entry can be created. Finally, we define a log that is used for checking our cryptographic assumption.

```plaintext
type key_prop : key \rightarrow \text{text} \rightarrow \text{Type}
type pkey (p:(\text{text} \rightarrow \text{Type})) = k: \text{key} \{ \text{key_prop } k == p \}
type entry = | Entry : k: \text{key} \rightarrow t: \text{text} \{ \text{key_prop } k t \} \rightarrow m: \text{tag} \rightarrow \text{entry}
let log_m = ref []
```

Next we define the types for MACing: $\text{keygen}$ attaches the property $p$ to the new key. Our type signatures have been adapted from [4] to fit in the state monad $\text{ST}$, which we require for verifying our protocol. The function $\text{mac}$ requires that the attached $\text{key_prop}$ holds on $k$ and $t$, while $\text{verify}$ ensures that $\text{key_prop}$ holds if the verification succeeds.

```plaintext
val keygen: p:(\text{text} \rightarrow \text{Type}) \rightarrow \text{pkey } p
val mac: k: \text{key} \rightarrow t: \text{text} \{ \text{key_prop } k t \} \rightarrow \text{ST } \text{tag}
  (\text{requires } (\text{fun } h \rightarrow \text{True})) (\text{ensures } (\text{fun } h \times h' \rightarrow \text{True})) (\text{modifies } (\text{a_ref } \text{log}\cdot m))
val verify: k: \text{key} \rightarrow t: \text{text} \rightarrow \text{tag} \rightarrow \text{ST } (b: \text{bool} \{ b \Rightarrow \text{key_prop } k t \})
  (\text{requires } (\text{fun } h \rightarrow \text{True})) (\text{ensures } (\text{fun } h \times h' \rightarrow \text{True})) (\text{modifies } (\text{no_ref}))
```

Intuitively, the MAC interface requires that a property holds before a MAC code is created, and ensures that the same property holds if the MAC code passes verification. The property $\text{sig_prop}$ attached to our protocol requires that the signed message is tagged and distinct from others, and that $\text{Signal } s \ c$ has been assumed, therefore the MAC is not forged by an adversary.

```plaintext
logic type Signal : \text{uint32} \rightarrow \text{uint16} \rightarrow \text{Type}
logic type sig_prop (msg:message) = (\exists s \ c. \ msg \Rightarrow \text{Format.signal } s \ c \land \text{Signal } s \ c)
```

This property is attached to the key $k$ when it is generated, and this allows us to verify the weak agreement condition.
2.2 Strong Authentication

As previously mentioned we ensure strong authentication by typing the \texttt{log_recv} function:

```
val log_recv: s: uint32 \to c: uint16 \to ST (unit)
  (requires (fun h \to List.for_all (fun e \to (Recv s c) <> e) (sel h log_p) \land c > cnt_max (sel h log_p)))
  (ensures (fun h \times h' \to sel h' \log_p = Recv s c :: sel h log_p \land c = cnt_max (sel h' log_p)))
  (modifies (a_ref log_p))
```

Among other conditions, this type requires that whenever we log a \texttt{Recv s c} event, the event is not present in the log. Typing every call of \texttt{log_recv}, in combination with the \texttt{assume/assert} mechanism, provides the desired injective correspondence: no more than a single occurrence of a receive event can appear in the log \textit{and} whenever we accept a message, it is produced by the honest principal.

We achieve this result by proving the following lemma and invariant:

```
val max_lemma: s: uint32 \to c: uint16 \to (\forall e. List.mem e l) \to
Lemma(\forall e. List.mem e l) \Rightarrow (Recv s c) <> e)
```

```
let invariant h = cnt_max (sel h log_p) = sel h receiver_cnt \&\& Heap.contains h receiver_cnt
  \&\& Heap.contains h sender_cnt \&\& Heap.contains h log_p \&\& receiver_cnt <> sender_cnt
```

Intuitively, \texttt{max_lemma} asserts that whenever we receive a signal \texttt{s} that is signed with a counter \texttt{c} greater than all the counters in the log, then the event \texttt{Recv s c} is not in the log. The invariant ensures that a set of properties are maintained on the heap across executions of sender and receiver. In particular, we maintain the invariant that the highest counter in \texttt{log_p} is equal to the current value of \texttt{receiver_cnt}, and that the \texttt{sender_cnt} and \texttt{receiver_cnt} are disjoint memory locations, hence \texttt{sender} and \texttt{receiver} do not interfere.

3 Conclusion

We showed how to prove injective agreement on a simple stateful protocol that uses counters for replay protection. This technique is general enough to be directly applicable to protocols that use counters and timestamps for this purpose, and we believe that it is adaptable to similar mechanisms, such as the “strike-register” of the QUIC protocol.

References