Automatically Detecting Authentication Limitations in Commercial Security Protocols

Stephen H. Brackin *
Arca Systems, Inc.
303 E. Yates St.
Ithaca, NY 14850

Abstract

Protocol failure, which occurs when an active wiretapper can obtain confidential information or impersonate a legitimate user, without performing cryptanalysis, by blocking, replaying, or modifying messages, is a surprisingly difficult, and surprisingly common, problem. This paper describes how the Automatic Authentication Protocol Analyzer, 2nd Version (AAPA2), a fast and completely automatic tool for finding the vulnerabilities that give rise to protocol failure, reveals errors in assumptions about the authentication capabilities of two large commercial protocols.

1. Introduction

Cryptographic protocols, called simply *protocols* for the remainder of this paper, are short sequences of message exchanges, usually involving encryption, intended to establish secure communication over insecure networks. A *session* is one execution of a protocol, and a *principal* is a participant in a session. The basic goals for protocols are *nondisclosure* (i.e., not revealing information to anyone not meant to receive it) and *authentication* (i.e., confirming the identities of principals to each other).

Deciding whether protocols actually achieve these goals, or can be subverted by an active wiretapper who blocks, modifies, replays, or mislabels messages, is a notoriously difficult problem [1]. *Protocol failure* [16] occurs when an active wiretapper, or *attacker*, can obtain restricted information or impersonate a legitimate principal, from a correct implementation of the protocol and without performing rapid cryptanalysis, by interfering in one or more sessions.

*This work was supported by the Advanced Research Projects Agency through Rome Laboratory contract F30602-97-C-0303.

Some tools for detecting protocol failure construct possible attacks, using algebraic properties of the algorithms used in a protocol [13, 15, 14]. Since the number of possible attacks grows exponentially with the number of messages in the protocol, the number of fields in these messages, the number of modeled algebraic properties of the algorithms, and the number of possible simultaneous protocol runs, many protocol analysts consider only simple protocols or leave out much of protocols' detail. Lowe [12] has investigated performing simplifications rigorously, so that they never hide protocol failures.

The Automatic Authentication Protocol Analyzer, 2nd Version (AAPA2) takes a different approach. It addresses only part of the protocol failure problem, though a large part — the AAPA2 gives correct results for 44 to 46 of the 53 protocols in a library of protocols analyzed in the literature [7] — and it addresses this part quickly and automatically. The time the AAPA2 takes to analyze a protocol is only quadratic in the size of the protocol, so it can analyze complicated protocols in detail. By being able to consider these details, an AAPA2 analysis can pick subtle errors out of huge amounts of obscuring complexity.

The AAPA2 user begins by writing a specification of a protocol and its expected authentication properties in the Interface Specification Language, 2nd Version (ISL2) [6]. The only hard part of the whole AAPA2 analysis process is extracting the information needed for an ISL2 specification from informal protocol documentation or sales material. ISL2 itself is so simple that it can be explained in a few paragraphs; see Section 2.

After the ISL2 specification is written, the AAPA2 does the rest automatically. It translates the ISL2 into Higher Order Logic (HOL) [11], uses the HOL98 theorem prover [17] to automatically construct proofs [5] of the protocol's desired properties in the BGNY2 identification logic [4], and translates its proved and unproved results back into ISL2. The AAPA2 user does not need to know HOL.

This paper describes using the AAPA2 to analyze two complicated commercial protocols, the main- and coin-sequence protocols from CyberCash, Inc. of Reston, VA, a leading provider of secure internet commercial services (see, for example, [10]). The AAPA2 shows that these protocols do *not* authenticate customers to merchants or merchants to customers; they just confirm that *someone* with good credit bought or sold the agreed product at the agreed price. This is adequate for most, but not all, commercial purposes; see Section 5.4.

These results do not show problems in the protocols, because the protocols were not intended to have these authentication capabilities [9]. What these results do show, though, are examples of how the AAPA2 reveals specification errors concealed by huge amounts of detail, errors that could cause commercial or governmental users of commercial security software to accidentally choose protocols inadequate for their needs.

The rest of this paper is organized as follows: Section 2 introduces ISL2, giving enough information to explain the ISL2 specifications of the main- and coin-sequence protocols in Sections 5 and 6. Section 3 describes basic AAPA2 operation and the different warning and failure messages that the AAPA2 produces. Section 4 provides an overview of the main- and coin-sequence protocols' ISL2 specifications, pointing out properties of the various types of data used and instances in which the specifications are written in particular ways to avoid AAPA2 limitations. Section 5 describes the main-sequence protocol and the AAPA2's analysis of it, and Section 6 describes the coin-sequence protocol and the AAPA2's analysis of it. Finally, Section 7 summarizes the benefits of performing an AAPA2 analysis for protocol designers and users.

2. Interface Specification Language, 2nd Version (ISL2)

The AAPA2's Interface Specification Language, 2nd Version (ISL2) [6] follows standard notation, uses intuitive terminology, and is largely self-explanatory.

An ISL2 specification has six parts:

- An optional `NAME` string that the AAPA2 uses to label its outputs.
- A `DEFINITIONS` section that names all the pieces of plaintext and functions used in a correct protocol session, and gives all of the specification's assumptions about them.
- An optional `ABBREVIATIONS` section that defines macros for simplifying the rest of the specification.
- An `INITIALCONDITIONS` section that gives the protocol principals' assumed initial conditions.
- A `PROTOCOL` section that defines, on a stage by stage basis, the messages exchanged in a correct session — i.e., the pieces of data exchanged and the statements that these exchanges are intended to convey.
- A `GOALS` section that defines, on a stage by stage basis, any authentication conditions that the message transfer at that stage is intended to establish.

The user declares each piece of plaintext as having a type that the user names. The AAPA2 takes these type declarations as assumptions that the protocol's implementations can distinguish any two pieces of plaintext of different named types. The AAPA2's current implementation requires that all principal names (or equivalently, network addresses) be of a type distinct from that of all other pieces of plaintext, but these other pieces of plaintext can be of any non-zero number of types.

Functions in ISL2 can be declared as being of any one of the following types:

- `ENCRYPT` — encryption function;
- `HASH` or `KEYED HASH` — hash or message authentication code function;
- `KEYEXCHANGE` — function for creating shared symmetric keys from public and private keys;
- `PASSWORD` — password table;
- `PRIVKEY` — private-key table or creation function;
- `PUBKEY` — public-key table or creation function;
- `SYMKEY` — symmetric-key table or creation function; or
- `TYPEPRESERVING` — function modifying a piece of plaintext in a way that does not change its type, e.g., a function that increments or decrements a nonce or timestamp.

The `DECLARATIONS` section also gives the inverses of any functions that the specification assumes are invertible. It gives inverses in three forms:

- f `HASINVERSE` g — neither f nor g uses a key and g is the inverse of f;

- f `WITH ANYKEY HASINVERSE` g `WITH ANYKEY` — f and g are symmetric-key encryption/decryption functions, and g used with any key is the inverse of f used with the same key; or

- f `WITH` $k1$ `HASINVERSE` g `WITH` $k2$ — f and g are public- or private-key encryption/decryption functions, and g used with the key $k2$ is the inverse of f used with the key $k1$.

After the `DECLARATIONS` section, `{x}f(k)` denotes `x` encrypted with function `f` and key `k`.

ISL2 follows the notation convention that an initial `^` in a name means "not" or "opposite of", so that if `PKX` is a public key then `^PKX` is the corresponding private key. The AAPA2 translates `^` to `UN`.

The specifications in Sections 5 and 6 also follow the naming convention that abbreviations end with an underscore, but that is just a convenience for the reader; it is not part of the definition of ISL2.

The `INITIALCONDITIONS`, `PROTOCOL`, and `GOALS` sections use the following language for expressing assumed, communicated, or established properties of protocol principals and the data they hold:

- `Believes` — the principal has adequate reason to believe the statement.

- `Conveyed` — the principal was the true source of the data item during the session.

- `Fresh` — the data item was created for the current session, not taken from another session.

- `PrivateKey` — for the principal, using the decryption function, the data item is one of this principal's private keys.

- `PublicKey` — for the principal, using the encryption function, the date item is one of this principal's public keys;

- `SharedSecret` — for the two principals, the data item is either known only to these principals, will be known only to these principals, or will be known only to these principals and other principals that both of these principals trust.

See [4] for the formal definitions of all these constructs.

In the `PROTOCOL` section, the `||` operator binds a statement to a data item. The protocol assumes that the principal originating this data item will not send it unless this principal believes this statement.

Finally, ISL2 accepts and ignores C-style comments. The names following the first lines of all stages of both protocols in Sections 5 and 6 are the names of the types of messages being sent, as given in CyberCash's informal protocol specifications.

3. AAPA2 Operation

This section describes basic AAPA2 operation. The AAPA2 asks whether the *possibility* of attacker interference makes it impossible for legitimate principals to reach their desired authentication conclusions. Even if things have gone perfectly, these principals do not *know* that they have gone perfectly; every successful attack creates the impression that nothing has gone wrong.

The BGNY2 logic that the AAPA2 uses to construct its proofs departs from logics in the BAN [8] family by *not* assuming that principals can always determine the structure of the message fields they receive and the statements these fields are intended to convey. Rather, it models computationally feasible type and equality checks that principals can perform to identify these fields, and imposes *explicitness* restrictions to guarantee that the statements these fields convey are well defined.

The identifications that can validly be made on the basis of the type and equality checks that a principal performs depend, of course, on exactly what those type and equality checks are, but neither ISL2 nor BGNY2 model protocols in that detail. Instead, the BGNY2 logic assumes that principals perform *all* the type and equality tests that they might reasonably be expected to perform, assigning a computable type to every field.

As noted in Section 2, ISL2 allows the user to specify different types of plaintext as being of types that the protocol's implementations will distinguish. The BGNY2 logic also assumes that all types of cryptotext and hash codes are distinct from both each other and all forms of plaintext; this makes the realistic assumption that cryptotext and hash codes are produced with headers that identify the algorithms used to produce them. An attacker could falsify these headers, of course, but not expect legitimate principals to willingly encrypt and send out information having a header other than the expected one. Finally, the BGNY2 logic assumes that tuples of different lengths are of identifiably different types. This is realistic, since software carrying out a protocol will malfunction if it does not check that the data it is operating on is of roughly the expected form.

In considering the possible fields that could be confused with an expected field, the BGNY2 logic only considers fields sent during a correct session. This excludes attacker-constructed fields and fields obtained by oracle attacks; that level of attacker creativity is too difficult to model. This limitation is not so severe as it might seem, though, because fields whose sources can be determined always involve secrets assumed unavailable to the attacker, and feasible type checks prevent what would otherwise be most oracle attacks. The BGNY2 logic also does two things that cover a broad range of attack possibilities:

- It only allows a principal to identify fields using equality tests if this principal can be confident that the values compared against are playing the roles they play in a correct session. This models that the attacker can change the values principals receive to cause subsequent tests using these values to succeed when they should fail.

- It requires that fields be identifiably distinct from fields sent *later* in the session. This models that the attacker can substitute fields from later points in earlier or parallel sessions if the tests that principals perform to check freshness are inadequate.

The BGNY2 logic's explicitness rule only allows a principal to believe the statement bound to a field if every piece of plaintext in this statement is explicitly contained in the field, treating encryption by a private key as containing the name of the principal having this private key and encryption by a shared-secret symmetric key as containing the names of the principals sharing it. This protects against the attacker sending the field to a principal other than the one intended to receive it. Except for the possibilities raised by failed explicitness checks, though, the BGNY2 logic makes the fundamental, flawed, assumption made by logics in the BAN family that principals do not give away their secrets, ignoring attacks that *trick* legitimate protocol participants into giving away their secrets.

In constructing its proofs, the AAPA2 follows the basic algorithm described in [2]. On a stage by stage basis, where stage 0 gives the protocol's assumed initial conditions, it automatically constructs and proves, where possible, a collection of *default* goals that express the authentication properties of the protocol that are likely to be of interest. The default goals for a stage that begins with the receipt of a message include that the message's sender was able to send it, that its receiver can decrypt any encrypted parts of it, that its receiver can determine the source of each encrypted or hashed part of it, and that its receiver can believe any statements that the protocol binds to the sending of these parts. Default goals are often false, since the AAPA2 cannot tell what keys a principal is expected to have in its possession at a particular stage. If a default goal fails, the AAPA2 retries it at later stages.

After proving as many default goals as it can for a stage, the AAPA2 checks whether the proved default goals have the specified authentication goals for that stage, the *user-set goals*, as easy consequences. If not, if signals a *user-goal failure*. The AAPA2 also signals three additional types of failures:

- The BGNY2 logic allows a principal to infer the source of an encrypted or hashed field only if this principal can distinguish the field from other fields readily available to an attacker; if not, the AAPA2 signals an *identification failure*.

- The BGNY2 logic allows a principal to determine the statement conveyed by a field only if the protocol binds a unique statement to this field; if not, the AAPA2 signals an *ambiguity failure*.

- The BGNY2 logic allows a principal to believe a statement bound to a field only if every piece of plaintext in this statement is explicitly contained in the field itself; if not, the AAPA2 signals an *explicitness failure*.

The AAPA2 gives a *warning*, but does not call it a failure, if it finds that a protocol has principals do something to check for freshness — e.g., check timestamps — other than look for information that they created earlier in the session themselves. It has these "second-class failures" as a convenience to users, since it would otherwise label most protocols as failed [7]. Since they can involve messages from different sessions, the AAPA2's identification failures are particularly significant when they occur along with the AAPA2's warnings.

The AAPA2 completes each analysis by making properly labeled *false* assumptions as necessary [5]. This enables it to find more than one problem in an ISL2 specification in a single execution, and prevents earlier vulnerabilities from hiding later ones.

The AAPA2's false assumptions also precisely identify the problem if the AAPA2 signals an explicitness failure. In testing explicitness, the AAPA2 checks that the set S of pieces of plaintext mentioned in a statement is a subset of the set F of pieces of plaintext contained in a message field. In testing whether one set is a subset of another, the AAPA2 uses the equivalence $S \subseteq F \iff S \cup F = F$, so a failed explicitness check causes the AAPA2 to assume that a set equals one of its proper subsets. Since the AAPA2 lists the elements of a set so that the last elements added to the set are listed first, reading from left to right, any elements of $S \cup F$ that are not also in F will be the leftmost elements, in a left-to-right list, of the elements of $S \cup F$. This makes it easy to determine which pieces of information needed to communicate a statement are missing from a message field intended to communicate this statement.

The AAPA2 produces terminal output describing its progress in analyzing the protocol, any failures that it encounters, and any false assumptions that it makes to continue its analysis. After it finishes the protocol's last stage, it produces terminal output stating whether it raised warnings and/or found failures. If it finds one or more failures for an ISL2 specification in a file `foo.isl`, it produces a file `foo.fail` containing ISL2 descriptions of all its unproved — usually partially proved — default goals and a file `foo.prvd` containing ISL2 descriptions of all the theorems that it did prove. Optional command-line flags cause it to produce "failed" and "proved" files even if it does not find a failure.

4. Notes on the Specifications

This section describes basic attributes of the ISL2 specifications in Sections 5 and 6. It describes the atomic pieces of data used in both specifications and the assumptions the specifications make about them. It also describes the simplifications, adaptations, and potential inaccuracies in these specifications.

The three principals in both protocols, as they are specified in this paper, are `Wallet`, `CashRegister`, and `Gateway`. These are collections of programs representing a customer, a merchant, and CyberCash, respectively.

The main-sequence protocol uses both RSA public-key encryption and DES symmetric-key encryption. The coin-sequence protocol uses only DES encryption, because it is computationally less expensive than RSA encryption. Both protocols use the MD5 hash algorithm.

The `Date`, `MerchantDate`, and `ServerDate` fields give not only the date, but also the time. They are used as timestamps, and sometimes used to infer that a message is fresh. The AAPA2 warns about these inferences.

`Gateway` has public keys for each `CashRegister` and `Wallet` in the main-sequence protocol. `CashRegister` and `Wallet` use the indexes `MerchantCyberKey` and `CyberKey` to tell `Gateway` that they are using the keys `PKMerchantCyberKey` and `PKCyberKey`, respectively. `CashRegister` and `Wallet` have the public keys `PKCashRegister` and `PKWallet`, but only `Gateway` uses these keys.

There are symmetric keys that are used as shared secrets between `Gateway` and `CashRegister` or `Gateway` and `Wallet` in both protocols. In the main-sequence protocol, these keys are conveyed to `Gateway` by encrypting them with appropriately chosen `Gateway` public keys.

In both protocols, all secure communication is with `Gateway`, though encrypted messages from `Gateway` to `Wallet` are sent through `CashRegister`. Both `CashRegister` and `Wallet` trust `Gateway`, but neither trusts the other and `Gateway` trusts neither of them.

The other data items in the protocols, typically irrelevant to protocol failure, involve things such as credit card data, price and currency-exchange data, response codes, and error messages. Their names describe them as well as they need to be understood in this paper.

The ISL2 models make the following simplifications:

Both models are inaccurate in that they identify *types* of data elements with data elements, treating the types as if there were only one element of each type. The item `Type`, for instance, in the main- sequence specification, assumed to originate with `CashRegister`, is actually one of several message-type identifiers known to all principals. This inaccuracy could be removed by complicating the model slightly. The coin-sequence specification makes more accurate use of type and version identifiers, having different ones for different message types.

Prices used by the `Wallet` and `CashRegister` are the same; there is no disagreement over price.

The coin-sequence specification is for only one payer-payee pair; lists are one element long.

ISL2 does not have an "X is involved in the same transaction as Y and agrees with Y about its details" predicate. Both specifications use assertions that X conveyed particular messages it was expected to convey in lieu of assertions using such a predicate.

The coin-sequence specification puts session IDs and session indexes together, in order, at the end of lists containing them so that the AAPA2 will be able to show that the pair is fresh. The AAPA2's BGNY2 logic is not complete enough to deduce that a list must be fresh if a pair of things in that list is fresh, regardless of where or in what order the two parts of the pair occur. This possibility extends to any tuple of elements of any list considered in any order. It would be expensive to check for these possibilities, since there are so many of them, and this case does not arise often enough to make doing so worthwhile.

`MerchantAmount` is a single single piece of data, with functions to extract the currency and numerical value, in the coin-sequence protocol.

5. Main-Sequence Protocol

This section describes the main-sequence protocol, gives an ISL2 specification for it that turns out to accidentally assert authentication properties for this protocol that it is not intended to have, gives the

AAPA2's analysis of this specification, and shows how the AAPA2's analysis picks the error out of a huge amount of detail.

5.1. Informal Description

The main-sequence protocol handles credit card purchases by `Wallet` from `CashRegister`. The following stage descriptions informally define the protocol *as it is specified here* — i.e., plausibly, but partially incorrectly. Note that although `CashRegister` and `Wallet` both need to trust `Gateway`, neither needs to trust the other.

1. `CashRegister` sends `Wallet` a payment request including a price and a signed hash of data including this price and other transaction details to be forwarded to `Gateway`.

2. `Wallet` sends `CashRegister` information including an unsigned copy of the hash `CashRegister` sent `Wallet`, which `CashRegister` can use for integrity checks, a symmetric-key session key encrypted with one of `Gateway`'s public keys, and an opaque section encrypted with this session key. `CashRegister` cannot decrypt either the session key or the opaque section, but forwards them to `Gateway`. The opaque section includes a signature by `Wallet` and a copy of the hash signed by `CashRegister` that `Wallet` received in the stage-1 payment request.

3. `CashRegister` sends `Gateway` information including the encrypted `Wallet` session key and opaque section from stage 2, its own symmetric-key session key encrypted with another of `Gateway`'s public keys, and a new merchant opaque section encrypted with the `CashRegister` session key that includes a signature by `CashRegister` and confirms that `CashRegister` and `Wallet` are involved in the same transaction.

4. `Gateway` verifies that `Wallet` and `CashRegister` are both involved in the same transaction, prepares replies for both of them, "opaque" and "merchant opaque" sections encrypted with their respective session keys, and sends `CashRegister` both of these replies. `CashRegister` decrypts the "merchant opaque" section and confirms that `Gateway` has confirmed that it and `Wallet` are involved in the same transaction and agree about its details, particularly the price.

5. `CashRegister` forwards the opaque section from stage 4 to `Wallet`. `Wallet` decrypts this section and confirms that `Gateway` has confirmed that it and `Wallet` are involved in the same transaction and agree about its details.

5.2. ISL2 Specification

The protocol's ISL2 specification follows.

```
/*===========================================
Main-sequence ISL2 specification
Stephen H. Brackin
===========================================*/

NAME: "CyberCash main-sequence";

DEFINITIONS:

Name: CashRegister, Gateway, Wallet;

Data:
 Accepts, AcquirerRefDataOptional,
 ActionCode, AddnlResponseDataOptional,
 Amount, AuthorizationCode,
 AvsInfoOptional, BeginTransaction,
 CardCIdOptional, CardCityOptional,
 CardCountryOptional, CardExpirationDate,
 CardName, CardNumber,
 CardOtherFieldsOptional,
 CardPostalCodeOptional,
 CardPrefixOptional, CardSalt,
 CardStateOptional, CardStreetOptional,
 CardType, CyberKey, Date,
 DebuggingInfoOptional,
 DescriptionListOptional, EndTransaction,
 Id, MerchantAmount,
 MerchantAmount2Optional, MerchantCcId,
 MerchantCyberKey, MerchantDate,
 MerchantDba, MerchantLocationOptional,
 MerchantMessage, MerchantOrderId,
 MerchantResponseCode,
 MerchantSwMessageOptional,
 MerchantSwSeverityOptional,
 MerchantSwVersion, MerchantTransaction,
 MerchantUrlOptional, Message, Note,
 OrderId, Payload, PayloadNote,
 ProcessorErrorCodeFuture, ReportFeeOptional,
 ResponseCode, ResponseDetailCodeFuture,
 RetrievalReferenceNumberOptional,
 ServerDate, ServerDateMerchantOptional,
 ServiceCategory, SwMessageOptional,
 SwSeverityOptional, SwVersion,
 TerminalIdFuture, Transaction,
 TransactionDescriptionOptional,
 TransactionOriginal,
 TransactionStatusOriginal, Type,
 TypeOriginalOptional, UrlCancel, UrlFail,
 UrlPayTo, UrlSuccess;

ENCRYPT FUNCTIONS: Des,Rsa;
HASH FUNCTIONS: MD5;
PRIVKEY FUNCTIONS: ^PK,^PKC,^PKW;
PUBKEY FUNCTIONS: PK,PKC,PKW;
SYMKEY FUNCTIONS: SKC,SKW;

Des WITH ANYKEY HASINVERSE Des WITH ANYKEY;
Rsa WITH PK(MerchantCcId) HASINVERSE
 Rsa WITH ^PK(MerchantCcId);
Rsa WITH ^PK(MerchantCcId) HASINVERSE
 Rsa WITH PK(MerchantCcId);
Rsa WITH PK(Id) HASINVERSE Rsa WITH ^PK(Id);
Rsa WITH ^PK(Id) HASINVERSE Rsa WITH PK(Id);
Rsa WITH PKW(Gateway) HASINVERSE
 Rsa WITH ^PKW(Gateway);
Rsa WITH ^PKW(Gateway) HASINVERSE
 Rsa WITH PKW(Gateway);
Rsa WITH PKC(Gateway) HASINVERSE
 Rsa WITH ^PKC(Gateway);
Rsa WITH ^PKC(Gateway) HASINVERSE
 Rsa WITH PKC(Gateway);

ABBREVIATIONS:

^CyberKey_ = ^PKW(Gateway);
^MerchantCyberKey_ = ^PKC(Gateway);
CyberKey_ = PKW(Gateway);
MerchantCyberKey_ = PKC(Gateway);
^PKCashRegister_ = ^PK(MerchantCcId);
^PKWallet_ = ^PK(Id);
PKCashRegister_ = PK(MerchantCcId);
PKWallet_ = PK(Id);
CashRegisterSessionKey_ = SKC(CashRegister);
WalletSessionKey_ = SKW(Wallet);
CardExpirationDateOptional_ =
 CardExpirationDate;
CardNameOptional_ = CardName;
CardNumberOptional_ = CardNumber;
CardSaltOptional_ = CardSalt;
CardTypeOptional_ = CardType;
DatePR1_ = MerchantDate;
KeyCH1_ = MD5(PKWallet_);
MerchantDateOptional_ = MerchantDate;
MerchantKey_ = MD5(MerchantCyberKey_);
MerchantOpaqPrefixCM1_ =
 {CashRegisterSessionKey_}
  Rsa(MerchantCyberKey_);
MerchantSignedHashKey_ =
 MD5(PKCashRegister_);
OpaqPrefixCH1_ =
 {WalletSessionKey_}Rsa(CyberKey_);
PayloadHash_ = MD5(Payload);
ServiceCategoryOptional_ = ServiceCategory;
MerchantSignedHash_ =
 {MD5(Accepts, DatePR1_, MerchantAmount,
      MerchantCcId, MerchantOrderId,
      MerchantSignedHashKey_, Note, Type,
      UrlCancel, UrlFail, UrlPayTo,
      UrlSuccess)
 }Rsa(^PKCashRegister_);
PrHash_ =
 MD5(Accepts, Date, MerchantAmount,
     MerchantCcId, MerchantOrderId,
     MerchantSignedHashKey_, Note, Type,
     UrlCancel, UrlFail, UrlPayTo,
     UrlSuccess);
PrSignedHash_ = MerchantSignedHash_;
MerchantSignatureCM1_ =
 {MD5(CyberKey, Date, Id, MerchantAmount,
      MerchantCcId, MerchantCyberKey,
      MerchantDate, MerchantTransaction,
      OrderId, PrHash_, PrSignedHash_,
      ServerDateMerchantOptional,
      Transaction, Type)
 }Rsa(^PKCashRegister_);
SignatureCH1_ =
 {MD5(Amount, CardCIdOptional,
      CardCityOptional, CardCountryOptional,
      CardExpirationDate, CardName,
      CardNumber, CardOtherFieldsOptional,
      CardPostalCodeOptional,
      CardPrefixOptional, CardSalt,
      CardStateOptional, CardStreetOptional,
      CardType, CyberKey, Date, Id,
      MerchantCcId, MerchantSignedHashKey_,
      OrderId, PrHash_, PrSignedHash_,
      SwVersion, Transaction, Type)
     }Rsa(^PKWallet_);
MerchantOpaqueCM1_ =
 {Date, DescriptionListOptional, Id,
  MerchantAmount, MerchantDba,
  MerchantLocationOptional, MerchantMessage,
  MerchantSignedHashKey_,
  MerchantSwMessageOptional,
  MerchantSwSeverityOptional,
  MerchantSwVersion, MerchantUrlOptional,
  OrderId, PrHash_, PrSignedHash_,
  RetrievalReferenceNumberOptional,
  ServerDateMerchantOptional,
  TerminalIdFuture, Transaction,
  TransactionDescriptionOptional, Type,
  MerchantKey_, MerchantSignatureCM1_
 }Des(CashRegisterSessionKey_);
MerchantOpaqueCM6_ =
 {AcquirerRefDataOptional, ActionCode,
  AddnlResponseDataOptional,
  AuthorizationCode, AvsInfoOptional,
  CardCIdOptional, CardCityOptional,
  CardCountryOptional,
  CardExpirationDateOptional_,
  CardNameOptional_, CardNumberOptional_,
  CardPostalCodeOptional, CardPrefixOptional,
  CardStateOptional, CardStreetOptional,
  CardTypeOptional_, Date,
  DebuggingInfoOptional, Id, MerchantMessage,
  MerchantSignedHashKey_,
  MerchantSwMessageOptional,
  MerchantSwSeverityOptional, OrderId,
  ProcessorErrorCodeFuture, PrHash_,
  PrSignedHash_, MerchantResponseCode,
  ResponseDetailCodeFuture,
  RetrievalReferenceNumberOptional,
  ServerDate, TerminalIdFuture, Transaction,
  Type
 }Des(CashRegisterSessionKey_);
OpaqueCH1_ =
 {Amount, CardCIdOptional, CardCityOptional,
  CardCountryOptional, CardExpirationDate,
  CardName, CardNumber,
  CardOtherFieldsOptional,
  CardPostalCodeOptional, CardPrefixOptional,
  CardSalt, CardStateOptional,
  CardStreetOptional, CardType, SwVersion,
  KeyCH1_, SignatureCH1_
 }Des(WalletSessionKey_);
OpaqueCM6_ =
 {Amount, AuthorizationCode, CardCIdOptional,
  CardCityOptional, CardCountryOptional,
  CardExpirationDateOptional_,
  CardNameOptional_, CardNumberOptional_,
  CardOtherFieldsOptional,
  CardPostalCodeOptional, CardPrefixOptional,
  CardSaltOptional_, CardStateOptional,
  CardStreetOptional, CardTypeOptional_,
  MerchantDba, MerchantLocationOptional,
  MerchantUrlOptional, Message, OrderId,
  ResponseCode, ServerDate,
  SwMessageOptional, SwSeverityOptional,
  TransactionDescriptionOptional
 }Des(WalletSessionKey_);

INITIALCONDITIONS:

CashRegister Received         /* functions */
 Des,Rsa,MD5,PK,SKC;
CashRegister Received              /* keys */
 ^PKCashRegister_,
 MerchantCyberKey_;
CashRegister Believes      /* key beliefs */
 (PrivateKey CashRegister Rsa
   ^PKCashRegister_;
  PublicKey Gateway Rsa MerchantCyberKey_;
  SharedSecret CashRegister Gateway
   CashRegisterSessionKey_);
CashRegister Received        /* other data */
 Accepts, DescriptionListOptional, Gateway,
 MerchantAmount, MerchantAmount2Optional,
 MerchantCcId, MerchantCyberKey,
 MerchantDate, MerchantDba,
 MerchantLocationOptional, MerchantMessage,
 MerchantOrderId, MerchantSwMessageOptional,
 MerchantSwSeverityOptional,
 MerchantSwVersion, MerchantTransaction,
 MerchantUrlOptional, Note, Payload,
 PayloadNote,
 RetrievalReferenceNumberOptional,
 ServerDateMerchantOptional,
 TerminalIdFuture,
 TransactionDescriptionOptional, Type,
 UrlCancel, UrlFail, UrlPayTo, UrlSuccess,
 Wallet;
CashRegister Believes     /* other beliefs */
 (Fresh MerchantDate; Trustworthy Gateway);

Gateway Received              /* functions */
 Des,Rsa,MD5,^PKC,^PKW,PK,PKC,PKW;
Gateway Believes           /* key beliefs */
 (PrivateKey Gateway Rsa ^CyberKey_;
  PrivateKey Gateway Rsa ^MerchantCyberKey_;
  PublicKey CashRegister Rsa PKCashRegister_;
  PublicKey Wallet Rsa PKWallet_);
Gateway Received             /* other data */
 AcquirerRefDataOptional, ActionCode,
 AddnlResponseDataOptional,
 AuthorizationCode, AvsInfoOptional,
 DebuggingInfoOptional, MerchantResponseCode,
 Message, ProcessorErrorCodeFuture,
 ResponseCode, ResponseDetailCodeFuture,
 ServerDate, SwMessageOptional,
 SwSeverityOptional;
Gateway Believes          /* other beliefs */
 (Fresh Date; Fresh MerchantDate);

Wallet Received               /* functions */
 Des,Rsa,MD5,PK,SKW;
Wallet Received                    /* keys */
 ^PKWallet_, CyberKey_;
Wallet Believes            /* key beliefs */
```

```
 (PrivateKey Wallet Rsa ^PKWallet_;
  PublicKey Gateway Rsa CyberKey_;
  SharedSecret Wallet Gateway
   WalletSessionKey_);
Wallet Received              /* other data */
 Amount, CardCIdOptional, CardCityOptional,
 CardCountryOptional, CardExpirationDate,
 CardName, CardNumber,
 CardOtherFieldsOptional,
 CardPostalCodeOptional, CardPrefixOptional,
 CardSalt, CardStateOptional,
 CardStreetOptional, CardType, CyberKey,
 Date, Id, OrderId, ServiceCategory,
 SwVersion, Transaction;
Wallet Believes           /* other beliefs */
 (Fresh Date; Fresh ServerDate;
  Trustworthy Gateway);

PROTOCOL:

1. CashRegister -> Wallet:         /* PR1 */
   Accepts, MerchantAmount,
   MerchantAmount2Optional, MerchantCcId,
   MerchantOrderId, MerchantDate,
   MerchantSwVersion, Note, Payload,
   PayloadNote, Type, UrlCancel, UrlFail,
   UrlPayTo, UrlSuccess,
   MerchantSignedHashKey_, PayloadHash_,
   MerchantSignedHash_;

2. Wallet -> CashRegister:         /* CH1 */
   CyberKey, Date, Id, MerchantCcId,
   MerchantDateOptional_,
   MerchantSignedHashKey_, OrderId,
   ServiceCategoryOptional_, Transaction,
   Type, PrHash_, PrSignedHash_,
   OpaqPrefixCH1_, OpaqueCH1_;

3. CashRegister -> Gateway:        /* CM1 */
   CyberKey, MerchantCcId, MerchantCyberKey,
   MerchantDate, MerchantTransaction,
   ServiceCategory, OpaqPrefixCH1_,
   OpaqueCH1_, MerchantOpaqPrefixCM1_,
   MerchantOpaqueCM1_;

4. Gateway -> CashRegister:        /* CM6 */
   MerchantCcId, MerchantTransaction,
   MerchantDate, ServiceCategoryOptional_,
   MerchantOpaqueCM6_
    ||(Wallet Conveyed SignatureCH1_),
   OpaqueCM6_
    ||(CashRegister Conveyed
        MerchantSignatureCM1_);

5. CashRegister -> Wallet:         /* CH2 */
   Date, MerchantCcId, MerchantDate,
   MerchantMessage, MerchantResponseCode,
   MerchantSignedHashKey_, MerchantSwVersion,
   Id, PrHash_, PrSignedHash_,
   ServiceCategoryOptional_, Transaction,
   Type, OpaqueCM6_;

GOALS:
3. Gateway Believes Wallet Conveyed
    SignatureCH1_;
   Gateway Believes CashRegister Conveyed
    MerchantSignatureCM1_;

4. CashRegister Believes Gateway Conveyed
    MerchantOpaqueCM6_;
   CashRegister Believes Wallet Conveyed
    SignatureCH1_;

5. Wallet Believes Gateway Conveyed
    OpaqueCM6_;
   Wallet Believes CashRegister Conveyed
    MerchantSignatureCM1_;
```

5.3. AAPA2 Response

The AAPA2's response to this specification, edited to fit IEEE article format, follows. The AAPA2 produces this response, on a 128-meg Ultra 1, in 1 hour, 48 minutes, and 13 seconds.

```
Beginning CyberCash main-sequence proofs
Warning!  Principal Gateway believes term
  Date
is fresh, but it does not create this term
Warning!  Principal Gateway believes term
  MerchantDate
is fresh, but it does not create this term
Warning!  Principal Wallet believes term
  ServerDate
is fresh, but it does not create this term

Proving default goals, stage 1
Retrying failed default goals, stage 1
Proving user goals, stage 1
Proving default goals, stage 2
Retrying failed default goals, stage 2
Proving user goals, stage 2
Proving default goals, stage 3
Retrying failed default goals, stage 3
Proving user goals, stage 3
Proving default goals, stage 4
```

```
Explicitness failure, stage: 4!  Term

  {AcquirerRefDataOptional, ActionCode,
  AddnlResponseDataOptional,
  AuthorizationCode, AvsInfoOptional,
  CardCIdOptional, CardCityOptional,
  CardCountryOptional, CardExpirationDate,
  CardName, CardNumber,
  CardPostalCodeOptional,
  CardPrefixOptional, CardStateOptional,
  CardStreetOptional, CardType, Date,
  DebuggingInfoOptional, Id,
  MerchantMessage, MD5(PK(MerchantCcId)),
  MerchantSwMessageOptional,
  MerchantSwSeverityOptional, OrderId,
  ProcessorErrorCodeFuture,
  MD5(Accepts, Date, MerchantAmount,
  MerchantCcId, MerchantOrderId,
  MD5(PK(MerchantCcId)), Note, Type,
  UrlCancel, UrlFail, UrlPayTo, UrlSuccess),
  {MD5(Accepts, MerchantDate,
  MerchantAmount, MerchantCcId,
  MerchantOrderId, MD5(PK(MerchantCcId)),
  Note, Type, UrlCancel, UrlFail, UrlPayTo,
  UrlSuccess)}Rsa(UNPK(MerchantCcId)),
  MerchantResponseCode,
  ResponseDetailCodeFuture,
  RetrievalReferenceNumberOptional,
  ServerDate, TerminalIdFuture, Transaction,
  Type}Des(SKC(CashRegister))

does not contain the data needed to
communicate statement

  Wallet Conveyed {MD5(Amount,
  CardCIdOptional, CardCityOptional,
  CardCountryOptional, CardExpirationDate,
  CardName, CardNumber,
  CardOtherFieldsOptional,
  CardPostalCodeOptional, CardPrefixOptional,
  CardSalt, CardStateOptional,
  CardStreetOptional, CardType, CyberKey,
  Date, Id, MerchantCcId,
  MD5(PK(MerchantCcId)), OrderId,
  MD5(Accepts, Date, MerchantAmount,
  MerchantCcId, MerchantOrderId,
  MD5(PK(MerchantCcId)), Note, Type,
  UrlCancel, UrlFail, UrlPayTo, UrlSuccess),
  {MD5(Accepts, MerchantDate, MerchantAmount,
  MerchantCcId, MerchantOrderId,
  MD5(PK(MerchantCcId)), Note, Type,
  UrlCancel, UrlFail, UrlPayTo,
  UrlSuccess)}Rsa(UNPK(MerchantCcId)),
  SwVersion, Transaction, Type)}Rsa(UNPK(Id))

Making the false assumption

  AX1: {Wallet, CashRegister, Gateway,
  AcquirerRefDataOptional, ActionCode,
  AddnlResponseDataOptional,
  AuthorizationCode, AvsInfoOptional,
  CardCIdOptional, CardCityOptional,
  CardCountryOptional, CardExpirationDate,
  CardName, CardNumber,
  CardPostalCodeOptional, CardPrefixOptional,
  CardStateOptional, CardStreetOptional,
  CardType, DebuggingInfoOptional, Id,
  MerchantMessage, MerchantSwMessageOptional,
  MerchantSwSeverityOptional, OrderId,
  ProcessorErrorCodeFuture, Date, Accepts,
  MerchantDate, MerchantAmount, MerchantCcId,
  MerchantOrderId, Note, UrlCancel, UrlFail,
  UrlPayTo, UrlSuccess, MerchantResponseCode,
  ResponseDetailCodeFuture,
  RetrievalReferenceNumberOptional,
  ServerDate, TerminalIdFuture, Transaction,
  Type} =
  {CashRegister, Gateway,
  AcquirerRefDataOptional, ActionCode,
  AddnlResponseDataOptional,
  AuthorizationCode, AvsInfoOptional,
  CardCIdOptional, CardCityOptional,
  CardCountryOptional, CardExpirationDate,
  CardName, CardNumber,
  CardPostalCodeOptional, CardPrefixOptional,
  CardStateOptional, CardStreetOptional,
  CardType, DebuggingInfoOptional, Id,
  MerchantMessage, MerchantSwMessageOptional,
  MerchantSwSeverityOptional, OrderId,
  ProcessorErrorCodeFuture, Date, Accepts,
  MerchantDate, MerchantAmount, MerchantCcId,
  MerchantOrderId, Note, UrlCancel, UrlFail,
  UrlPayTo, UrlSuccess, MerchantResponseCode,
  ResponseDetailCodeFuture,
  RetrievalReferenceNumberOptional,
  ServerDate, TerminalIdFuture, Transaction,
  Type};

and continuing analysis of protocol

Retrying failed default goals, stage 4
Proving user goals, stage 4
Proving default goals, stage 5

Explicitness failure, stage: 5!  Term
```

```
  {Amount,  AuthorizationCode,
   CardCIdOptional, CardCityOptional,
   CardCountryOptional, CardExpirationDate,
   CardName, CardNumber,
   CardOtherFieldsOptional,
   CardPostalCodeOptional,
   CardPrefixOptional, CardSalt,
   CardStateOptional, CardStreetOptional,
   CardType, MerchantDba,
   MerchantLocationOptional,
   MerchantUrlOptional, Message, OrderId,
   ResponseCode, ServerDate,
   SwMessageOptional, SwSeverityOptional,
   TransactionDescriptionOptional}
  Des(SKW(Wallet))

does not contain the data needed to
communicate statement

  CashRegister Conveyed {MD5(CyberKey, Date,
  Id, MerchantAmount, MerchantCcId,
  MerchantCyberKey, MerchantDate,
  MerchantTransaction, OrderId,
  MD5(Accepts, Date, MerchantAmount,
  MerchantCcId, MerchantOrderId,
  MD5(PK(MerchantCcId)), Note, Type,
  UrlCancel, UrlFail, UrlPayTo, UrlSuccess),
  {MD5(Accepts, MerchantDate, MerchantAmount,
  MerchantCcId, MerchantOrderId,
  MD5(PK(MerchantCcId)), Note, Type,
  UrlCancel, UrlFail, UrlPayTo,
  UrlSuccess)}Rsa(UNPK(MerchantCcId)),
  ServerDateMerchantOptional, Transaction,
  Type)}Rsa(UNPK(MerchantCcId))

Making the false assumption

  AX2: {CashRegister, Wallet, Gateway,
  Amount, AuthorizationCode, CardCIdOptional,
  CardCityOptional, CardCountryOptional,
  CardExpirationDate, CardName, CardNumber,
  CardOtherFieldsOptional,
  CardPostalCodeOptional, CardPrefixOptional,
  CardSalt, CardStateOptional,
  CardStreetOptional, CardType, MerchantDba,
  MerchantLocationOptional,
  MerchantUrlOptional, Message, OrderId,
  ResponseCode, ServerDate,
  SwMessageOptional, SwSeverityOptional,
  TransactionDescriptionOptional} =
  {Wallet, Gateway,
  Amount, AuthorizationCode, CardCIdOptional,
  CardCityOptional, CardCountryOptional,
  CardExpirationDate, CardName, CardNumber,
  CardOtherFieldsOptional,
  CardPostalCodeOptional, CardPrefixOptional,
  CardSalt, CardStateOptional,
  CardStreetOptional, CardType, MerchantDba,
  MerchantLocationOptional,
  MerchantUrlOptional, Message, OrderId,
  ResponseCode, ServerDate,
  SwMessageOptional, SwSeverityOptional,
  TransactionDescriptionOptional};

and continuing analysis of protocol

Retrying failed default goals, stage 5
Proving user goals, stage 5

Warning(s) and failure(s):
CyberCash main-sequence
```

5.4. Interpretation

Note in each of the explicitness failures that the AAPA2 raises for this specification that despite the huge amount of protected information that is transmitted, one critical item, the name of the customer or merchant, is missing. This makes possible the following scenario:

Bill Clinton to Tony Blair: "How would you like to buy a nice Trident submarine? Your choice of upholstery colors! Only $1 billion."

[Saddam Hussein intercepts message]

Saddam Hussein to Bill Clinton, falsifying address labels to pretend to be Tony Blair: "Sure. Make the upholstery red. Here's my CyberCash ID and the data that CyberCash can use to confirm my identity and the details of the trade."

Bill Clinton to CyberCash: "Here is my information and my customer's information for making a trade."

CyberCash to Bill Clinton: "Here is your confirmation that the purchaser's ID was valid and that a Trident submarine with red upholstery was sold for $1 billion. Here also is a package that you can send your customer to confirm the trade.".

Bill Clinton to Tony Blair: "Here is your trade confirmation from CyberCash."

[Saddam Hussein blocks message]

Later:

"How do you like your new submarine?"

"What submarine?"

The level of authentication provided by the main-sequence protocol is thus not adequate for military purposes or for commercial purposes in which the merchant must know the identity of the customer or the customer must know the identity of the merchant.

This authentication limitation of the protocol is not noted in [12], which is interesting because that effort uses techniques intended to be more thorough than an AAPA2 analysis.

The AAPA2's objections to this protocol can be eliminated by removing all statements bound to message fields and changing the `GOALS` section to simply:

```
3. Gateway Believes Wallet Conveyed
    SignatureCH1_;
   Gateway Believes CashRegister Conveyed
    MerchantSignatureCM1_;

4. CashRegister Believes Gateway Conveyed
    MerchantOpaqueCM6_;

5. Wallet Believes Gateway Conveyed
    OpaqueCM6_;
```

The AAPA2 still outputs warnings because the protocol takes the timestamps `Date`, `MerchantDate`, and `ServerDate` as adequate evidence of freshness, but it signals no failures. This time the AAPA2 completes its analysis on a 128-meg Ultra 1 in 43 minutes, 35 seconds.

6. Coin-Sequence Protocol

This section describes the coin-sequence protocol, gives an ISL2 specification for it that turns out to accidentally assert authentication properties for this protocol that it is not intended to have, gives the AAPA2's analysis of this specification, and shows how the AAPA2's analysis picks the error out of a huge amount of detail.

6.1. Informal Description

As noted in Section 4, for simplicity the ISL2 specification given here shows only the case of a single `Wallet` making purchases from a single `CashRegister`. The generalization to many `Wallet`s making purchases from a single `CashRegister` is straightforward. The ISL2 specification also assumes that all optional fields are present.

The coin-sequence protocol is in many ways similar to the main-sequence protocol. In both protocols, `Wallet` sends information to `CashRegister`, which forwards it to `Gateway`. `Gateway` debits `Wallet`'s account and credits `CashRegister`'s account, then sends information back to `CashRegister` to be forwarded to `Wallet`. The following are the main differences between the two protocols:

Instead of sending session keys encrypted with `Gateway`'s public keys, `Wallet` and `CashRegister` use session keys computed from `PayerSessionMasterKey` and `PayeeSessionMasterKey`, respectively. These master keys are distributed using a different protocol, not shown here. `Wallet` uses the main-sequence protocol to initialize its account with `Gateway`. As explained in Section 4, the ISL2 specification models the computation of the session keys by assuming that the computed session keys `PayerTransactionKey_` and `PayeeTransactionKey_` are initially held shared secrets between `Wallet` and `Gateway` and `CashRegister` and `Gateway`, respectively.

Instead of constructing signatures by computing private-key RSA encryptions of hashes, `Wallet`, `CashRegister` and `Gateway` construct signatures by computing ordinary MD5 hashes of information including shared secrets. The secret shared for this purpose by `Wallet` and `Gateway` is `PayerSessionSalt` and the secret shared for this purpose by `CashRegister` and `Gateway` is `PayeeSessionSalt`.

`CashRegister` does not give a signature to `Wallet`, and `Wallet` does not incorporate this signature into the information it signs for `Gateway`. For the sorts of small purchases made with the coin protocol, it is presumably not important that the initial request for payment came from the merchant from whom the purchase was actually made.

The following stage descriptions give an informal definition of the coin-sequence protocol *as it is specified here* — i.e., plausibly, but partially incorrectly.

1. `CashRegister` sends `Wallet` a request for payment, including a price.

2. `Wallet` sends `CashRegister` a message, encrypted with `PayerTransactionKey_`, to be forwarded to `Gateway`. This message includes the price `Wallet` has agreed to pay and a signature produced with `PayerSessionSalt`.

3. `CashRegister` sends `Gateway` the encrypted message from `Wallet`, as one of a list of such messages, along with its own message, encrypted with `PayeeTransactionKey_`, giving the prices it has agreed to accept. `Gateway` confirms that all the `Wallet` messages are for transactions with this `CashRegister`, and that they agree on the details, including the price, of the transactions.

4. `Gateway` sends `CashRegister` a reply containing a hash of data including `PayeeSessionSalt`, confirming that it has accepted the list of coin purchases from `CashRegister`, encrypting this reply with `PayeeTransactionKey_`. It also sends

`CashRegister` a list of replies, encrypted with appropriate `PayerTransactionKey_` values and containing signatures that are hashes of data including appropriate `PayerSessionSalt` values, to be forwarded to the various `Wallet`s.

5. `CashRegister` forwards a list of `Gateway` replies to a particular `Wallet`, from stage 4, to that `Wallet`, which can confirm `Gateway`'s billing of its purchases to its account.

6.2. ISL2 Specification

This protocol's ISL2 specification follows:

```
/*============================================
Coin-sequence ISL2 specification
Stephen H. Brackin
============================================*/

NAME: "CyberCash coin-sequence";

DEFINITIONS:

Name: CashRegister, Gateway, Wallet;

Data:
 Accepts, CollectedAmount, Fee,
 ForeignExchangeOptional, MerchantAmount,
 MerchantAmount2Optional, MerchantCcId,
 MerchantDate, MerchantMessageOptional,
 MerchantOrderId,
 MerchantResponseCodeOptional,
 MerchantSwVersion, MessageOptional, Note,
 OrderIdOptional, PayeeIndex, PayeeSessionId,
 PayerIndex, PayerSessionId, Payload,
 PayloadNote, ProblemOptional,
 RequestVersion, ResponseCode,
 ServiceCategory, SubType, SubVersion,
 TypeCA1, TypeCA2, TypeCA3, TypeCA4, TypePR1,
 UrlCancel, UrlFail, UrlPayTo, UrlSuccess,
 VersionCA1, VersionCA2, VersionCA3,
 VersionCA4;

ENCRYPT FUNCTIONS: Des;

HASH FUNCTIONS: MD5;

SYMKEY FUNCTIONS:
 GenerateKey,SMaster,SPayload,SSalt;

TYPEPRESERVING FUNCTIONS: ExtractCurrency;

Des WITH ANYKEY HASINVERSE Des WITH ANYKEY;

ABBREVIATIONS:

PayeeSessionMasterKey_ = SMaster(PayeeIndex);
PayeeSessionSalt_ = SSalt(PayeeIndex);
PayerSessionMasterKey_ = SMaster(PayerIndex);
PayerSessionSalt_ = SSalt(PayerIndex);
PayloadKey_ = SPayload(Wallet);

Amount_ = MerchantAmount;
CollectedAmountList_ = CollectedAmount;
IndexList_ = PayeeIndex;
MerchantAmountList_ = MerchantAmount;
MessageOptionalList_ = MessageOptional;
NoteHash_ = MD5(Note);
OrderIdOptionalList_ = OrderIdOptional;
PayeeCurrency_ =
 ExtractCurrency(MerchantAmount);
PayeeId_ = MerchantCcId;
PayeeTransactionKey_ =
 GenerateKey
  (PayeeIndex,PayeeSessionMasterKey_,TypeCA2,
   VersionCA2);
PayerIndexList_ = PayerIndex;
PayerSessionIdList_ = PayerSessionId;
PayerTransactionKey_ =
 GenerateKey
  (PayerIndex,PayerSessionMasterKey_,TypeCA1,
   VersionCA1);
PayloadHash_ = MD5(Payload);
PayloadKeyOptional_ = PayloadKey_;
PayloadKeyOptionalList_ = PayloadKey_;
ResponseCodeList_ = ResponseCode;
ServiceCategoryOptional_ = ServiceCategory;
SessionIdList_ = PayeeSessionId;
SubTypeList_ = SubType;
SubVersionList_ = SubVersion;

NoteHashList_ = NoteHash_;
PayloadHashOptional_ = PayloadHash_;
ServerSignatureCA3_ =
 MD5(Fee, MessageOptional, PayeeSessionSalt_,
     ProblemOptional, RequestVersion,
     ResponseCode, ServiceCategory, SubType,
     SubVersion, TypeCA3, VersionCA3,
     CollectedAmountList_, IndexList_,
     MessageOptionalList_,
     OrderIdOptionalList_, ResponseCodeList_,
     SessionIdList_, SubTypeList_,
     SubVersionList_,
     PayeeSessionId,  /* Shows pair fresh */
     PayeeIndex);
ServerSignatureCA4_ =
 MD5(Amount_, ForeignExchangeOptional,
     MessageOptional, OrderIdOptional,
     PayloadHashOptional_,
     PayloadKeyOptional_, PayerSessionSalt_,
     RequestVersion, ResponseCode, TypeCA3,
     VersionCA3,
     PayerSessionId,   /* Shows pair fresh */
     PayerIndex);
SignatureCA1_ =
 MD5(Amount_, NoteHash_, OrderIdOptional,
     PayeeCurrency_, PayeeId_,
     PayerSessionSalt_, PayloadHashOptional_,
     TypeCA1, VersionCA1,
     PayerSessionId,   /* Shows pair fresh */
     PayerIndex);

OpaqueCA1_ =
 {Amount_, SignatureCA1_
 }Des(PayerTransactionKey_);
OpaqueCA3_ =
 {Fee, MessageOptional, ProblemOptional,
  RequestVersion, ResponseCode, SubType,
  SubVersion, CollectedAmountList_,
  OrderIdOptionalList_, IndexList_,
  MessageOptionalList_, ResponseCodeList_,
  SessionIdList_, SubTypeList_,
  SubVersionList_, ServerSignatureCA3_
 }Des(PayeeTransactionKey_);
OpaqueListCA2_ = OpaqueCA1_;
OpaqueListCA3_ =
 {Amount_, ForeignExchangeOptional,
  MessageOptional, OrderIdOptional,
  PayloadHashOptional_, PayloadKeyOptional_,
  RequestVersion, ResponseCode,
  ServerSignatureCA4_
 }Des(PayerTransactionKey_);
PayloadHashOptionalList_ =
 PayloadHashOptional_;

OpaqueCA4_ = OpaqueListCA3_;
SignatureCA2_ =
 MD5(PayeeSessionSalt_, ServiceCategory,
     SubType, SubVersion, TypeCA2,
     VersionCA2, MerchantAmountList_,
     NoteHashList_, OrderIdOptionalList_,
     PayerIndexList_, PayerSessionIdList_,
     PayloadHashOptionalList_, SubTypeList_,
     SubVersionList_, PayeeSessionId,
     PayeeIndex);
OpaqueCA2_ =
 {SubType, SubVersion, MerchantAmountList_,
  NoteHashList_, OrderIdOptionalList_,
  PayerIndexList_, PayerSessionIdList_,
  PayloadHashOptionalList_,
  PayloadKeyOptionalList_,
  SubTypeList_, SubVersionList_,
  SignatureCA2_
 }Des(PayeeTransactionKey_);

INITIALCONDITIONS:

CashRegister Received         /* functions */
 Des,GenerateKey,MD5,SPayload;
CashRegister Received              /* keys */
 PayeeSessionMasterKey_, PayeeSessionSalt_;
CashRegister Believes       /* key beliefs */
 (SharedSecret CashRegister Gateway
   PayeeSessionMasterKey_;
  SharedSecret CashRegister Gateway
   PayeeSessionSalt_;
  SharedSecret CashRegister Wallet
   PayloadKey_);
CashRegister Received        /* other data */
 Accepts, Gateway, MerchantAmount,
 MerchantAmount2Optional, MerchantCcId,
 MerchantDate, MerchantMessageOptional,
 MerchantOrderId,
 MerchantResponseCodeOptional,
 MerchantSwVersion, Note, PayeeIndex,
 PayeeSessionId, Payload, PayloadNote,
 SubType, SubVersion, TypeCA2, TypeCA4,
 TypePR1, UrlCancel, UrlFail, UrlPayTo,
 UrlSuccess, VersionCA2, VersionCA4, Wallet;
CashRegister Believes
 (Fresh PayeeSessionId,PayeeIndex;
  Trustworthy Gateway;
  SharedSecret CashRegister Gateway
   PayeeTransactionKey_);

Gateway Received              /* functions */
 Des,ExtractCurrency,GenerateKey,MD5,SMaster,
 SSalt;
Gateway Believes            /* key beliefs */
 (SharedSecret Gateway CashRegister
   PayeeSessionMasterKey_;
  SharedSecret Gateway CashRegister
   PayeeSessionSalt_;
  SharedSecret Gateway Wallet
   PayerSessionMasterKey_;
  SharedSecret Gateway Wallet
   PayerSessionSalt_);
Gateway Received             /* other data */
 CollectedAmount, Fee,
 ForeignExchangeOptional, MerchantCcId,
 MessageOptional, ProblemOptional,
 ResponseCode, RequestVersion, TypeCA1,
 TypeCA3, VersionCA1, VersionCA3;
Gateway Believes
 (Fresh PayeeSessionId,PayeeIndex;
  Fresh PayerSessionId,PayerIndex;
  SharedSecret Gateway CashRegister
   PayeeTransactionKey_;
  SharedSecret Gateway Wallet
   PayerTransactionKey_);

Wallet Received               /* functions */
 Des,ExtractCurrency,GenerateKey,MD5;
Wallet Received                    /* keys */
 PayerSessionMasterKey_, PayerSessionSalt_;
Wallet Believes             /* key beliefs */
 (SharedSecret Wallet Gateway
   PayerSessionMasterKey_;
  SharedSecret Wallet Gateway
   PayerSessionSalt_);
Wallet Received              /* other data */
 OrderIdOptional, PayerIndex, PayerSessionId,
 ServiceCategory, TypeCA1, TypeCA3,
 VersionCA1, VersionCA3;
Wallet Believes
 (Fresh PayerSessionId,PayerIndex;
  Trustworthy Gateway;
  SharedSecret Wallet Gateway
   PayerTransactionKey_);

PROTOCOL:

1. CashRegister -> Wallet: /* PR1 */
   Accepts, MerchantAmount,
   MerchantAmount2Optional, MerchantCcId,
   MerchantOrderId, MerchantDate,
   MerchantSwVersion, Note, PayloadNote,
   TypePR1, UrlCancel, UrlFail, UrlPayTo,
   UrlSuccess, PayloadHash_,
   {Payload}Des(PayloadKey_);

2. Wallet -> CashRegister: /* CA1 */
   NoteHash_, OrderIdOptional,
   PayeeCurrency_, PayeeId_, PayerIndex,
   PayerSessionId, PayloadHashOptional_,
   ServiceCategory, TypeCA1, VersionCA1,
   OpaqueCA1_;

3. CashRegister -> Gateway: /* CA2 */
   PayeeIndex, ServiceCategory,
   PayeeSessionId, TypeCA2, VersionCA2,
   OpaqueCA2_, OpaqueListCA2_;

4. Gateway -> CashRegister: /* CA3 */
   PayeeIndex, ServiceCategory,
   PayeeSessionId, TypeCA3, VersionCA3,
   OpaqueCA3_
    ||(Wallet Conveyed SignatureCA1_),
   OpaqueListCA3_
    ||(CashRegister Conveyed SignatureCA2_);

5. CashRegister -> Wallet: /* CA4 */
   MerchantMessageOptional,
   MerchantResponseCodeOptional,
   OrderIdOptional, PayerIndex,
   ServiceCategoryOptional_, PayerSessionId,
   TypeCA4, VersionCA4, OpaqueCA4_;

GOALS:

3. Gateway Believes CashRegister Conveyed
    OpaqueCA2_;
   Gateway Believes CashRegister Conveyed
    SignatureCA2_;
   Gateway Believes Wallet Conveyed
    OpaqueCA1_;
   Gateway Believes Wallet Conveyed
    SignatureCA1_;

4. CashRegister Believes Gateway Conveyed
    OpaqueCA3_;
   CashRegister Believes Gateway Conveyed
    ServerSignatureCA3_;
   CashRegister Believes Wallet Conveyed
    SignatureCA1_;

5. Wallet Believes Gateway Conveyed
    OpaqueListCA3_;
   Wallet Believes Gateway Conveyed
    ServerSignatureCA4_;
   Wallet Believes CashRegister Conveyed
    SignatureCA2_;
   Wallet Possesses PayloadKey_;
```

6.3. AAPA2 Response

The AAPA2's response to this specification, edited to fit IEEE article format, follows. The AAPA2 produces this response, on a 128-meg Ultra 1, in 28 minutes and 27 seconds.

```
Beginning CyberCash coin-sequence proofs
Warning!  Principal CashRegister believes
 term PayeeSessionId,PayeeIndex
is fresh, but it does not create this term
Warning!  Principal Gateway believes term
  PayeeSessionId,PayeeIndex
is fresh, but it does not create this term
Warning!  Principal Gateway believes term
```

```
  PayerSessionId,PayerIndex
is fresh, but it does not create this term
Warning!  Principal Wallet believes term
  PayerSessionId,PayerIndex
is fresh, but it does not create this term

Proving default goals, stage 1
Retrying failed default goals, stage 1
Proving user goals, stage 1
Proving default goals, stage 2
Retrying failed default goals, stage 2
Proving user goals, stage 2
Proving default goals, stage 3
Retrying failed default goals, stage 3
Proving user goals, stage 3
Proving default goals, stage 4

Explicitness failure, stage: 4!  Term

  {Fee, MessageOptional, ProblemOptional,
  RequestVersion, ResponseCode, SubType,
  SubVersion, CollectedAmount,
  OrderIdOptional, PayeeIndex,
  MessageOptional, ResponseCode,
  PayeeSessionId, SubType, SubVersion,
  MD5(Fee, MessageOptional,
  SSalt(PayeeIndex), ProblemOptional,
  RequestVersion, ResponseCode,
  ServiceCategory, SubType, SubVersion,
  TypeCA3, VersionCA3, CollectedAmount,
  PayeeIndex, MessageOptional,
  OrderIdOptional, ResponseCode,
  PayeeSessionId, SubType, SubVersion,
  PayeeSessionId, PayeeIndex)}
  Des(GenerateKey(PayeeIndex,
  SMaster(PayeeIndex), TypeCA2, VersionCA2))

does not contain the data needed to
communicate statement

  Wallet Conveyed MD5(MerchantAmount,
  MD5(Note), OrderIdOptional,
  ExtractCurrency(MerchantAmount),
  MerchantCcId, SSalt(PayerIndex),
  MD5(Payload), TypeCA1, VersionCA1,
  PayerSessionId, PayerIndex)

Making the false assumption

  AX1: {Wallet, CashRegister, Gateway, Fee,
  ProblemOptional, RequestVersion,
  ServiceCategory, TypeCA3, VersionCA3,
  CollectedAmount, MessageOptional,
  OrderIdOptional, ResponseCode, SubType,
  SubVersion, PayeeSessionId, PayeeIndex} =
  {CashRegister, Gateway, Fee,
  ProblemOptional, RequestVersion,
  ServiceCategory, TypeCA3, VersionCA3,
  CollectedAmount, MessageOptional,
  OrderIdOptional, ResponseCode, SubType,
  SubVersion, PayeeSessionId, PayeeIndex};

and continuing analysis of protocol

Retrying failed default goals, stage 4
Proving user goals, stage 4
Proving default goals, stage 5

Explicitness failure, stage: 5!  Term

  {MerchantAmount, ForeignExchangeOptional,
  MessageOptional, OrderIdOptional,
  MD5(Payload), SPayload(Wallet),
  RequestVersion, ResponseCode,
  MD5(MerchantAmount,
  ForeignExchangeOptional, MessageOptional,
  OrderIdOptional, MD5(Payload),
  SPayload(Wallet), SSalt(PayerIndex),
  RequestVersion, ResponseCode, TypeCA3,
  VersionCA3, PayerSessionId, PayerIndex)}
  Des(GenerateKey(PayerIndex,
  SMaster(PayerIndex), TypeCA1, VersionCA1))

does not contain the data needed to
communicate statement

  CashRegister Conveyed
  MD5(SSalt(PayeeIndex), ServiceCategory,
  SubType, SubVersion, TypeCA2, VersionCA2,
  MerchantAmount, MD5(Note), OrderIdOptional,
  PayerIndex, PayerSessionId, MD5(Payload),
  SubType, SubVersion, PayeeSessionId,
  PayeeIndex)

Making the false assumption

  AX2: {CashRegister, Wallet, Gateway,
  MerchantAmount, ForeignExchangeOptional,
  MessageOptional, OrderIdOptional,
  RequestVersion, ResponseCode, TypeCA3,
  VersionCA3, PayerSessionId, PayerIndex} =
  {Wallet, Gateway,
  MerchantAmount, ForeignExchangeOptional,
  MessageOptional, OrderIdOptional,
  RequestVersion, ResponseCode, TypeCA3,
  VersionCA3, PayerSessionId, PayerIndex};
```

```
and continuing analysis of protocol

Retrying failed default goals, stage 5
Proving user goals, stage 5

Warning(s) and failure(s):
CyberCash coin-sequence
```

6.4. Interpretation

The results of the AAPA2 analysis of the coin-sequence protocol are very similar to the results of the AAPA2 analysis of the main-sequence protocol, and they have similar interpretations. Despite the huge amount of protected information that is transmitted, the names of the customer and merchant are missing.

The big difference in this case is that the protocol's authentication limitation is of less commercial interest. The coin-sequence protocol is used to make large numbers of small purchases — e.g., to buy candy bars. No one would be concerned if Saddam Hussein pretended to be Tony Blair to buy a candy bar, though they might be if he did so to buy thousands of candy bars.

As in the main-sequence case, the AAPA2's objections to this protocol can be eliminated by removing all statements bound to message fields and changing the GOALS section to simply:

```
3. Gateway Believes CashRegister Conveyed
    OpaqueCA2_;
   Gateway Believes CashRegister Conveyed
    SignatureCA2_;
   Gateway Believes Wallet Conveyed
    OpaqueCA1_;
   Gateway Believes Wallet Conveyed
    SignatureCA1_;

4. CashRegister Believes Gateway Conveyed
    OpaqueCA3_;
   CashRegister Believes Gateway Conveyed
    ServerSignatureCA3_;

5. Wallet Believes Gateway Conveyed
    OpaqueListCA3_;
   Wallet Believes Gateway Conveyed
    ServerSignatureCA4_;
   Wallet Possesses PayloadKey_;
```

The AAPA2 still outputs warnings because the protocol takes the pairs

```
PayeeSessionId,PayeeIndex
```

and

```
PayerSessionId,PayerIndex
```

as adequate evidence of freshness, but it signals no failures. This time the AAPA2 completes its analysis on a 128-meg Ultra 1 in 23 minutes, 16 seconds.

7. Benefits of the AAPA2 Analysis

This section notes the benefits of performing an AAPA2 analysis for designers, commercial customers, and government customers of commercial protocols.

7.1. Finding False Assumptions

The AAPA2 reveals errors in specifications of the expected authentication properties of protocols. In the case of the CyberCash main- and coin-sequence protocols, the author produced essentially identical specifications of these protocols two years ago, as part of another analysis [3], and did not find the errors reported in this paper despite spending months studying informal protocol documentation, asking many questions to CyberCash designers, showing the specifications as part of a project review, and formally analyzing the specifications with a predecessor to the AAPA2. The AAPA2's explicitness failures for these protocols came as a surprise to everyone involved in the earlier effort.

This experience with the CyberCash protocols is a realistic example of how the AAPA2 can help designers, commercial users, or government users of commercial protocols. Protocol documentation or sales material is almost always informal, protocols frequently involve large amounts of information, and getting a false impression of intended protocol properties is easy. The AAPA2 can quickly pick these errors out of masses of detail, and let designers or users of protocols deal with them before they affect the design or choice of software used for security-critical applications.

7.2. Providing Protocol-Failure Assurance

Although the AAPA2 does not detect all types of protocol failure, or all instances of failure for failure types that it does detect, the AAPA2 does detect most instances of protocol failure [7]. Performing an AAPA2 analysis is, for protocol designers, a quick and easy protection against common errors.

7.3. Raising Basic Issues

Writing and debugging an ISL2 specification naturally leads the AAPA2 user to ask and answer critical questions such as which parties communicate, which pieces of information do they exchange, where do these pieces of information come from and go, and which parties can be trusted.

7.4. Providing a Design Review

The process of formally modeling a protocol from the protocol's design specifications gives the benefits of a detailed external review of these specifications for clarity, completeness, and consistency.

7.5. Providing Sound Documentation

An ISL2 specification serves as a succinct, precise summary of a protocol. The AAPA2's formal-proof process forces this summary to be complete and consistent. This specification can then become a highly useful part of the documentation for a protocol or for a product or tool that uses this protocol.

References

[1] R. Anderson and R. Needham. Programming satan's computer. In J. van Leeuwen, editor, *Computer Science Today*, number 1000 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

[2] S. Brackin. Deciding cryptographic protocol adequacy with HOL: The implementation. In *Theorem Proving in Higher Order Logics*, number 1125 in Lecture Notes in Computer Science, pages 61–76, Turku, Finland, August 1996. Springer-Verlag.

[3] S. Brackin. Automatic formal analyses of two large commercial protocols. In *Proceedings of the 1997 DIMACS Workshop on Design and Formal Verification of Security Protocols*, Piscataway, NJ, September 1997.

[4] S. Brackin. A highly effective logic for automatically analyzing cryptographic protocols. In *Proceedings of Computer Security Foundations Workshop XII*, Mordano, Italy, June 1999. IEEE. Submitted for publication.

[5] S. Brackin. Implementing thorough and convenient automatic cryptographic protocol analysis in HOL98. In *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, Nice, France, September 1999. Springer-Verlag. To be submitted.

[6] S. Brackin. Using computably checkable types in automatic protocol analysis. In *1999 Workshop on Formal Methods and Security Protocols*, Trento, Italy, July 1999. To be submitted.

[7] S. Brackin. Automatically detecting most vulnerabilities in cryptographic protocols. *IEEE Journal on Selected Areas in Communications (JSAC), Special Issue on Network Security*, January 2000. Submitted for publication.

[8] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. Technical Report 39, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, February 1990.

[9] S. Crocker. Personal communication. Dr. Crocker is a former CyberCash executive, January 1999.

[10] J. Gleick. Here come the cyber dollars. *Reader's Digest*, December 1996. From *New York Times Magazine.*

[11] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, Cambridge, 1993.

[12] M. L. Hui and G. Lowe. Safe simplifying transformations for security protocols. Technical report, University of Leicester, Leicester, UK, December 1998.

[13] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, number 1055 in Lecture Notes in Computer Science, pages 147–166, New York, 1996. Springer-Verlag.

[14] C. Meadows. The NRL Protocol Analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[15] J. Millen, S. Clark, and S. Freedman. The Interrogator: Protocol security analysis. *IEEE Trans. on Software Engineering*, SE-13(2):274–288, February 1987.

[16] J. Moore. Protocol failure in crypto systems. *Proceedings of the IEEE*, 76(5):594–602, May 1988.

[17] K. Slind. `HOL98`. At `www.cl.cam.ac.uk/Research/HVG/FTP`, 1998.