

Refinement of Security Protocol Data Types to Java

Holger Grandy, Kurt Stenzel, Wolfgang Reif

E-Mail: {grandy, stenzel, reif}@informatik.uni-augsburg.de

Abstract. In this paper we illustrate the mapping of abstract data types to a real programming language during a refinement of security protocol specifications. We illustrate that new security and correctness problems arise on the concrete level and show a possible solution.

1 Introduction

It is hard to get communication protocols secure on the design level. Research has brought a variety of methods to ensure security, for example model checking based approaches (e.g. [BMV03]), or specialized protocol analyzers (e.g. [Mea96]). Our approach for the analysis of security protocols uses Abstract State Machines [Gur95] and interactive verification with KIV, our interactive theorem prover. Details can be found in [HGRS06].

But ensuring security of a protocol on the design level is not sufficient. It is an equally important step to get a correct and secure implementation. The verification of sequential Java programs is supported in KIV with a calculus and semantics [Ste04]. Together with the established refinement approach of Downward Simulation which has been adapted to Abstract State Machines ([Sch01] [Bör03] [Sch05]) the correctness of a protocol implementation can be proven. An implementation is correct if it makes the same state changes and has the same input/output behavior as an abstract specification. We already developed a framework for the refinement of protocol specifications to Java ([GHR06]), its further elaboration is current work in progress. In this paper we illustrate which problems arise in this context and which solutions can be found.

2 Data Types for Security Protocols

One problem when refining a protocol specification to a concrete implementation is the mapping of abstract data types to the programming language. On the abstraction level of a protocol specification most verification approaches have the concept of typed messages like nonces, encrypted documents, signatures or hash values. Those types are used to ensure certain properties of the messages, e.g. that an encrypted document cannot be decrypted without knowledge of the key or that a plain text cannot be derived from a hash value. Those properties are crucial for the verification of security properties.

For example, in our verification framework, messages are modelled using an abstract data type *Document*. A part of its specification is given as an example:

Document = ... | **IntDoc**(value : int) | **HashDoc**(hash : Document) |
EncDoc(key : Key, doc : Document) |
Doclist(docs : Documentlist) | ...

A document is either an IntDoc that represents an integer, a HashDoc that represents the hash value of a document, or an EncDoc that represents a document which is encrypted with a key. A Doclist contains a list of other documents.

For the refinement a similar distinction of the type of the messages is crucial because it must be proven that the implementation produces the same output as the abstract specification claims (including the same type). When using the Java language it is a natural approach to use objects for the representation of messages. For example, the implementation for IntDocs would be:

```
public class IntDoc extends Document {
    private byte[] value;
    public IntDoc(byte[] val) { value = val; } ...}
```

The integer part of the abstract type IntDoc is implemented with a byte array which represents the arbitrarily large abstract integer. The other types of documents are described later on. When using such a implementation for security protocols, there must be a method for sending and receiving the Java document instances.

3 A Middleware for Data Type Mapping

The first idea when trying to map the abstract document to a concrete class instance would be to transform the abstract integer into a byte array and to construct an instance of class IntDoc with this array. Unfortunately this does not consider all the possibilities that are present in the real application. An attacker in the real world may have access to the communication subsystem and therefore may be able to send a Java IntDoc whose value field is null. When using the mapping above, the concrete object with a null value would not be considered during the refinement, because it cannot be constructed from an abstract IntDoc (the abstract int type has no null-value).

Another aspect is the encoding of an integer as a byte array. When mapping the abstract integer into a byte array an unambiguous representation for the abstract data type has to be used. This means that the byte array may not contain leading zeros. Again, the attacker is able to generate real objects that have no counterpart on the abstract level, i.e. a byte array with leading zeros.

This means that the attacker on the concrete level has more abilities than an attacker on an abstract level. This can lead to new implementation flaws (e.g. a Nullpointer Exception in the case above).

Another example of invalid messages on the concrete level is the possibility to have cyclic pointer structures in the Java programming language. Consider the implementation of the Doclist data type:

```

public class Doclist extends Document {
    private Document[] docs;
    public Doclist(Document[] docs) {this.docs = docs;} ...}

```

On the abstract level cyclic term generated data types are not possible. But in an implementation using pointer structures a cyclic reference from the array of documents contained in an instance of class Doclist to the instance itself is possible.

A refinement method has to deal with those invalid documents and it must be proved that these augmented attacker capabilities do not lead to new implementation flaws on the concrete level. Therefore we use a middleware based approach. The middleware must be capable of detecting whether a received message has an abstract counterpart. The middleware works by sending and receiving byte array representations of the Document class instances. Those representations are constructed using an encoding of type, length and value of the Document into a byte array. With this middleware another layer of abstraction is added to the protocol implementation. The property that the middleware checks whether an incoming message has an abstract counterpart document is used as a postulate in the refinement proofs, and can be verified in a separate step.

On the one hand, the middleware is used for the transformation of input from the communication subsystem into Java objects. On the other hand, it is also needed for the cryptographic operations. As an example, consider the implementation of the abstract data type EncDoc(key,doc) which models encrypted documents:

```

public class EncDoc extends Document {
    private byte[] crypt;
    public EncDoc(Key k, Document d) { crypt = Crypto.encrypt(k,d);} ... }

```

The abstract data type EncDoc contains both the key used for encryption and the plaintext document. The access functions for the data type specify that access to the plain text is only possible when the correct decryption key is given (the decrypt operation works). On the concrete level, the plain text and the key are not included, only the encrypted data. Encryption in the Java Cryptographic Architecture works on arrays of bytes. Because of that encrypted data is stored inside EncDocs as a byte array. To solve this mismatch, the encrypted *byte array encoding* of the plaintext Document doc is stored inside an EncDoc instance. When a decrypt operation is called in Java on the EncDoc instance again the middleware is called to decode the result of the decryption, which is also a byte array, back to a Java object. The assumption in this case is that for an invalid decryption (e.g. with the wrong key) no correct encoding of a document instance can result.

To prove correctness of the middleware, a predicate *isAbstractDoc* on Java pointer structures is formulated, which defines whether a pointer structure is a valid representation of an abstract document data type. Then the property for the receive operation in the middleware is that it returns the pointer structure corresponding to the abstract input document if *isAbstractDoc* holds and throws

an exception otherwise. The following theorem (which has been formally proved by KIV) ensures that exactly the abstract documents are covered:

isAbstractDoc holds for a Java pointer structure p if and only if there exists an abstract document whose transformation to Java leads to a pointer structure p' , which is behavioral equivalent to p .

A correct middleware which checks this predicate justifies to use a direct transformation from abstract documents to Java pointer structures (omitting the byte array representation) in the refinement proofs. This works by construction of the instances of the Java classes implementing the data type Document. Additionally, a placeholder type is added to the abstract document data type, which corresponds to all invalid cases on the concrete level. The refinement proof then proceeds by proving that the implementation performs the same state changes and shows the same input/output behavior when receiving invalid documents (exception case in the middleware) as the abstract specification does when receiving the placeholder document (specified as the error case on the abstract level). For all other valid documents there must be a (1,1)-relation.

Current work includes the verification of the middleware and its encoding strategy. This includes the proof that all invalid documents described above can be found using the encoding and that all valid encodings of documents are returned as correct pointer structures (the middleware checks *isAbstractDoc*). Furthermore we are working on a method to transfer this approach of implementing messages in security protocols by Java class instances to smartcards. A new challenge arises: allocating memory at run time must be avoided because of the missing garbage collection. We imagine a configurable middleware, that is initialized with only those document instances that are needed for implementing a certain security protocol and then reuses those instances during the different protocol runs. The reuse of those documents leads to new verification challenges such as the problem of pointer sharing.

4 Conclusion and Related Work

We illustrated the problem of mapping abstract data types to a concrete implementation during the refinement of communication protocols.

[MM03] describes a similar approach for Java Smart Cards. The authors specify protocols using a high level specification language for proving security properties and a more concrete one which works on the level of byte arrays. They specify lengths and contents of messages using byte arrays and then use static program analysis on the JavaCard implementation to decide whether the implementation is correct. Because of the automated analysis and the fact that implementation correctness is undecidable this approach cannot give reliable answers in every case.

[TH04] uses the Spi Calculus for specifying security protocols and a code generation engine to transform this specification to an implementation. They also map messages of the protocol to Java objects. Their mapping does not

address the problem of augmented attacker capabilities on the concrete level which is described in this paper.

The solution described in this paper uses a layered approach by adding middleware concepts to the refinement. The verification of correctness of the middleware implementation seems feasible and is current work in progress.

References

- [BMV03] David Basin, Sebastian Mödersheim, and Luca Viganò. An On-The-Fly Model-Checker for Security Protocol Analysis. In *Proceedings of Esorics'03*, LNCS 2808, pages 253–270. Springer-Verlag, Heidelberg, 2003.
- [Bör03] E. Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 15 (1–2):237–257, November 2003.
- [GHR06] Holger Grandy, Dominik Haneberg, Wolfgang Reif, and Kurt Stenzel. Developing Provably Secure M-Commerce Applications. In Günter Müller, editor, *Emerging Trends in Information and Communication Security*, volume 3995 of *LNCS*, pages 115–129. Springer, 2006.
- [Gur95] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9 – 36. Oxford Univ. Press, 1995.
- [HGR06] D. Haneberg, H. Grandy, W. Reif, and G. Schellhorn. Verifying Smart Card Applications: An ASM Approach. Technical Report 2006-08, Universität Augsburg, 2006.
- [Mea96] Catherine Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
- [MM03] R. Marlet and D. Le Metayer. Verification of Cryptographic Protocols Implemented in JavaCard. In *Proceedings of the e-Smart conference (e-Smart 2003)*, Sophia Antipolis, 2003.
- [Sch01] G. Schellhorn. Verification of ASM Refinements Using Generalized Forward Simulation. *Journal of Universal Computer Science (J.UCS)*, 7(11):952–979, 2001. URL: <http://hyperg.iicm.tu-graz.ac.at/jucs/>.
- [Sch05] G. Schellhorn. ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. *Journal of Theoretical Computer Science*, vol. 336, no. 2-3:403–435, May 2005.
- [Ste04] K. Stenzel. A formally verified calculus for full Java Card. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST) 2004, Proceedings*, Stirling Scotland, July 2004. Springer LNCS 3116.
- [TH04] B. Tobler and A. Hutchison. Generating Network Security Protocol Implementations from Formal Specifications. In *CSES 2004 2nd International Workshop on Certification and Security in Inter-Organizational E-Services at IFIPWorldComputerCongress*, Toulouse, France, 2004.