

Using Checkable Types in Automatic Protocol Analysis

Stephen H. Brackin *

Arca Systems / Exodus Communications

303 E. Yates St.

Ithaca, NY 14850

Abstract

The Automatic Authentication Protocol Analyzer, 2nd Version (AAPA2) is a fast, completely automatic tool for formally analyzing cryptographic protocols. It correctly identifies vulnerabilities or their absence in 43 of 51 protocols studied in the literature, and it finds errors in previously asserted authentication properties of two large commercial protocols. This paper describes the AAPA2 and its modeling of type, equality, and inequality tests performed by protocol participants. This description includes defining the AAPA2's Interface Specification Language, 2nd Version (ISL2), which expresses user assumptions about identifiably distinct plaintext types.

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. Whether they actually do so, or can be subverted by an active wiretapper who blocks, replays, or modifies messages, is a notoriously difficult problem [1]. The basic issues are *authentication* (i.e., whether participants know whom they are communicating with), and *nondisclosure* (i.e., whether participants reveal information to anyone not meant to receive it).

Distortions on a protocol's execution by an active wiretapper, or *attacker*, who can potentially control all communication on the network are called *attacks*. Five of the many types of attacks that have been considered in the literature follow [13]:

- Freshness — the attacker replaces a message field with the same field from an earlier protocol run.

- Type — the attacker puts a message field of one type in place of a message field of another type.
- Binding — the attacker passes off one principal's public key as another principal's public key.
- Parallel Session — the attacker puts message fields from one protocol run into message fields in another protocol run, with two or more runs in progress simultaneously.
- Oracle — the attacker sends a message to a legitimate principal knowing that this principal's response will perform some computation useful to the attacker.

The Automatic Authentication Protocol Analyzer, 2nd Version (AAPA2) [9, 10] finds vulnerabilities to freshness, type, binding, parallel session, and some oracle attacks [8, 7]. The earlier Automatic Authentication Protocol Analyzer (AAPA) [3, 2, 5] finds vulnerabilities only to freshness attacks [6, 4].

For 51 of the 53 protocols in an on-line library, by Clark and Jacob, of protocols studied in the literature [13], the AAPA2 correctly identifies 13 as failed and 30 as not failed [8]. It misses vulnerabilities in 3 and raises false alarms in 5 others, though 2 of the false alarms obviously arise from its inability to model algebraic properties of operations used in these protocols [8]. Typos in [13] make it impossible to evaluate the AAPA2's performance on the remaining 2 protocols [8].

The AAPA, by contrast, correctly identifies only 6 of the Clark-Jacob protocols as failed and 33 as not failed [6, 8]. It misses vulnerabilities in 10 and raises false alarms in 2 others, though 1 of the false alarms obviously arises from the AAPA's inability to model algebraic properties of operations used in this protocol [6, 8]. (One of the AAPA "failure identifications" in [6] is actually a false alarm [8].)

The AAPA2 also finds authentication limitations [7] missed by the AAPA [4] in two large commercial protocols from CyberCash, Inc.

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

All these AAPA2 and AAPA results are conditional on the assumption that the protocols are implemented in such a way that protocol participants perform the reasonable type, equality, and inequality tests, described in Section 2, that the AAPA2 assumes these participants perform.

The AAPA2 produces its results for the Clark-Jacob library, on a 128-meg Ultra 1, in an average of only 2.6 minutes per protocol [8]. The AAPA2 is 3 to 6 times slower than the AAPA, but this difference is almost entirely due to how the two are implemented. The AAPA2 is based on HOL98 [21], which uses Moscow ML, which is interpreted; the AAPA is based on HOL90 [20], which uses Standard ML of New Jersey, which is compiled. The AAPA2 uses HOL98 capabilities to find multiple vulnerabilities in single protocols, something that the AAPA cannot do. Since the AAPA2 is still fast, this new functionality is far more important than the loss of speed.

Both the AAPA2 and AAPA produce their results by starting with specifications of protocols and their desired properties in a simple specification language. They both translate these specifications into Higher Order Logic (HOL), automatically construct proofs of the protocols' desired properties, and translate the proved and unproved results back into the simple specification language. They both examine only correct protocol executions, but they both ask whether the *possibility* of attacker interference makes it impossible for legitimate protocol participants to reach their desired authentication conclusions.

The AAPA2 and AAPA both construct their proofs using *authentication logics* derived from the Gong, Needham, Yahalom (GNY) logic [15], which itself is derived from the original Burrows, Abadi, and Needham (BAN)[12] authentication logic. These logics are collections of rules formalizing authentication inferences that an analyst could validly make on the basis of the information that a legitimate protocol participant has received. If the received information does not justify a desired conclusion, then this typically points to a protocol vulnerability that the analyst can easily exploit to construct an attack.

Authentication logics avoid the combinatorial explosion faced by tools that construct attacks on protocols [16, 19, 17], but they currently miss many failures because they over-simplify the problem. They currently make all their authentication deductions, for instance, by assuming that there have been no nondisclosure failures, even in response to attacks.

The AAPA2 represents a major step toward removing the over-simplifications that have limited the effectiveness of the authentication logics. While the

AAPA2's BGNV2 authentication logic [9] still assumes that there have been no nondisclosure failures, it does *not* assume, as the AAPA's BGNV authentication logic [3] and all earlier authentication logics do, that protocol participants can always correctly identify and interpret the messages that they receive. Instead, the BGNV2 logic does the following:

- It assumes that legitimate protocol participants perform a computationally feasible set of type, equality, and inequality tests on the message fields that they receive, and it requires that these tests uniquely identify every field that is intended to convey an assertion.
- It requires that the assertion conveyed by every message field that is intended to convey an assertion be explicit in the field itself, not dependent on this field's context.

These additional message-identification and message-explicitness requirements are the main sources of the AAPA2's greater effectiveness compared to the AAPA.

This paper describes the BGNV2 logic's message-identification and message-explicitness requirements. Its description of the BGNV2 message-identification requirements includes giving the full definition of the AAPA2's Interface Specification Language, 2nd Version (ISL2), which allows the user to assume which plaintext types will be distinguished by the legitimate protocol participants' type tests.

The remainder of this paper is organized as follows. Section 2 describes the BGNV2 logic's message-identification requirements and how they incorporate information from ISL2 specifications. Section 3 defines ISL2 and how it expresses user assumptions about testably distinct plaintext types. Section 4 describes the BGNV2 logic's message-explicitness requirements, and how they signal potential ambiguity without raising too many false alarms. Section 5 gives an example, an ISL2 specification and AAPA2 analysis of the Wide-Mouthed Frog protocol. Finally, Section 6 gives suggestions for future work.

In the remainder of this paper, *field* denotes either a message field or a subfield of a message field, and *principal* denotes a legitimate protocol participant. The paper also describes inferences as being made by the principals themselves rather than by an analyst examining the data that these principals have received.

2. Message-Identification

This section describes the BGNV2 logic's message-identification requirements.

The identifications that principals can validly make on the basis of the tests that they perform depend on exactly what these tests are. Neither ISL2 nor BGN2, though, model protocols in that detail. Instead, BGN2 assumes that principals perform *all* the type and equality tests that they might reasonably be expected to perform, and some inequality tests.

The identifications that principals can validly make also depend on exactly what pieces of data are available to the attacker, since this determines which possible deceptions principals must avoid. Although it does cover a reasonably large subset of these possibilities, BGN2 is currently inadequate on this score, which explains why the AAPA2 misses vulnerabilities to oracle attacks for 3 of the 51 Clark-Jacob protocols discussed in Section 1. Section 6 describes a feasible extension to BGN2 that should correct this problem.

The following three subsections describe the type, equality, and inequality tests that BGN2 assumes principals perform on all message fields that they receive that are intended to convey assertions. The fourth subsection describes what BGN2 assumes is the set of all data items available to the attacker that might be used to convey an assertion. BGN2 requires that the tests a principal performs on any message field intended to convey an assertion uniquely identify this message field among the members of this set.

2.1. Type Tests

As explained in Section 3, ISL2 allows the user to name different functions and pieces of plaintext, to define different types of plaintext, and to assign types to all function outputs and to all pieces of plaintext. BGN2 assumes that the tests principals perform always distinguish objects of different types — BGN2 takes this as the functional definition of “type”.

If an ISL2 specification of a protocol identifies two pieces of plaintext as being of different types, then this means that user assumes that the software implementing this protocol will perform tests that always distinguish these two pieces of plaintext. These tests need not be explicit. For example, if trying to interpret a network address as a time stamp will cause the software implementing a protocol to produce a core dump, then the user can safely specify network addresses and time stamps as being of different types.

BGN2 extends these ISL2 type assignments to type assignments for all fields exchanged in a protocol run by making the following assumptions:

- Encrypted and hashed fields are labeled with the names of the functions that produce them, so

fields produced by different encryption and/or hash functions are always of different types.

- Tuples of different lengths are of different types.
- All plaintext fields are of different types from all encrypted and hashed fields.
- Algorithms sent as code are of a different type from all other forms of plaintext.
- Passwords, symmetric keys, and public or private keys are of types different from each other and from all user-specified types of plaintext.

These assumptions are realistic. In implemented protocols, encrypted or hashed information is often labeled with the name and version number of the software used to produce it, so that the recipient can know which software to use for decrypting or recomputing this information. The attacker could falsify the labels on encrypted or hashed values, but not expect legitimate participants to willingly encrypt and send out information having a label other than the expected one. Software implementing a protocol will also malfunction if it does not check that the information it is operating on is of roughly the expected form.

In implementing these assumptions, BGN2 defines a HOL concrete-recursive type `:TypeName` — essentially a parameterized, potentially infinite, enumerated type — whose elements name the types that principals assign to the fields that they receive and the fields that they expect to receive. The `:TypeName` constructors and their interpretations follow:

- **CryptType**: this constructor, parameterized by a function name, denotes the type of crypto-text produced by this function.
- **FunType**: the type of an algorithm sent as code.
- **HashType**: this constructor, parameterized by a function name, denotes the type of hash code produced by this function.
- **PassType**: the type of a password.
- **PlainType**: this constructor, parameterized by a user-defined plaintext type name, denotes this type of plaintext.
- **PubKeyType**: the type of a public or private key.
- **SymKeyType**: the type of a symmetric key.
- **TnType**: the type of the empty field.
- **TxType**: the type of non-existent fields.

- **TyType**: this constructor, parameterized by two `:TypeName` values, denotes the type of an ordered pair whose first element is of the first `:TypeName` type and whose second element is of the second `:TypeName` type.

BGNY2 assigns a `:TypeName` value to a field as a function of the field and a *function environment*, which the AAPA2 computes from a protocol’s ISL2 specification. The function environment is a tuple of functions that express the assumptions the ISL2 specification makes about the algorithms, keys, and type tests used in the protocol. These functions tell the following:

- Which functions are encryption/decryption functions;
- Which functions are hash functions;
- Which functions are key-exchange functions;
- Which functions are tables of passwords;
- Which functions are tables of private keys;
- Which functions are tables of public keys;
- Which functions are tables of symmetric keys;
- Which functions preserve plaintext type; and
- Which user-defined type name is the type of each piece of plaintext.

2.2. Equality Tests

BGNY2, like the defaults for the Common Authentication Protocol Specification Language (CAPSL) [18], assumes that principals perform *all* the equality tests that they are capable of performing. If a principal holds a field and receives something that it expects to contain this field, possibly encrypted in a form that this principal can decrypt, then BGNY2 assumes that this principal will test that the field it receives contains the field that it holds.

If a principal holds a field, receives a field that it expects to contain the field that it holds, and finds that the field it receives indeed does contain the field that it holds, then this by itself proves nothing. If the field that the principal holds is something that this principal received as cleartext, for example, then the attacker might have replaced what this field should have been with what it is now in order to make the equality test that the principal just performed succeed when it should have failed! (Section 5 gives an example.) Only if the principal can be confident that the field it holds is playing the role that this field is supposed to play

does the equality test give evidence that the message field this principal receives is what this message field is supposed to be.

BGNY2 addresses this problem using its **RightRole** construct, which asserts that a field is playing its intended role in the protocol run. A field is playing its intended role if it was placed in that role by a legitimate principal not influenced by an attack.

The BGNY2 rules formally defining **RightRole** [9] involve many HOL concepts extraneous to this paper. These concepts include inductively defined relations and the “deep embedding” of an authentication logic into HOL [3]. These rules’ English interpretations, though, which follow, are straightforward:

- A principal can be confident that the fields it initially holds are playing their intended roles.
- If a principal can be confident that a field it receives is from another principal in the current protocol run and is of an expected form, then this principal can be confident that this field is playing its intended role.
- If a principal can be confident that a computed (e.g., encrypted) field is playing its intended role, and is capable of inverting the computation (e.g., decrypting), then this principal can be confident that the result of this inversion (e.g., the plaintext) is playing its intended role.
- If a principal can be confident that a hash code is playing its intended role and is capable of re-computing this hash code, then this principal can be confident that the field hashed is playing its intended role.
- A principal can be confident that a pair of elements is playing its intended role if and only if this principal can be confident that each element of this pair is playing its intended role.

BGNY2 interprets the results of a principal’s equality tests by using the **RightRole** statements that this principal believes — i.e., the set of fields held by this principal that the principal can be confident are playing their intended roles. BGNY2 only allows a principal to infer the identity of a field from this principal’s equality tests if the subset of these tests consisting of tests against values in the “playing their right roles” set uniquely identify the field.

2.3. Inequality Tests

BGNY2 assumes that principals perform a very limited amount of *inequality* testing. It assumes that if

principals receive fields containing what they expect to be other principals' names, and they are capable of extracting these names, then they will object if these names are actually their own names. This is easy to implement, analogous to network software that detects and short-circuits messages sent from a site to itself.

2.4. Fields Available to the Attacker

In considering the possible fields that could be confused with an expected field, BGNV2 only considers fields sent during a correct protocol run. This excludes attacker-constructed fields and fields constructed by legitimate principals in response to oracle attacks. It is possible to model that level of attacker creativity, but doing so requires further complicating the logic — see Section 6.

BGNV2 does require, though, that fields be identifiably distinct from fields sent *later* in the protocol run. This models that the attacker can substitute fields from later points in earlier or parallel protocol runs if the tests that principals perform to determine freshness are inadequate, which they are in many protocols [8].

The AAPA2 gives a *warning* if it finds that a protocol has principals do something to check for freshness — e.g., check a timestamp — other than look for fields that they created earlier in the protocol run themselves. Since they can involve messages from different protocol runs, identification failures are particularly significant when they occur along with the AAPA2's warnings.

For practical purposes, BGNV2's model of the fields available to an attacker is also not so inaccurate as it might seem. The issue of message identification only arises for messages that are intended to convey assertions, and only messages whose sources can be reliably determined can reliably convey assertions. (BGNV2 restricts believing the assertion conveyed by a message to messages whose sources can be reliably determined.) Fields whose sources can be reliably determined always involve secrets assumed unavailable to the attacker, so the attacker can only obtain these fields from messages sent by legitimate principals. Feasible type tests also prevent what would otherwise be most oracle attacks, so the attacker can often only obtain fields that are sent during the normal execution of the protocol.

3. ISL2

This section gives the full definition of ISL2. It gives the BNF grammar for ISL2, with interspersed comments. These comments include notes on semantic restrictions imposed by the AAPA2, error messages produced by the AAPA2, and informal descriptions of

the meanings of ISL2 constructs asserting properties of principals and fields.

A protocol specification has six parts: a name string that the AAPA2 uses to label its outputs; definitions naming the functions and pieces of plaintext used in the protocol; an optional abbreviations section that introduces substitutions for simplifying the remainder of the specification; the protocol's assumed initial conditions; the definition of a correct run of the protocol; and finally the authentication conditions a correct protocol run is expected to achieve.

```
<ProtSpec> ::=
  <Name> <Defs> [<Abbs>] <Init> <Prot> <Goals>
```

The name string begins and ends with the " character, but can contain any character besides the " character, denoted as in regular expressions by [^"]

```
<Name> ::= 'NAME' ':' ''' <String> ''' ';'
<String> ::= [<String>] [[^"]]
```

The definitions section is a list of “declarations” or “relation specifications”. A “declaration” asserts that a list of identifiers either names plaintext fields of a user-supplied type or names functions of a type determined by ISL2 keywords. A “relation specification” gives the inverse of an invertible function or an identity expressing the critical properties of a key-exchange function; it relates a function, function with key, or function applied to a list containing private and public keys to another function, function with key, or function applied to a list containing private and public keys.

The AAPA2 treats declaring two plaintext fields as being of different named types as assuming that the protocol is implemented in such a way that protocol principals never confuse one with the other. The AAPA2's current implementation requires that all principal names (or equivalently, network addresses) be assumed to be of a type different from all other plaintext types. The other plaintext fields can be assumed to be of any non-zero number of types. Plaintext type names can be arbitrary ISL2 identifiers.

The AAPA2 requires that every function be declared in such a way that the type of its outputs is well-defined. For encryption and hash functions, the AAPA2 assumes that their outputs are produced with headers that identify the algorithms used to produce them. A “type preserving” function — e.g., one that increments or decrements a nonce or timestamp — produces outputs of the same types as its inputs.

The AAPA2 also requires that all keys and passwords be given by functions. (Tables are a type of function.) This avoids the error, made with the AAPA, of

having all specifications implicitly assume that a principal that holds a key always knows the role that this key will play in a correct protocol run. This correction allows a server, for instance, to hold a key that it shares with the principal who initiates a protocol run without having this server know *which* of the keys that it holds is the one that it shares with the initiator of the protocol run.

ISL2 identifiers are standard in that they can contain numeric characters or underscores if these characters are preceded by alphabetic characters, but non-standard in two ways: they can begin with one of the special characters \sim or \wedge meaning “not”, so these characters can be used to construct intuitive names for the private key corresponding to a public key or the inverse of an invertible function; and they can contain the prime character $'$. In its output files, the AAPA2 translates both \sim and \wedge to UN and translates $'$ to PR.

```
<Defs> ::= 'DEFINITIONS' ':' <DecOrRelList>
<DecOrRelList> ::=
  [<DecOrRelList>] <DecOrRel>
<DecOrRel> ::= <Dec> | <RelSpec>
<Dec> ::= <IdOrFun> ':' <IdList> ';'
<IdOrFun> ::=
  <Id> | (<Qualifier> 'FUNCTIONS')
<Id> ::= ['~' | '^'] <Chr> [<GenChrList>]
<Chr> ::= 'A' | ... | 'Z' | 'a' | ... | 'z'
<GenChrList> ::=
  (<Chr> | <Dgt> | '_' | ''') [<GenChrList>]
<Dgt> ::= '0' | ... | '9'
<Qualifier> ::=
  'ENCRYPT' | ['KEYED'] 'HASH' |
  'KEYEXCHANGE' | 'PASSWORD' | 'PRIVKEY' |
  'PUBKEY' | 'SYMKEY' | 'TYPEPRESERVING'
<IdList> ::= [<IdList> ','] <Id>
```

If a relation specification says that one function is the inverse of another, both functions must be given with or both without keys; if both are given with keys, the ANYKEY value can be used as the key for both functions. For public-key encryption, the keys, and possibly the functions, will be different. For symmetric-key encryption, the keys (or ANYKEY values) will be the same, though the functions might be different. Representative examples in the various cases are

```
Minus1 HASINVERSE Plus1;
DES WITH ANYKEY HASINVERSE DES WITH ANYKEY;
RSA WITH Kp(A) HASINVERSE RSA WITH ~Kp(A);
```

If the relation specification says that one function value is equal to another, the same function must be given on both sides of the EQUALS, the function must have been declared as a key-exchange function, and this

function must be given with lists of arguments whose first elements are private keys and whose second elements are public keys.

```
<RelSpec> ::=
  <Id> ['WITH' <GenKey>] <RelType> <Id>
  ['WITH' <GenKey>] ';';
<GenKey> ::=
  <Data> | ('(' <DataList> ')') | 'ANYKEY'
<RelType> ::= 'HASINVERSE' | 'EQUALS'
```

The optional abbreviations section simply defines identifiers as abbreviations for more complicated ISL2 expressions.

```
<Abbs> ::= 'ABBREVIATIONS' ':' <AbbList> ';';
<AbbList> ::= [<AbbList> ';'] <Abb>
<Abb> ::= <Id> '=' <DataList>
```

The initial-conditions section is syntactically an arbitrary list of ISL2 statements, though the proof attempt will fail with an error message if this list of statements includes other than **Believes** or **Received** statements.

```
<Init> ::=
  'INITIALCONDITIONS' ':' <StmtList> ';';
<StmtList> ::= [<StmtList> ';'] <Stmt>
```

Informal descriptions of the meanings of the various ISL2 statement constructors follow. These constructors’ formal meanings are given via the function BGNy2 defined in the AAPA2’s underlying HOL theory [9]. For brevity, these descriptions say “principal” rather than “principal named by a plaintext message field”.

- **Believes**: The principal has adequate reason to believe the statement.
- **Conveyed**: The principal was the creator and source, during the current protocol run, of the field.
- **Fresh**: The field was created for the current protocol run.
- **Possesses**: The principal has received the field or is capable of computing this field from fields that it has received.
- **PrivateKey**: The key, for the algorithm, is one of the principal’s private keys.
- **PublicKey**: The key, for the algorithm, is one of the principal’s public keys.
- **Received**: The principal received the field before the current protocol run, or received this field as, or as part of, some message sent earlier in the current run.

- **RightRole**: The field is playing the role in the current protocol run that it plays in a correct run [9, 10]. **RightRole** often appears in the AAPA2’s `.fail` and `.prvd` output files (see Section 5), but it is seldom useful in ISL2 specifications.
- **SharedSecret**: The field is unguessable, and if the pair of principals possess it, or come to possess it through secure means, then they are or will be the only ones, other than principals they both trust, who possess it.
- **Trustworthy**: If the principal was the source of a field with an associated statement, then this is adequate reason for believing this statement.

Statements are defined in terms of “statement sublists”, which are lists of implicitly conjuncted statements separated by semicolons, and by lists of fields.

```

<Stmt> ::=
  (<Id> 'Believes' <Stmt>) |
  (<Id> 'Conveyed' <DataList>) |
  ('Fresh' <DataList>) |
  (<Id> 'Possesses' <DataList>) |
  ('PrivateKey' <Id> <Id> <Data>) |
  ('PublicKey' <Id> <Id> <Data>) |
  (<Id> 'Received' <DataList>) |
  ('RightRole' <DataList>) |
  ('SharedSecret' <Id> <Id> <DataList>) |
  ('Trustworthy' <Id>) |
  ((' <StmtSubList> '))
<StmtSubList> ::= [<StmtSubList> ';' ] <Stmt>
<DataList> ::= [<DataList> ',' ] <Data>

```

A field is either an identifier — a piece of plaintext or an abbreviation — or an identifier previously declared as a function applied to a list of arguments, or an expression denoting signed or encrypted fields. The field $\langle x \rangle h(k)$ denotes x together with a message authentication code produced from x with key-dependent hash function h and key k . The field $[x](h, f)(k)$ denotes x together with a signature that is the encryption, using function f and key k , of the hash of x produced using non-key-dependent function h . Finally, $\{x\}f(k)$ denotes x encrypted with function f and key k .

Every hashed, encrypted, or signed field can optionally have an associated *extension*, which is a statement bound to the field that the protocol expects a principal to believe or else it would not send this field. For signed fields, the AAPA binds the extension to the message authentication code or encrypted hash code that serves as the signature part of the field. The field $t|(slist)$ denotes field t with the implicitly conjuncted list of statements $slist$ as an extension.

The ISL2 treatment of extensions comes from the GNY logic [15], but the AAPA2 restricts extensions to hashed or encrypted fields, since only hashed or encrypted fields are protected from being secretly modified in transit, only fields that cannot be secretly modified in transit can be confidently identified as coming from a particular principal, and only fields coming from a particular principal can be confidently believed to have the properties the protocol expects them to have.

```

<Data> ::=
  (<Id> [<Ext>]) |
  (<Id> '(' <DataList> ')') [<Ext>]) |
  ('<' <DataList> '>') <Id> '(' <DataList> ')')
  [<Ext>]) |
  ('[' <DataList> ']') '(' <Id> ',' <Id> ')')
  '(' <DataList> ')') [<Ext>]) |
  ('{' <DataList> '}' <Id> '(' <DataList> ')')
  [<Ext>])
<Ext> ::= '|'| '(' <StmtSubList> ')')

```

Each message reception begins a new *stage* in a protocol run. The definition of a correct protocol run is a list of messages, separated by semicolons. Each message is given by a header giving the protocol stage, the sending principal, the receiving principal, and the field sent. The AAPA2 automatically adds the names of the sending principal and the intended receiving principal as the first two fields of every message sent.

```

<Prot> ::= 'PROTOCOL' ':' <MsgList> ';'
<MsgList> ::= [<MsgList> ';' ] <Msg>
<Msg> ::= <Header> ':' <DataList>
<Header> ::= <Number> '.' <Id> '->' <Id>
<Number> ::= ('1'|...|'9') [<DgtList>]
<DgtList> ::= [<DgtList>] <Dgt>

```

Finally, the goals section gives the correct run’s expected authentication properties, by protocol stage, as statements. A stage can have no associated statements, in which case it can be omitted, or it can have several associated statements, in which case these statements are given in a list with each statement terminated by a semicolon. While stages can be omitted, they must be given in increasing order.

```

<Goals> ::= GOALS ':' <GoalStmtList>
<GoalStmtList> ::=
  [<GoalStmtList>] <GoalStmt>
<GoalStmt> ::= <Number> '.' <StmtList> ';'

```

4. Explicitness Requirements

This section describes how BGN2 protects against attacks that involve tricking the principals who receive

message fields into interpreting these fields in ways that the principals sending these fields did not intend. Such attacks can involve modifying neighboring plaintext fields (e.g., “from” and “to” labels) or redirecting message fields to principals not intended to receive them. Binding attacks typically work in this way.

BGNY2 protects against such attacks by requiring that a principal only believe the statement bound to a message field if *every* piece of plaintext in this statement is explicitly contained in the message field itself. This prevents misinterpretations arising from attacker-induced distortions in the message field’s context.

To avoid what would otherwise be a large number of extraneous explicitness failures, though, BGNY2 treats any field encrypted with a principal’s private key as containing the name of this principal, and it treats any field encrypted with a symmetric key that is a secret shared by two principals as containing the names of these principals. Since being a private key or a shared secret depends on the current state of the protocol run, this introduces some dependence on context, but not any *new* dependence on context — i.e., any dependence on context not already present in the authentication logics’ assumptions about shared secrets.

For the 51 Clark-Jacob protocols discussed in Section 1, BGNY2’s explicitness requirements point up potential vulnerabilities in only 8 protocols, and for 6 of these protocols there are attacks that successfully exploit these vulnerabilities [8].

5. Example

This section gives an example of the very simple, but nevertheless failed, protocol, the Wide-Mouthed Frog protocol [13], and the AAPA2’s analysis of it.

In this protocol, principals **A** and **B** communicate through a trusted server **S**, using symmetric-key encryption with keys that they each share with **S**, to exchange a newly created symmetric key for further communication. **A** begins the protocol by sending **S** a package, encrypted with a key **Kas** that **A** shares with **S**, containing a timestamp **Ta**, the name **B** of the principal that **A** wants to communicate with, and an unguessable, newly generated symmetric key **Kab** that **S** is to forward to **B**. **S** then sends **B** a package, encrypted with a key **Kbs** that **B** shares with **S**, containing **S**’s timestamp **Ts**, the name **A** of the principal that wants to communicate with **B**, and the key **Kab**.

The protocol’s ISL2 specification follows. In ISL2, a term of the form $\{x\}f(k)$ denotes x encrypted with function f and key k , and the \rightarrow operator means “sends”. Every message is implicitly labeled with the names of its sender and intended receiver — though

the attacker can change these labels before the message is received. The $||$ operator binds a statement to a field; the specification assumes that the principal sending this field will not send it unless this principal believes this statement. The statements in the **INITIALCONDITIONS** and **GOALS** sections, and bound to fields in the **PROTOCOL** section, are defined in Section 3, which also gives the full definition of ISL2.

```
NAME: "Wide-Mouthed Frog";
DEFINITIONS:
Name: A,B,S; Data: Ta,Ts;
ENCRYPT FUNCTIONS: E;
SYMKEY FUNCTIONS: Ka,Ks;
E WITH ANYKEY HASINVERSE E WITH ANYKEY;
```

```
ABBREVIATIONS:
Kab = Ka(B); Kas = Ks(A); Kbs = Ks(B);
```

```
INITIALCONDITIONS:
A Received E,Ka,B,S,Ta,Kas;
A Believes SharedSecret A B Kab;
B Received E,Kbs;
B Believes
(Fresh Ts; SharedSecret B S Kbs;
Trustworthy S);
S Received E,Ks,Ts;
S Believes
(Fresh Ta; SharedSecret S A Kas;
SharedSecret S B Kbs; Trustworthy A);
```

```
PROTOCOL:
1. A -> S: {Ta,B,Kab}E(Kas)
           ||(SharedSecret A B Kab);
2. S -> B: {Ts,A,Kab}E(Kbs)
           ||(SharedSecret A B Kab);
```

```
GOALS:
1. S Possesses Kab;
   S Believes SharedSecret A B Kab;
2. B Possesses Kab;
   B Believes SharedSecret A B Kab;
```

The following paragraphs give a brief introduction to AAPA2 operation; see [11] for more information. In constructing its proofs, the AAPA2 follows the algorithm described in [10]. On a stage by stage basis, it automatically constructs and proves, where possible, a collection of *default* goals that express the authentication properties of the protocol run that are likely to be of interest. The AAPA2 proves as many default goals as it can for a stage, then checks whether the proved default goals have the specified authentication goals for that stage as easy consequences.

If the AAPA2 is unable to prove all the specified authentication goals for a stage of the protocol, then it signals a *user-goal failure*. If the AAPA2 finds that a principal could otherwise infer the source of an encrypted or hashed field, but that this principal cannot distinguish this field from other fields that BGNy2 assumes are readily available to the attacker, then the AAPA2 signals an *identification failure*. If the AAPA2 finds that a principal could otherwise believe the statement bound to a field, but not every piece of plaintext in this statement is explicitly contained in the field itself, then 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 a timestamp — other than look for fields that they created earlier in the protocol run themselves. It has these “second-class failures” as a convenience to users, since it would otherwise label most protocols as failed [8].

The AAPA2 completes each analysis by making *false* assumptions as necessary [10]. This enables it to find more than one problem in a specification in a single execution, and prevents earlier vulnerabilities from hiding later ones.

The AAPA2 produces terminal output describing its progress in analyzing the protocol, any failures that it encounters, and any false assumptions that it makes. 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 default goals and a file `foo.prvd` containing ISL2 descriptions of any false axioms it assumed and all the theorems that it proved. It outputs axioms about sets in a pseudo-ISL2 that states equalities between sets of ISL2 statements or terms. If a “theorem” depends on one or more false axioms, the AAPA2 labels its output of that “theorem” with the names of these axioms.

The AAPA2 produces a formal analysis of the Wide-Mouthed Frog ISL2 specification, on a 128-meg Ultra 1, in 1 minute, 34 seconds. The AAPA2’s terminal output, edited to take up less space, follows:

```
Beginning Wide-Mouthed Frog proofs
Warning! Principal B believes term
  Ts
is fresh, but it does not create this term
Warning! Principal S believes term
  Ta
is fresh, but it does not create this term
Proving default goals, stage 1
Identification failure, stage: 1!
```

```
Receiver’s default tests do not distinguish:
  {Ta,B,Ka(B)}E(Ks(A))
```

```
from the following term(s) substitutable by
the attacker:
```

```
  {Ts,A,Ka(B)}E(Ks(B))
```

```
Making the false assumption
```

```
AX1: {{Ta,B,Ka(B)}E(Ks(A)),
      {Ts,A,Ka(B)}E(Ks(B))} =
      {{Ta,B,Ka(B)}E(Ks(A))};
```

```
and continuing analysis of protocol
```

```
Proving user goals, stage 1
```

```
Proving default goals, stage 2
```

```
Proving user goals, stage 2
```

Warning(s) and failure(s): Wide-Mouthed Frog

Since the AAPA2’s `.fail` and `.prvd` output files are typically useful only if it finds user-goal failures, and are indeed not useful for the Wide-Mouthed Frog example, this paper will omit giving them.

The AAPA2’s analysis of the Wide-Mouthed Frog protocol shows this protocol’s essential vulnerability: **S** cannot distinguish the field $\{Ta, B, Kab\}E(Kas)$ from the field $\{Ts, A, Kab\}E(Kbs)$, so the attacker can mislabel a copy of a message field that **S** sends **B** for **A** and pass it off as a request *from B* to communicate *with A*, in the process updating the timestamp **Ta** to **Ts**. The attacker can repeat this process indefinitely, alternately pretending to be **B** and **A**, and keep **A** and **B** using the key **Kab** until the attacker has been able to determine this key.

6. Future Work

This section describes possible future work.

The AAPA2’s major remaining limitation is that BGNy2 does not accurately model the set of data items available to the attacker. This is responsible for all 3 of the AAPA2’s “misses” on the 51 Clark-Jacob protocols discussed in Section 1 [8]. Although it is provably impossible to *exactly* determine the set of data items available to an attacker in less than exponential time [14], it should be possible to incorporate a reasonable *superset* of this set into an extension of BGNy2 without significantly degrading the AAPA2’s speed.

An extension of BGNy2 could define a superset of the data items available to an attacker as follows:

- Assume that the attacker has been able to induce an arbitrary number of previous and concurrent runs of the protocol.
- Assume that every output that could have been produced by a legitimate principal has been produced by this principal — i.e., assume that all

possibilities covered by conditional tests have occurred.

Such a superset will be infinite, but the extended logic could still use unification techniques to prove that a fixed set of type, equality, and inequality tests performed by a principal will succeed in identifying a unique member of this set.

The extended logic could also incorporate use of this superset into all of its rules dealing with shared secrets, to require that everything treated as a shared secret is not even potentially made available to the attacker. By doing so, the extended logic would become the first authentication logic to address nondisclosure as well as authentication.

Finally, given the AAPA2's success with modeling type tests implicitly performed by protocol principals, the Common Authentication Protocol Specification Language (CAPSL) [18], a standard specification language being developed for all protocol designers and all protocol analysis tools, should incorporate such tests. CAPSL already allows user-defined types, but does not include operators that convert these types into functions testing whether particular fields are of particular user-defined types.

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. A HOL extension of GNY for automatically analyzing cryptographic protocols. In *Proceedings of Computer Security Foundations Workshop IX*, County Kerry, Ireland, June 1996. IEEE.
- [4] 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.
- [5] S. Brackin. An interface specification language for automatically analyzing cryptographic protocols. In *Internet Society Symposium on Network and Distributed System Security*, San Diego, CA, February 1997. IEEE.
- [6] S. Brackin. Evaluating and improving protocol analysis by automatic proof. In *Proceedings of Computer Security Foundations Workshop XI*, Rockport, MA, June 1998. IEEE.
- [7] S. Brackin. Automatically detecting authentication limitations in commercial security protocols. In *Proceedings of the 22nd National Conference on Information Systems Security*, Alexandria, VA, October 1999. IEEE.
- [8] S. Brackin. Empirical Tests of the Automatic Authentication Protocol Analyzer, 2nd Version (AAPA2). Technical Report 99006, Arca Systems / Exodus Communications, Ithaca, NY, June 1999.
- [9] S. Brackin. A highly effective logic for automatically analyzing cryptographic protocols. Technical Report 99023, Arca Systems / Exodus Communications, Ithaca, NY, May 1999.
- [10] S. Brackin. Implementing effective automatic cryptographic protocol analysis. Technical Report 99032, Arca Systems / Exodus Communications, Ithaca, NY, June 1999.
- [11] S. Brackin. User's Manual for the Automatic Authentication Protocol Analyzer, 2nd Version (AAPA2). Technical Report 98027, Arca Systems / Exodus Communications, Ithaca, NY, June 1999.
- [12] 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.
- [13] J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0. A continually updated library, available at www.cs.york.ac.uk/~jac/, of protocols analyzed in the literature.
- [14] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Workshop on Formal Methods and Security Protocols*, Trento, Italy, July 1999.
- [15] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *Proceedings of the Symposium on Security and Privacy*, pages 234–248, Oakland, CA, May 1990. IEEE.
- [16] 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.
- [17] C. Meadows. The NRL Protocol Analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
- [18] J. Millen. CAPSL web page. At www.csl.sri.com/~millen/caps1, 1998.
- [19] J. Millen, S. Clark, and S. Freedman. The Interrogator: Protocol security analysis. *IEEE Trans. on Software Engineering*, SE-13(2):274–288, February 1987.
- [20] K. Slind. HOL90.7. At research.att.com/dist/ml/ho190/ho190.7.tar.gz, 1994.
- [21] K. Slind. HOL98. At www.cl.cam.ac.uk/Research/HVG/FTP, 1998.