Formal representation and reasoning approaches in modelling cryptographic protocols

Andrei Lapets

May 7, 2008

1 Introduction

We survey at a very high level some examples of existing work involving the formal verification of the properties of cryptographic protocols. For the purposes of this survey, formal verification includes the use of logics, proof systems, and inductive reasoning. The goal of this survey is twofold. First, it reviews a few existing attempts found in the literature to model cryptographic protocols in order to verify their security , and identifies the decisions made by researchers attempting to apply formal modelling tools. Second, it assembles a detailed list of the features a formal modelling tool for cryptographic protocols should possess, and discusses the manner in which the formal modelling of various features of cryptographic protocols might be accomplished with such features.

It is important to note that there is a great deal of literature on security features and static analysis of security within general-purpose programming languages. However, the scope of this survey is restricted to literature which deals with static, formal models that are used in designing, representing, illustrating, and proving properties about protocols. It ignores work that deals with real-world secure programs and protocol implementations. Particularly, it is *not* the goal of this survey to suggest features for a practical programming language for concrete *implementations* of cryptographic protocols.

2 Formal Verification of Protocols: Examples

The literature uses various kinds of terminology in describing protocols. In order to make our discussions uniform, we adopt the following terminology: a *participant* can be as a program, process, user, or application instance which is running some protocol; a *protocol* is a specification of how one or more participants can and should behave behave; and a *protocol instance* is an execution of a protocol by one or more participants. A *role* is one of the possible roles a participant might play when participating in a protocol instance; a *message* is any communication between participants.

We now review a few examples from the literature in which the authors attempted to use formal tools (either in the form of proof systems or simply formal reasoning on paper) to model cryptographic protocols. We make mention of how authors decided to represent formally several aspects of a

protocol and its proof of security:

- representation of algorithms and primitive operations;
- representation of interactions between protocol participants;
- how proofs of security are defined over these representations.

This can be viewed as a high-level list of features which must be considered when creating a formal modelling tool for cryptographic protocols, and we will refine this list in detail in the next section.

2.1 Backes, Jacobi, Pfitzmann, and Waidner

In modelling a family of protocols which involve interaction [BJP02], the authors attempt to create a scheme for transmitting messages in a pre-determined order whose security can be verified formally in a manner that is sound with respect to the security characteristics of the underlying cryptographic primitives.

In their models, participants are represented as a set of machines, and messages between them are sent across named ports in the style of CSP [Hoa78], similar to the approach taken in related work by other authors [DS97]. Here, the authors represent all possible protocol executions as a probability distribution over the configurations of the participants and their security parameters. They define a notion of indistinguishability for such distributions, and this is then used in bisimulation arguments in which a real protocol execution is compared to an ideal protocol execution. Composition is achieved using a substitution argument: if a partially ideal protocol is secure, then replacing the ideal portion of that protocol with a concrete implementation whose behavior is not distinguishable yields a secure protocol.

In their models, the authors abstract away the underlying concrete hardness assumptions. Operations are represented as primitives, and security is ensured by assuming appropriate properties about those primitives. These properties then propagate into the probabilistic analyses of executions. In their model, the authors are forced to represent interactions as a space of states with transitions between them.

The authors make special note of the fact that some of their proofs are difficult to represent and verify formally because they contain probabilistic arguments. The authors would naturally be forced to create their own constructions for probabilistic arguments within the formal modelling tools [ORS92] they are using (namely, PVS).

In related work [BPW03], the authors construct a cryptographic library which provides abstract operations signatures and encryptions for participants. A "database" (effectively a list of states) is used to keep track of what each of the participants knows at any given time step, and the semantics of these primitives is defined in terms of this record. The authors manage to preserve the probabilistic nature of this semantics within their models of the library. It is interesting that the authors use types and type annotations. For example, cryptographic objects are tagged with their parameters, such as public keys. However, these annotations are mostly used for sanity checks, and are not an essential part of the probabilistic analyses or security proofs. Reduction proofs which

specify the probability of a protocol being broken are performed against the security properties of the primitives.

2.2 Paulson [Pau98]

Work by Paulson [Pau98] provides a good example of how a logic system can be used to inductively construct proofs which govern the properties of protocols.

The author represents interaction without using any kind of process calculus. The main focus of the model are the messages: protocol interaction instances are represented as a list of (possibly interleaved) exchanges of messages between agents interacting with each other. The space of possible sequences of actions is defined inductively: agents can take some set of possible actions given a starting state. Proofs of security are derived by proving invariants over the space of reachable states (that is, by induction over the possible next steps each agent can take in an interaction). Each state can contain information about the knowledge of the agents involved in a protocol. The messages themselves are defined as expressions composed of primitive abstract operations for hashing, encrypting, values, and keys. Algebraic properties which correspond to the underlying semantics of these operators are assumed or derived.

Unfortunately, this approach requires many simplifications, making the results of the formal models subject to many restrictions. Whether these restrictions would apply in an actual setting is difficult to determine, and this casts doubt on the usefulness of the model and accompanying formal proofs. Make more of the details and assumptions explicit (as, perhaps, axioms in a formal proof system) would help address these shortcomings, and may even lead to results which are more general.

2.3 Abadi and Rogaway [AR00]

The work of Abadi and Rogaway [AR00] focuses on the messages sent between participants. In this work, the authors make no attempt to explicitly model participants or the complex and interleaving interactions which may occur in a setting with many participants. Instead, their proofs deal with individual messages and their security. Participants communicate using an inductively defined message grammar which consists of abstract primitive cryptographic operations which stand in for the usual operations a cryptographic protocol might offer (e.g. “sign”, “encrypt”). The authors abstract away completely the computational assumptions (e.g. the discrete logarithm assumption) and concrete operations (e.g. computing exponents modulo a prime) underlying the messages and their security.

The security of messages is proven by first defining reduction rules from message descriptions to a probability ensemble which can then be used to make arguments about the indistinguishability of messages. The conversion of formal expressions into probability ensembles in this work is reminiscent of the techniques used by Jones et al. in modelling the value of financial contracts [JES00] and by Ramsey and Pfeffer in their definition of a stochastic lambda calculus [RP02] which enables statistical queries on probability distributions of possible computation outcomes. This work demonstrates that the ability to derive probability distributions and analyses of the outcomes of computations can benefit users who wish to model cryptographic protocols. These analyses are then linked directly to abstract versions of underlying hardness assumptions (that is, no mention of

real assumptions is made, such as the discrete logarithm problem, it is only assumed that a one-way function exists).

Overall, this work suggests that formal, partially automated reduction proofs involving assumptions about adversaries with limited computing power and probability ensembles are possible, and the authors go so far as to provide concrete reduction algorithms. This is encouraging, and it is worth investigating whether such analyses can be combined with more sophisticated models of interaction, and with concrete primitive operations and explicit hardness assumptions.

3 Possible Directions for Formal Modelling Tools

The literature on formally modelling and verifying cryptographic protocols suggests several interesting features which the designers of formal modelling tools might with to provide. We now compile a list of these features using a bottom-up approach, and specify how they would interact and rely on one another. This can be viewed as a rough sketch of the relationships between the different component grammars that a formal modelling language for cryptographic protocols might possess. A serious research effort in this direction might try to identify which of these can be represented using existing formal modelling tools, and which require the addition of new features, or the definition of entirely new modelling languages. Note that if different formal modelling tools are used to model different components of a protocol and its associated proofs, recognizing (and formally proving) the way in which these tools can be combined in a sound manner is essential. It is worth noting that new features which cater to this specific area may be applicable in other areas. Thus, pursuing the ideas listed below may have consequences beyond the problem of modelling cryptographic protocols.

This list is by no means exhaustive, nor is it the only way to organize a taxonomy for the approaches taken in the literature, but it does help direct future work by identifying what features may be suitable for addressing each of the issues which arise when modelling cryptographic protocols formally.

3.1 Typed Algorithms

Underlying everything, there must exist a typed language for algorithm descriptions which has familiar constructs found in popular programming languages, and in pseudocode (particularly, representing an algorithm as a set of states with corresponding transitions should not be the user's *only* option, but mathematical constructs such as sets and groups should be easy to incorporate in algorithm descriptions).

In this language, it should also be possible to specify spaces of possible executions based on a range of inputs. If security proofs are to be represented in a detailed and faithful manner, it is essential that it be possible to reason about spaces of possible computations or executions of algorithms (including probability distributions on possible computations). Thus, the type system must be rich enough to represent several characteristics of computations essential for proofs of security:

- probability distributions on the values of computed values;

- randomness assumptions on computed values (e.g. random variables).

Applications for such features can readily be found in other areas. For example, in computational learning theory, the PAC learning model relies on probabilistic arguments about computed values. Bayesian learning algorithms might also be represented in such a language.

The language and type system must also be capable of representing the mathematical tools and constructs found in all protocols. These might be concrete, or abstract:

- concrete mathematical constructs: sets, groups, arithmetic primitives (e.g. modulus operations in algebraic groups), and their associated types;
- abstract cryptographic primitives and operations: hashing, encrypting, decrypting, signing, verifying, and associated types reflecting their high-level security properties.

Any such language must have an associated evaluation semantics, and this semantics can then be extended when interaction between algorithms is introduced.

3.1.1 Existing Tools

It is not known whether existing features in the latest typed languages (such as dependent types) are expressive enough to allow the encoding of any of the above components, and this is worth investigating. It is likely that most of the necessary type annotations, including those governing lists of properties about randomness and probability distributions, can indeed be encoded using dependent types.

3.2 Interaction

As any interesting cryptographic protocol usually involves more than one participant (even if that participant is simply an adversary or environment observing an interaction instance), it is essential for any formal modelling language to provide ways to represent this interaction. Improving upon existing models of concurrent processes (such as CSP or the π-calculus) is a separate research discipline, but making such models easy to incorporate in formal models of protocols is just as essential as the ability to represent algorithms in a natural and familiar syntax. Regardless of the model chosen, the two important concepts about which users must be able to reason are spaces of possible interaction instances, and particular interactions instances.

3.2.1 Spaces of possible interaction instances

Representations of spaces of possible interaction instances are useful for proving that a certain property holds for all possible interactions, or that an interaction with a specific property or event has a particular probability of occurring. Such spaces can be represented in many ways, but we list the two common ones seen in the literature we have referenced above:

- non-deterministic reduction rules on a process algebra or calculus (e.g. CSP [Hoa78] or the π-calculus [SW01])
- a global, ordered list of messages (with possible restrictions on interleaving and order).

These two are related: both can be defined inductively by providing a number of possible next cases given a particular configuration of the participants. Proofs regarding the consequences of particular message lists could be built up inductively over the structure of such lists. Nonce values can be used to enforce the order in which participants playing certain roles should generate random values.

Typically, the messages used for communication between algorithms are related closely to the algorithm description language, because they consist of results of computations. The typed language of algorithms must thus be extended with the following features:

- a particular algorithm can be coupled with a *role* (e.g. prover, verifier) in a protocol, to represent the fact that a participant adopting that role must perform computations and send messages as specified by that particular algorithm;
- a particular algorithm or computed value can be coupled with the identity of a participant (to distinguish it from other participants);
- type annotations representing the owner of a computed value, or any participants which could influence that value, with special annotations for data from trusted or special participants (e.g. an algorithm might return a truly random string);
- type annotations representing the view/privacy of a computed value (i.e., who can see the value other than the owner);
- message-passing primitives (e.g. channels in a process calculus) which propagate appropriately the type annotations of values being sent and delivered.

It should be possible to use the type annotations governing view and ownership properties of values in conjunction with their probability distribution type annotations in constructing reductions proofs of security. For example, if a particular participant constructed its own private random value using a trusted source, under appropriate privacy assumptions, the type annotation for that value should reflect that only that participant knows it. When this value is revealed (or is used to pad other information which is then revealed), the type annotations for this revealed value could indicate that it reveals nothing else about the state of the participant.

3.2.2 Particular interaction instances

Descriptions of particular interactions can be just as useful in proofs about protocols. For example, the impossibility result for universal composition in the work of Canetti et al. [Can07] relies on a *particular* set of interaction events between three participants.

If a the spaces of possible interactions are already inductively defined, particular interactions should be obtained as a consequence of this. However, this may not guarantee that properties about a

participant running a particular protocol can be inferred from the way it behaves in an interaction instance. Allowing such inferences to be made (most likely in the form of type annotations on the computations of the participant) greatly improves the expressive ability of a formal modelling language.

3.2.3 Dependency Analysis and Algebraic Laws

Proofs of security involving multiple participants usually involve some determination of what agents can compute given the public information. The probability and ownership annotations which permit this might have several other interesting applications. For example, suppose that we wish to minimize the amount of data which must be transferred between participants while still ensuring the requirements of the protocol are met. This involves determining, given a collection of values and equations over those values, which subsets of values is derivable given some other subset of those values. Information flow/dependency graphs between variables could be used to address such questions, and the results could be applied to minimizing the sizes of messages passed between participants.

3.2.4 Example: DLA

For a simple example of the kind of protocol a formal tool should be able to model, let us consider an interactive protocol for proof of knowledge based on the discrete logarithm assumption. The protocol has two roles for two distinct participants: prover, and verifier. The prover wants to convince the verifier (with high probability) that it knows a secret. Their behavior is specified in the following manner:

- Prover:
  - Generate random prime p and generator $g \in \mathbb{Z}_p^*$ where $g^q = g$ for some q. All of these values are public.
  - Generate a secret key $x \in \mathbb{Z}_q$.
  - Select a random $r \in \mathbb{Z}_q$, and send to the verifier $g^r \bmod p$.
  - Wait for the verifier to send a value c, and return $a = r + cx \bmod q$.
- Verifier:
  - Wait for the prover to send r.
  - Select a random $c \in_R \mathbb{Z}_q$ and send it to the prover.
  - Once a is received from the prover, accept only if $g^a = Ry^c$.

Notice that the above protocol is simply a pair of algorithm descriptions, and any variant of a process calculus could be used to encode this algorithm. There are several facts about this protocol which must be proven: soundness (that an honest verifier will accept), completeness (that the prover indeed knows a secret with high probability), and zero-knowledge (that the verifier learns nothing

about x). It is worth investigating whether annotating the values chosen and computed by the algorithm above with probability and ownership types can help encode and verify these proofs, or even automate some parts of the proof-writing process. For this particular protocol, the discrete logarithm assumption is essential, as $g^r \bmod p$ is a public value.

When the above protocol is modified so that it works for *many* values (i.e. x becomes a vector $\overline{x}$), the naive approach is to send many different values between the two participants. However, through some manipulation of which participant sees the values, there is a way to ensure that the amount of data passed between the prover and verifier remains constant (and not linear in $|\overline{x}$). It is also worth investigating whether it is possible to use information flow analysis to model the data being sent between participants, as this may lead to an automated algorithm for finding the minimum amount of data which is needed to ensure soundness, completeness, and zero knowledge.

3.2.5 Other Examples

There are many other instances in proofs involving protocols where an algorithm's behavior is specified in an implicit manner. For example, in the proofs of universal composability in the work of Canetti et al. [Can07], proofs often begin with the specification of an *ideal* probabilistic behavior for one or more participants running certain algorithms. Then, given these assumptions, the behavior of these same algorithms in a different context is derived *algebraically* from the specification of their ideal behavior.

Unfortunately, there is no straightforward way to use existing tools to encode implicit algebraic specifications of black-box algorithms, to manipulate them and combine them with algorithm definitions, to combine them with a process calculus or other interaction scheme, and to statically verify the soundness of such manipulations. Any of the above examples can be used as a starting point for deriving a list of necessary features for a formal modelling tool.

3.3 Propositions, Assumptions, and Reductions

In the work reviewed, explicit uses of the underlying mathematical assumptions do not appear. One possible explanation is that the number of such underlying assumptions (e.g. RSA, the discrete logarithm) is small. However, if it were actually *easier* to encode these low-level assumptions using modelling tools than to represent them in some other manner, this criticism would become irrelevant.

Given the degree to which security proofs rely on underlying computational hardness assumptions, a formal modelling language should allow the user to specify concretely as many details of these assumptions as they wish. The security proof could then rely on abstract assumptions only to a degree that the user deems to be tolerable. Even if the number of such assumptions is small, this level of detail can provide a user more confidence in the proof of the security of a protocol, eliminating the need for abstraction if none is desired. It is also likely that features which allow this level of detail will be useful in modelling problems in other areas.

This functionality could be provided by allowing a user to encode propositions about spaces of algorithms. Depending on the level of detail, it would also be necessary to include primitives for

common algebraic constructions (e.g. primes multiplicative groups modulo primes), random values and probabilities distributions, and trusted values. These propositions could then be taken into account when statically analyzing claims about algorithm descriptions which involve the operations found in the descriptions of hardness assumptions. This would allow users to encode reduction proofs (e.g. given an algorithm capable of computing a result with a given probability, another algorithm can be constructed for computing a related, hard-to compute result), and to specify oracles.

3.3.1 Example

As an example, let us consider the discrete logarithm assumption. Let PPT be the space of probabilistic polynomial-time algorithms, let $P \subset \mathbb{N}$ be the set of prime numbers, and let N be the space of negligible functions of one argument. Then one version of the discrete logarithm assumption can be stated as follows:

$$\forall A : \text{PPT}.\forall p \in P.\forall g \in \mathbb{Z}_p^*.\forall x.\exists \eta \in N.\, \Pr[A(p, g, g^x \bmod p) = x] < \eta(\log p).$$

Notice that this is a proposition characterizing the space PPT. Many security proofs are proven by assuming a protocol is insecure (that there is an efficient algorithm A' which can break it under some conditions), and then calling the algorithm A' as a subroutine in an algorithm $A \in \text{PPT}$ which does not satisfy the above assumption.

Notice how such propositions rely on the ability to implicitly define the probabilistic behavior of an algorithm. Even within the assumption itself, we have the following subexpression:

$$Pr[A(p, g, g^x \bmod p) = x] < \eta(\log p).$$

This is a statement about the probability distribution of the results returned by the algorithm A (it can also be viewed as a probability distribution on the accuracy of this particular algebraic equality).

It is worth noting that no formal modelling tools exist which would make it unnecessary to abstract away the concrete impossibility assumptions (such as the discrete logarithm assumption) and the probabilistic arguments used in the proofs of cryptographic protocols. Thus, researchers are forced to assume that perfect cryptographic primitives exist, and composition arguments which involve such primitives thus seem less convincing to skeptics.

4 Discussion and Observations

We have identified at a high level several features of a formal modelling language that would be suitable for modelling cryptographic protocols. The formal modelling approaches used in the surveyed literature suggest that there is still a serious gap between the goals of the research community producing formal modelling and verification tools, and the problems which researchers in other areas of computer science (in this case, research on cryptographic protocol) want to model.

One major disadvantage is that most formal tools introduce a steep learning curve for a user wishing to construct a model and security proof. Even worse, once the model has been constructed, it is not easily comprehensible to anyone lacking specialized knowledge of the modelling tool. Particularly problematic (but often necessary) is the the translation of descriptions of straightforward algorithms into their formal counterparts within the languages of formal modelling tools. Given the fact that formal modelling tools are themselves languages, the difficulty of this translation is troubling (but understandable, given the strategies and stated goals of the designers of formal modelling tools). Users are routinely forced to encode computations as transitions over state spaces, despite the fact that in many cases, the pseudocode for their algorithms have very simple, even pure (stateless) structure. Such code can be translated mechanically into a set of predicates on states with relative ease, though such a translation would obviously make the generation of error messages at the time of verification more difficult.

It would be encouraging to see formal modelling tools which more readily cater to researchers in other areas by providing more friendly primitives and useful features. This includes features for describing algorithms, for describing interaction between many components (thus letting researchers avoid implementing their own models for passing messages), for describing probability distributions (as well as other properties) on values found in algorithms, and for verifying reduction and simulation proofs. Such features should be designed in a manner which does not obfuscate the arguments being encoded due to their representation in the model's syntax. Reasoning using the formal model should be *easier* than reasoning on paper, and the representation should elucidate the reasoning being used. Until these conditions are met, formal modelling tools are unlikely to be adopted outside the language design community on a wide scale.

It is also of interest whether any of these formal approaches can already be represented and implemented using language features found in existing systems. It is necessary to distinguish how existing formal representation and verification techniques can be used, and how existing techniques need to be adapted to be both more understandable and more effective when used for actual problems.

References

[AR00] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *TCS '00: Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*, pages 3–22, London, UK, 2000. Springer-Verlag.

[BJP02] M. Backes, C. Jacobi, and B. Pfitzmann. Deriving cryptographically sound im- plementations using composition and formally verified bisimulation, 2002.

[BPW03] M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations, 2003.

[Can07] Ran Canetti. Obtaining universally composable security: Towards the bare bones of trust. Cryptology ePrint Archive, Report 2007/475, 2007.

[DS97] Bruno Dutertre and Steve Schneider. Using a PVS embedding of CSP to verify authentication protocols. In *Theorem Proving in Higher Order Logics*, pages 121–136, 1997.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[JES00] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 280–292, New York, NY, USA, 2000. ACM.

[ORS92] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[Pau98] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[RP02] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Symposium on Principles of Programming Languages*, pages 154–165, 2002.

[SW01] Davide Sangiorgi and David Walker. *The Pi-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.