Queensland University of Technology
Brisbane Australia

This is the author's version of a work that was submitted/accepted for publication in the following source:

Boyd, Colin & Viswanathan, Kapaleeswaran (2003) Towards a Formal Specification of the Bellare-Rogaway Model for Protocol Analysis. In Abdallah, A, Ryan, P, & Schneider, S (Eds.) *Formal Aspects of Security - FASec 2002*, December, 2002, London, United Kingdom.

This file was downloaded from: http://eprints.qut.edu.au/25095/

Notice: *Changes introduced as a result of publishing processes such as copy-editing and formatting may not be reflected in this document. For a definitive version of this work, please refer to the published source:*

Towards a Formal Specification of the Bellare-Rogaway Model for Protocol Analysis

Colin Boyd and Kapali Viswanathan

Information Security Research Centre,
Queensland University of Technology, Brisbane, Australia
{boyd,kapalee}@isrc.qut.edu.au

Abstract. We propose a way to unify two approaches to analysis of protocol security, namely complexity theoretic cryptographic analysis and formal specification with machine analysis. We present a specification in Sum of the Bellare–Rogaway cryptographic model and demonstrate its use with a specification of a protocol of Jakobsson and Pointcheval. Although this protocol has a proof in the Bellare-Rogaway model, its original version was flawed. We show that our specification can be used to find the flaw.

1 Introduction

Even though the problem of key establishment is now widely understood in its basic form, new protocols continue to be proposed for specialised situations such as use of devices of low computational power, multi-user groups, and where the shared keys are low entropy passwords. Meanwhile, more complex protocols for application areas like electronic payments, electronic auctions and fair exchange of assets are too difficult to be handled in full generality by most analysis techniques. There is therefore still a pressing need for improved methods for protocol analysis.

The problem of how to gain confidence in the security of protocols has been approached by two different research communities: the cryptography community and the computer security community. The cryptography community has built on the definitions for cryptographic primitives to provide security proofs for a relatively small number of protocols. The computer security community has generally used formal methods to specify protocols which are generally used in one of two ways. The first way is to search for insecure points within the state space using a model checker such as FDR [15] or Murϕ [16]. The second way is to use a theorem prover, such as Isabelle [17], to arrive at results which are true in all states.

Both cryptographic analysis and formal specification have their strengths and limitations, but the two communities have worked almost independently and there are even contradictions in what can constitute a secure protocol. A major issue is how cryptography is handled. In the cryptography community proofs are usually *reduction* proofs for which the aim is to show that if the protocol is

broken then some computationally difficult problem (or perhaps a generic cryptographic primitive) can be broken. The computer security community generally has assumed a model of "perfect cryptography" which means that it is *impossible* to obtain a ciphertext from a plaintext, or a plaintext from a ciphertext, without the correct key. Apart from ignoring probabilistic outcomes, this also means that authentication and confidentiality are not treated independently and that different kinds of confidentiality cannot be differentiated. A second major difference is that proofs in the cryptography community are human generated ("mathematician's proofs") rather than machine checkable proofs which can be obtained with software theorem provers.

This paper contributes to unifying the two approaches by suggesting and exploring a way of combining the ideas from the two communities. Abadi and Rogaway [1] have examined the difference in approach between the "perfect cryptography" assumption and cryptographic definitions of confidentiality. However, no attempt seems to have been made to analyse the same adversarial model with the two different approaches. We believe that this is a promising way forward. In this paper the Bellare-Rogaway model used for cryptographic proofs is formally specified, animated and analysed in a machine checkable way. The long-term aim of this is to gain the benefits of both worlds, by allowing proofs which are more accessible, more meaningful and more likely to be correct. This ultimately means a greater assurance in the security of protocols used to protect our communications.

We illustrate the potential use of our formal specification by using it to explore a protocol of Jakobsson and Pointcheval [14] that was proved secure in Bellare and Rogaway's model. Despite this proof the original version of the protocol turned out to be flawed, as shown by Wong and Chan [18]. We demonstrate that the flaw could be revealed by machine analysis of our specification. We contend that this demonstrates the value of a combined approach to the problem of formal analysis of protocols.

The remainder of this paper is structured as follows. In the rest of this section we outline the Bellare–Rogaway model and the formal language and tool that we used. In Section 2 we give more details of the Bellare–Rogaway model and describe how we captured the model in the formal specification. Section 3 describes the Jakobsson–Pointcheval protocol and our formal analysis of it.

1.1 The Bellare–Rogaway Model

In the 1990s academic cryptography started its move towards a mature science by demanding new standards of proof. In the last decade commonly accepted formal definitions have been established for the main cryptographic services such as confidentiality, integrity and signature, and many specific algorithms have been proven to satisfy such definitions on the assumption of the computational difficulty of well established problems such as integer factorisation and discrete logarithms. Proofs for cryptographic protocols have taken longer to establish than those for the primitives that they use.

An important direction in cryptographic protocol research was pioneered by Bellare and Rogaway in 1993 [5] when they published the first mathematical proof that a simple entity authentication protocol was secure. This work, which covered only the two-party case, was followed up with a paper on server-based protocols [3] and various authors have extended the same idea to include public-key based key transport [6], key agreement protocols [7], password-based protocols [4, 8], and conference key protocols [10, 9].

The general approach is to produce a mathematical model that defines a protocol in which a powerful adversary plays a central role. The adversary essentially controls all the principals and can initiate protocol runs between any principals at any time. Insider attacks are modelled by allowing the adversary to corrupt any principals, and the adversary can also obtain previously used keys. Cryptographic algorithms may be modelled either with generic properties or as specific transformations. Security of protocols is defined in terms of matching conversations (for authentication) and indistinguishability (for confidentiality of keys).

Because cryptographic protocols are notoriously difficult to design correctly, a proof of security is a very valuable property. Nevertheless, there are as yet relatively few protocols available which have a security proof. Most new protocol designs continue to be published without any attempt to prove security, leading to the traditional cycle of protocol attack being followed by a fix before a new attack is found. Some of the reasons for this are the following.

Proofs are difficult to understand. Security proofs tend to be difficult to understand for the average practioner. They typically run to several pages of mathematical reasoning. The lack of accessibility of the proofs means that there are few people who check the proofs in any detail.

Proofs can be wrong. Although there are relatively few protocols with security proofs available, a number of protocols which have been proven secure have turned out to have significant security weaknesses. We give a detailed example of the protocol of Pointcheval and Jakobsson later.

The significance of proofs can be hard to assess. Protocol goals used in the Bellare–Rogaway model do not correspond closely to the traditional protocol goals of authentication and key establishment. In particular, a goal such as entity authentication is defined in terms of the protocol specific property of *matching conversations*. Protocols with different traditional properties cannot be differentiated by such a property [12].

A comparison between the Bellare–Rogaway model and analysis using formal specifications shows that they are largely complementary. Proofs of security are long and complex and therefore prone to error and difficult to understand. Formal specifications, on the other hand, have not used the complete cryptographic definitions. It therefore seems a natural way forward to combine the benefits of both approaches. The bridge to unify these separate domains is the model of the protocol and the adversary. The same model used in the mathematical proof will be formally specified. Understanding and validation of the model can also be enhanced through *animation* of the specification.

1.2 Sum and Possum

Because of the abstract nature of the model, we used a high level specification modelling the global view of the adversary. This means that we have no need to specify explicit communicating processes; instead we use a state based specification modelling operations as the possible adversary actions in the Bellare–Rogaway model. Our choice of formal language for the specifications was Z [2]. Some of the reasons for the choice of Z are the following.

- There are widely used and understood conventions for modelling state based systems in Z; this is the obvious way to structure a highly abstract protocol specfication. It is appropriate to specify the Bellare–Rogaway model as a state based specification since the adversary has a global view of the protocol which is updated following each action of the adversary.
- The notation and semantics of Z are based on set theory and first order predicate logic. This means that Z specifications are easily accessible for most computer scientists and mathematically trained users.
- Z is widely used in the research community and by practitioners, and is also approaching international standardisation. There are a number of friendly tools available to support development of Z specifications and animation which are freely available, at least for academic research purposes.

Animation is an established technique in software engineering to allow clients to verify that a formal specification corresponds to the real world expectations of the system. This informal process is part of the user acceptance process and allows the system functionality to be examined before implementation takes place. Many of these features make animation a valuable addition to the process of formal modelling of security protocols.

We used the software tool Possum. Possum was developed at the Software Verification Research Centre at the University of Queensland [13]. It provides animation of specifications written in Sum, which is essentially a markup of Z. Possum supports Z conventions for modelling state based systems and allows for manual as well as script based animations. It also supports a graphical front end to enable visual animation of specifications if required.

2 Specifying the Bellare–Rogaway Model

In this section we give an informal definition of the Bellare–Rogaway model and outline how the model was specified in Sum. We first consider the model of communication which governs what the adversary is allowed to do. We then explore the definition of security.

2.1 Model of communication

The model of communication used is independent of the details of the protocol and is the same for all protocols with the same set of principals and the same

protocol goals. The adversary controls all the communications that take place and does this by interacting with a set of *oracles*, each of which represents an instance of a principal in a specific protocol run. The principals are defined by an identifier U from a finite set and an oracle Π_U^s represents the actions of principal U in the protocol run indexed by integer s. Interactions with the adversary are called oracle *queries* and the list of allowed queries is summarised in Table 1. This list applies to the model appropriate for server based key transport protocols, as described in Bellare and Rogaway's 1995 paper [3]; additional queries are appropriate for other protocol types. We now describe each one informally.

$\mathsf{Send}(U, s, M)$	Send message M to oracle Π_U^s
$\mathsf{Reveal}(U, s)$	Reveal session key (if any) accepted by Π_U^s
$\mathsf{Corrupt}(U, K)$	Reveal state of U and set long-term key of U to K
$\mathsf{Test}(U, s)$	Attempt to distinguish session key accepted by oracle Π_U^s

Table 1. Queries available to the adversary in Bellare-Rogaway model

$\mathsf{Send}(U, s, M)$ - This query allows the adversary to make the principals run the protocol normally. The oracle Π_U^s will return to the adversary the next message that an honest principal U would do if sent message M according to the conversation so far. (This includes the possibilty that M is just a random string in which case Π_U^s may simply halt.) If Π_U^s accepts the session key or halts this is included in the response. The adversary can also use this query to start a new protocol instance by sending an empty message M in which case U will start a protocol run with a new index s.

$\mathsf{Reveal}(U, s)$ This query models the adversary's ability to find old session keys. If a session key K_s has previously been accepted by Π_U^s then it is returned to the adversary. An oracle can only accept a key once (of course a principal can accept many keys modelled in different oracles).

$\mathsf{Corrupt}(U, K)$ This query models insider attacks by the adversary. The query returns the oracle's internal state and sets the long-term key of U to be the value K chosen by the adversary. The adversary can then control the behaviour of U with Send queries.

$\mathsf{Test}(U, s)$ Once the oracle Π_U^s has accepted a session key K_s the adversary can attempt to distinguish it from a random key as the basis of determining security of the protocol. A random bit b is chosen; if $b = 0$ the K_s is returned while if $b = 1$ a random string is returned from the same distribution as session keys.

The $\mathsf{Send}(U, s, M)$ query implicitly assumes a specification of the protocol which is being analysed. This is provided in a rather informal manner in typical use of the Bellare–Rogaway model, by simply stating what each principal should

do on receipt of each protocol message. In our formal specification we give abstract operations corresponding to each possible message that can be invoked by the adversary. We explicitly define instances of each principal.

The $\mathsf{Reveal}(U, s)$ query corresponds to an explicit operation in the specification. Figure 1 shows the schema. The principal and instance are chosen as inputs and if the instance has accepted then its name is added to the set of revealed instances and the key it accepted is added to the set of exposed keys.

```
op schema Reveal is
dec
p? : Players;
i? : Instances
pred
(p?,i?) in accepted;
revealed' = revealed union {(p?,i?)};
exposedKeys' = exposedKeys union {entityKeys(p?,i?)};
changes_only{revealed,exposedKeys}
end Reveal;
```

Fig. 1. Sum schema specifying $\mathsf{Reveal}(U, s)$ query

We did not explicitly model the $\mathsf{Corrupt}(U, K)$ query in our specification since it was not needed to demonstrate the attack. This can easily be added, however, by allowing the long-term key of principal U to be added to the set of keys known to the adversary.

Since the $\mathsf{Test}(U, s)$ query models the probabilistic advantage of the adversary (discussed further below) we deliberately avoid modelling this query explicitly. Instead we look only for 'trivial' losses of session keys by recording which keys are known to the adversary. This allows us to keep the specification simple, while at the same time seems sufficient to capture mechanistic attacks which can be missed when concentrating on a complexity theoretic analysis.

2.2 Defining Security

Success of the adversary is measured in terms of its *advantage* in distinguishing the session key from a random key after running the Test query. If we define Good-Guess to be the event that the adversary guesses correctly whether $b = 0$ or $b = 1$ then

$$\mathsf{Advantage} = 2 \cdot \Pr[\mathsf{Good\text{-}Guess}] - 1.$$

A critical element in the definition of security is the notion of the *partner* of an oracle, which captures the idea of the principal with which any oracle 'thinks' it is communicating. The way of defining partner oracles has varied in different papers using the Bellare–Rogaway model. In the more recent research partners have been defined by having the same session identifier (SID) which consists of a

concatenation of the messages exchanged between the two. Partners must both have accepted the same session key and recognise each other as partners.

The Test query may only be used for an oracle which has not been corrupted and which has accepted a session key that has not been revealed in a Reveal query. In addition partner of the oracle to be tested must not have had a Reveal query.

Bellare and Rogaway define a protocol in this model to be secure if:

1. when the protocol is run without any intervention from the adversary then both principals will accept the same session key.
2. Advantage is a computationally *negligible* function (it decreases faster than any polynomial function in terms of the length of the cryptographic keys).

The first condition is a completeness criterion that guarantees that the protocol will run as expected in normal circumstances. The second condition says that the adversary is unable to find anything useful about the session key after interacting in the specified way. One may ask how this condition relates to more conventional protocol goals. For example, key establishment protocols are typically required to provide *key authentication* which means that a principals should know which other principals have. or may have, the new session key. Although the above definition appears to be concerned only with key confidentiality it does imply key authentication. For suppose that the session key is known to an oracle Π_U^s different from that recorded in an oracle $\Pi_{U'}^t$ to be tested. Then Π_U^s is not the partner of $\Pi_{U'}^t$ and so it can be opened by the adversary and so the protocol cannot be secure.

A key is defined as fresh in the Bellare-Rogaway model if it has been accepted by an oracle which has not been opened, its partner (if any) has not been opened, and the user it represents has not been corrupted. Since the Test query can only be performed on oracles with fresh keys all keys accepted in a secure protocol must be fresh.

In the formal specification every schema operation may potentially change the state of the protocol. Rather than define security directly we found it more natural to test the *insecurity* of the state of the protocol in the schema operation *Insecure* shown in Figure 2. The set *exposedKeys* is specified to denote the set of exposed keys. The only mechanism that can update *exposedKeys* is the operation *Reveal*. The *Insecure* operation verifies that for every key in the set of exposed keys there exists an instance of a player (an oracle) such that the following conditions hold:

1. the oracle has accepted the exposed key;
2. the oracle has not been revealed (and has accepted the exposed key); and,
3. the partner of the oracle has not been revealed and has accepted the exposed key.

This means that a Reveal query can be used to obtain a key that should not be know by the adversary. The *Insecure* operation may be invoked at any time to test whether an insecure state has been reached.

```
op schema Insecure is
pred
exists k: Keys @ (k in exposedKeys) and
(exists p:Players; i:Instances @
((p,i) in accepted diff revealed) and (entityKeys(p,i) = k) and
(partnerOf(p,i) in accepted diff revealed));
changes_only{}
end Insecure
```

Fig. 2. Sum schema specifying an insecure state

3 Specification and Analysis

Jakobsson and Pointcheval's protocol [14] was designed specifically for use between a low-power computational device and a powerful server. It is a combination of a Diffie-Hellman exchange and a Schnorr signature. Computation takes place in a subgroup of $\mathbb{Z}_p^*$ where p is a suitable large prime. The element g generates the subgroup which has prime order q with $q|(p-1)$. In Figure 3 the low power entity A is able to complete all the computations required to send the first message off-line as long as the public key of the server, B, is previously available. This means that only simple calculations need to be performed by A during the protocol run itself. A by-product of the improved efficiency is that forward secrecy is not provided, since once x_B is compromised any previously agreed session keys become available.

Shared Information: Three hash functions h_1, h_2, h_3 with outputs of lengths l_1, l_2, l_3 respectively. Security parameter k (suggested value $k = 64$).

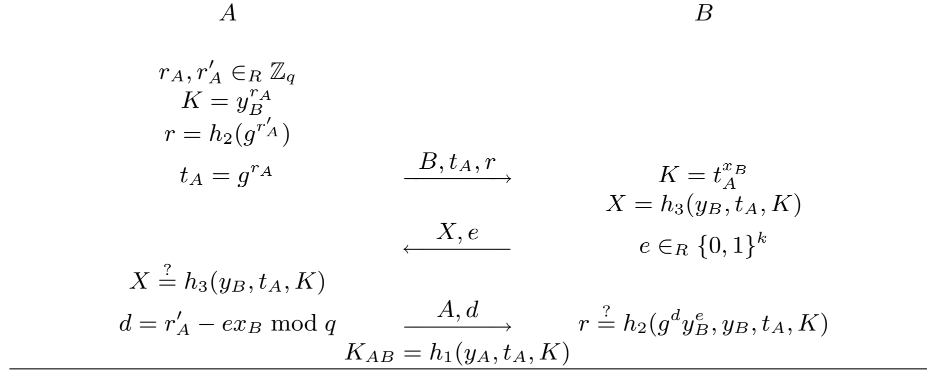

Fig. 3. Jakobsson-Pointcheval protocol

Figure 3 shows a traditional informal description of the protocol. This is the original version. In the revised version the message field r was defined by

$r = h_2(g^{r'_A}, y_B, t_A, K)$; this change connects the signature (d, r) to the session key to prevent an impersonation attack discovered by Wong and Chan [18]. The protocol, in both its original and revised versions, carried a proof of security in the Bellare–Rogaway model.

We specified the Jakobsson–Pointcheval protocol in Sum in a very abstract manner. We used the Possum animator to explore the specification and demonstrate the attack. The full specification is present in Appendix A.

3.1 Data types and protocol state

The data types that are central to the specification can be broadly classified into the following categories:

1. **Names for every instance of every player**. Every oracle, which is an instance of a protocol principal, is specified by an ordered pair of the form $(playerNumber, instanceNumber)$. When any two instances have the same *playerNumber*, it is assumed that both the instances contain the same long-term secret values that the player is specified to possess. The data type for these 2-tuples is named $Entities == Players \times Instances$.
2. **Names for every type of message that could be exchanged in the protocol**. Every message, which the original protocol design specifies for communication, is specified as a type. Therefore, the specification contains as many message types as the number of messages communicated during a single protocol run without any intervention from the adversary. Every communication of the protocol can then be specified as a collection of message types sent by an oracle to another oracle.

The state of the protocol is specified to hold a set of datasets which provides *a global view* of the protocol that an adversary could possess. The person animating the specification would then have the view of an adversary who has access to every communicated message by every entity and who has the capability to induce any entity to change state by providing the appropriate inputs. The adversary can obtains such inputs either by choosing random elements belonging to a particular message type or by using oracle queries to various entities, which are represented as schema operations. The schema operations are the only mechanisms that can be used to change the state of the protocol.

3.2 Specification of secret values

The cryptographic operations using secret keys that various entities possess are not explicitly modelled. Rather, they are modelled as operations that a valid entity, which possesses a set of valid secret keys, can perform. For example, the oracles that are modelled by schema operation *SendMessage1* are assumed to possess the long-term private key of the server. Therefore, this schema specifies the capability to compute shared Diffie-Hellman keys, which specified using the function $dhKeys(.)$. Thus, this schema specifies operations that any instance

of the server would perform when it receives the first communication of the protocol run. Similarly, the entities that are modelled by the schema operation *SendMessage2* are assumed to possess the long-term private key of the corresponding client. Therefore, this schema specifies the capability to perform Schnorr signature operations by the following operations:

```
.......
d' = d + 1;
dSeenBy' = dSeenBy func_override {(p?, j?) |--> d};
.......
```

That is the ability to update the signature counter, specified by d, and the "seen by" function specified by *dSeenBy*, which is specified to contain either the set of valid signatures that valid entities generated during some past operation or the set of claimed signatures that a particular entity received. The claimed signatures will be valid if the signature is claimed to be from an entity that indeed generated the signature, or invalid if the signature is claimed to be from an entity that did not generate the signature. The schema operation *SendMessage3* specifies a signature verification operation using the *dSeenBy* function along with some other functions. That is, valid signature generators, who possess the corresponding long-term private key, can *write* to the *dSeenBy* dataset and any verifier can *read* the *dSeenBy* dataset.

3.3 The initialisation operation

The protocol state is initialised using the schema named *init*. The initialisation function deletes the memory of every oracle, initialises the counters for the respective message types, initialises the lists of accepted and revealed oracles, and updates the *partnerOf* function.

Immediately after initialisation, every oracle is a partner of itself. The reason for this initialisation is to allow a consistent definition of the security of the protocol state as discussed in Section 2.2. This is because security depends on both the oracle and its partner (which may or may not exist) not having been asked a *Reveal* query. The schema operation *MatchPartner*, shown in Figure 4 is the only mechanism to alter the behaviour of the *partnerOf* function. The *MatchPartner* schema operation will update the *partnerOf* function only when certain preconditions are satisfied, which ensure that both parties possess the same session identifier.

3.4 Analysis of the specification

When the command can be executed, or when the preconditions are met, the animator provides an output (or a solution). The solution is the state information before and after the operation. Figure 5 shows a series of animation commands that were executed. Each numbered command is a user input, and the number is incremented if the command is successful. The outputs are not provided here

```
op schema MatchPartner is
dec
i?: Instances;
client?: Players
pred
exists c: Instances |
    (bSeenBy(0,i?) = bSeenBy(client?,c)) and
    (aSeenBy(0,i?) = aSeenBy(client?,c)) and
    (rSeenBy(0,i?) = rSeenBy(client?,c)) and
    (dSeenBy(0,i?) = dSeenBy(client?,c)) and
    (eSeenBy(0,i?) = eSeenBy(client?,c)) @
    (partnerOf' = partnerOf func_override {(0,i?) |--> (client?,c),
    (client?,c) |--> (0,i?)});
// this server instance thinks that client? is its partner
changes_only{partnerOf}
end MatchPartner;
```

Fig. 4. Sum specification of partner function

to save space; instead the string "solution" is inserted to represent the state information. When there is no solution the animator outputs the string "no solution." Finally an insecure state is reached revealing a successful attack. (In fact the attack shown is different from the one given by Wong and Chan [18] although it exploits the same weakness.)

Note that during the initialisation phase, every oracle was specified to be a partner of itself. Therefore, if an oracle has accepted a session key and it is its own partner (that is the schema operation named *MatchPartner* fails to match a partner for this oracle), then the protocol state can become insecure when an appropriate Reveal operation is performed. Therefore, to provide security, it is important that the schema operation *SendMessage3* has a solution, if the corresponding *MatchPartner* schema operation has a solution. The correspondence between these schema operations is specified by the input variable denoting the instance of a server oracle ($i?$) and the input variable denoting the client (*client?*). The proposed "fix" to the protocol by Jakobsson and Pointcheval adopts such a mechanism by specifying a similar sequence of operations for the *SendMessage3* and *MatchPartner* schema operations.

4 Conclusion

We have demonstrated that the Bellare–Rogaway model can be specified formally in a state-based specification with a simplified security definition. Using an example we have shown that this model could have found attacks that were missed even when protocols have been proven secure in the model. We believe that such an approach complements the human derived proofs. Furthermore the specification can help to clarify the meaning of these proofs, and animation of the specification allows the model to be more accessible to practitioners.

```
4 MutualAuth: StartClient{1/p?,0/j?}
solution
5 MutualAuth: StartClient{2/p?,0/j?}
solution
6 MutualAuth: SendMessage1{0/i?,10/b?,31/r?}
solution
7 MutualAuth: SendMessage1{1/i?,11/b?,30/r?}
solution
8 MutualAuth: SendMessage2{1/p?,0/j?,20/a?,41/e?}
solution
9 MutualAuth: SendMessage2{2/p?,0/j?,21/a?,40/e?}
solution
10 MutualAuth: SendMessage3{0/i?,2/client?,51/d?}
solution
11 MutualAuth: SendMessage3{1/i?,1/client?,50/d?}
solution
12 MutualAuth: MatchPartner{0/i?,1/client?}
no solution
12 MutualAuth: MatchPartner{1/i?,1/client?}
no solution
12 MutualAuth: MatchPartner{1/i?,2/client?}
no solution
12 MutualAuth: MatchPartner{0/i?,2/client?}
no solution
12 MutualAuth: Insecure
no solution
12 MutualAuth: Reveal{2/p?,0/i?}
solution
13 MutualAuth: Insecure
solution
```

Fig. 5. Animated attack on the Sum specification

We regard the work in this paper as a first step, demonstrating the potential of unifying cryptographic proofs and formal specifications. There are a number of ways that we are planning to extend this work.

- We intend to conduct a similar analysis of other protocols to gain a better understanding of how best to use the model.
- We plan to explore protocols which use different cryptographic properties. These will be modelled using different oracles available to the adversary, particularly encryption and decryption oracles.
- We would like to experiment with use of purpose-built model checkers to automate searching of the specification.
- We would like to use a theorem prover to provide machine checkable proofs at a high level to complement the human derived proofs.
- We intend to specify other models which provide cryptographic reduction proofs, particularly Canetti-Krawczyk's modular method [11].

The ultimate goal of this work would be a fully formalised proof of security for cryptographic protocols which is able to capture the probabilistic reduction to a known cryptographic primitive or computational problem. At present such a goal appears out of reach.

References

1. Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, to appear.
2. J. Sinclair B. Potter and D. Till. *An Introduction to Formal Specification and Z*. Prentice Hall, 1991.
3. M. Bellare and P. Rogaway. Provably secure session key distribution – the three party case. In *Proceedings of the 27th ACM Symposium on the Theory of Computing*, 1995.
4. Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In *Advances in Cryptology - Eurocrypt 2000*, pages 139–155. Springer-Verlag, 2000.
5. Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *Advances in Cryptology – CRYPTO'93*, pages 232–249. Springer-Verlag, 1993. Full version at `www-cse.ucsd.edu/users/mihir`.
6. S. Blake-Wilson and A. Menezes. Security proofs for entity authentication and authenticated key transport protocols employing asymmetric techniques. In *Security Protocols Workshop*. Springer-Verlag, 1997.
7. Simon Blake-Wilson and Alfred Menezes. Authenticated Diffie-Hellman key agreement protocols. In *Selected Areas in Cryptography*, pages 339–361. Springer-Verlag, 1999.
8. Victor Boyko, Phillip MacKenzie, and Sarvar Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In *Advanced in Cryptology - Eurocrypt 2000*. Springer-Verlag, 2000.
9. Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. Provably authenticated group diffie-hellman key exchange – the dynamic case. In *Advances in Cryptology - Asiacrypt 2001*, pages 290–309. Springer-Verlag, 2001.

10. Emmanuel Bresson, Olivier Chevassut, David Pointcheval, and Jean-Jacques Quisquater. Provably authenticated group Diffie-Hellman key exchange. In *CCS'01*, pages 255–264. ACM Press, November 2001.
11. Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *Advances in Cryptology – Eurocrypt 2001*, volume 2045 of *LNCS*, page ?? Springer-Verlag, 2001. `http://eprint.iacr.org/2001/040.pg.gz`.
12. Dieter Gollmann. Authentication – myths and misconceptions. *Progress in Computer Science and Applied Logic*, 20:203–225, 2001.
13. Dan Hazel, Paul Strooper, and Owen Traynor. Possum: An animator for the Sum specification language. In *Asia-Pacific Software Engineering Conference and International Computer Science Conference*, pages 42–51. IEEE Computer Society, 1997.
14. Markus Jakobsson and David Pointcheval. Mutual authentication for low-power mobile devices. In *Proceedings of Financial Cryptography*, pages ??–?? Springer-Verlag, 2001.
15. Gavin Lowe. Breaking and fixing the Needham-Schroeder public key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer-Verlag, 1996.
16. John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using Murϕ . In *IEEE Symposium on Security and Privacy*, pages 141–151. IEEE Computer Society Press, 1997.
17. Lawrence Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
18. Duncan S. Wong and Agnes H. Chan. Efficient and mutually authenticated key exchange for low power computing devices. In *Advances in Cryptology - Asiacrypt 2001*, pages 272–289. Springer-Verlag, 2001.

A Sum specification of Jakobsson-Pointcheval protocol

```
module MutualAuth is

// The following types are used to name the players in the protocol.

Players == 0 .. 4;  // Player 0 is the server
                    // and the remaining players are the clients.
                    // So, this spec considers a server and four clients.
Instances == 0 .. 4;
Entities == Players cross Instances;

// The following types of messages are communicated in the protocol.

MessageB == 10 .. 19;
MessageA == 20 .. 29;
MessageR == 30 .. 39;
MessageE == 40 .. 49;
MessageD == 50 .. 59;
```

```
// The following type represents the "index" of the cryptographic keys.

Keys == 60 .. 69;

//  The following schema represents the state of the protocol
// at any point of execution. The state is basically a collection
// of messages that have been communicated and the status of every
// instance of every player.

schema state is
dec
b: MessageB;
a: MessageA;
r: MessageR;
e: MessageE;
d: MessageD;
k: Keys;
dhKeys: MessageB -|-> Keys;
bSeenBy: Entities -|-> MessageB;
aSeenBy: Entities -|-> MessageA;
rSeenBy: Entities -|-> MessageR;
eSeenBy: Entities -|-> MessageE;
dSeenBy: Entities -|-> MessageD;
entityKeys: Entities -|-> Keys;
partnerOf: Entities -|-> Entities;
accepted: power Entities;
revealed: power Entities;
exposedKeys: power Keys
end state;

// The following schema initialises the state.
// 1) All message counters are reset to their initial values.
// 2) The sent message memory is erased.
// 3) The partnerOf function is initialised so that every entity
//  is a parter of itself.

schema init is
pred
b' = 10;
a' = 20;
r' = 30;
e' = 40;
d' = 50;
k' = 60;
dom bSeenBy' = {};
dom aSeenBy' = {};
dom rSeenBy' = {};
dom eSeenBy' = {};
dom dSeenBy' = {};
dom dhKeys' = {};
```

```
dom entityKeys' = {};
// Initially the dummy player is the partner for all instances.
partnerOf' = {
((0,0),(0,0)),((0,1),(0,1)),((0,2),(0,2)),((0,3),(0,3)),((0,4),(0,4)),
((1,0),(1,0)),((1,1),(1,1)),((1,2),(1,2)),((1,3),(1,3)),((1,4),(1,4)),
((2,0),(2,0)),((2,1),(2,1)),((2,2),(2,2)),((2,3),(2,3)),((2,4),(2,4)),
((3,0),(3,0)),((3,1),(3,1)),((3,2),(3,2)),((3,3),(3,3)),((3,4),(3,4)),
((4,0),(4,0)),((4,1),(4,1)),((4,2),(4,2)),((4,3),(4,3)),((4,4),(4,4))
};
exposedKeys' = {};
accepted' = {};
revealed' = {}
end init;

//  $Send({\cal B},j,``start'')$

op schema StartClient is
dec
p? : Players;
j? : Instances
pred
p? > 0 ;
changes_only{bSeenBy,rSeenBy,dhKeys,entityKeys,k,b,r};
bSeenBy' = bSeenBy func_override {(p?,j?) |--> b};
rSeenBy' = rSeenBy func_override {(p?,j?) |--> r};
dhKeys' = dhKeys func_override {b |--> k};
entityKeys' = entityKeys func_override {(p?,j?) |--> k};
k' = k + 1;
b' = b + 1;
r' = r + 1
end StartClient;

//  $Send({\cal A},i,({\cal A},B,r))$

op schema SendMessage1 is
dec
i?: Instances;
b?: MessageB;
r?: MessageR
pred
//br' = br func_override {i? |--> (b,r)};
entityKeys' = entityKeys func_override {(0,i?) |--> dhKeys(b?)};
a' = a + 1;
e' = e + 1;
aSeenBy' = aSeenBy func_override {(0,i?) |--> a};
eSeenBy' = eSeenBy func_override {(0,i?) |--> e};
bSeenBy' = bSeenBy func_override {(0,i?) |--> b?};
rSeenBy' = rSeenBy func_override {(0,i?) |--> r?};
changes_only{entityKeys,a,e,aSeenBy,eSeenBy,bSeenBy,rSeenBy}
end SendMessage1;
```

```
// $Send({\cal B}, j, (A, e))$

op schema SendMessage2 is
dec
p?: Players;
j?: Instances;
a?: MessageA;
e?: MessageE
pred
p? > 0; //this message is not sent to the server
d’ = d + 1;
dSeenBy’ = dSeenBy func_override {(p?,j?) |--> d};
aSeenBy’ = aSeenBy func_override {(p?,j?) |--> a?};
eSeenBy’ = eSeenBy func_override {(p?,j?) |--> e?};
// The check is $A = H_2 (y_A, B, K)$ is performed by server by
// checking if the corresponding values exist in memory of some
// server instance. $y_A$ is not specified here as it is assumed to be
// redundant.
exists s:Instances @ (entityKeys(p?,j?) = entityKeys(0,s)) and
                     (aSeenBy(0,s) = a?);
accepted’ = accepted union {(p?,j?)};
changes_only{d,dSeenBy,aSeenBy,eSeenBy,accepted}
end SendMessage2;

//  $Send({\cal A}, i, ({\cal B}, d))$

op schema SendMessage3 is
dec
i?: Instances;
client?: Players;
d?: MessageD
pred
client? > 0 ;
exists c: Instances @ (rSeenBy(0,i?) = rSeenBy(client?,c)) and
                      (eSeenBy(0,i?) = eSeenBy(client?,c)) and
                      (dSeenBy(client?,c) = d?);
dSeenBy’ = dSeenBy func_override {(0,i?) |--> d?};
accepted’ = accepted union {(0,i?)};
changes_only{dSeenBy,accepted}
end SendMessage3;

// The following schema is used to match the partners of various
// instances. This schema suitable adjusts the partnerOf function.

op schema MatchPartner is
dec
i?: Instances;
client?: Players
pred
```

```
exists c: Instances |
    (bSeenBy(0,i?) = bSeenBy(client?,c)) and
    (aSeenBy(0,i?) = aSeenBy(client?,c)) and
    (rSeenBy(0,i?) = rSeenBy(client?,c)) and
    (dSeenBy(0,i?) = dSeenBy(client?,c)) and
    (eSeenBy(0,i?) = eSeenBy(client?,c)) @
    (partnerOf' = partnerOf func_override {(0,i?) |--> (client?,c),
    (client?,c) |--> (0,i?)});
// this server instance thinks that client? is its partner
changes_only{partnerOf}
end MatchPartner;

// $Reveal($\cal U$, i)$

op schema Reveal is
dec
p? : Players;
i? : Instances
pred
(p?,i?) in accepted;
revealed' = revealed union {(p?,i?)};
exposedKeys' = exposedKeys union {entityKeys(p?,i?)};
changes_only{revealed,exposedKeys}
end Reveal;

// The following schema is not a part of the Bellare-Rogaway model.
// It is used to test if the protocol state is insecure. The protocol
// is insecure if there exists an exposed key such that there exists
// an instance of a player such that: 1) that player has accepted the
// exposed key and did not reveal the exposed key; and,
// 3) the partner of that player has accepted the "some" key and
// did not reveal the exposed key.

op schema InSecure is
pred
exists k: Keys @ (k in exposedKeys) and
(exists p:Players; i:Instances @
((p,i) in accepted diff revealed) and (entityKeys(p,i) = k) and
(partnerOf(p,i) in accepted diff revealed));
changes_only{}
end InSecure

end MutualAuth
```