GENERATION, ANALYSIS AND VERIFICATION OF CRYPTOGRAPHIC PROTOCOL IMPLEMENTATIONS

Ben Tobler and Andrew Hutchison

University of Cape Town

Ben Tobler
btobler@cs.uct.ac.za
+27 +21 650 3127
Dept. of Computer Science
Private Bag
Rondebosch
7701

Dr A.C.M. Hutchison
hutch@cs.uct.ac.za
+27 +21 650 3127
Dept. of Computer Science
Private Bag
Rondebosch
7701

ABSTRACT

Network security is an area of increasing importance in commercial, public and private environments. Much research has been done in the area of design and analysis of the cryptographic protocols that provide this security. However, there has been little focus on research into the correctness of the implementations of these protocols, as is evidenced by the number of security flaws found in implementations of cryptographic protocols in commercial software systems on a regular basis. In this research project we investigate the development of a code generation tool for generating protocol implementations that can be proven to meet their specifications. Requirements for generating such high integrity code involve using a cryptographic protocol specification language that has formal semantics, ideally a target implementation language that also has formal semantics and a translation process between the two that is proven to preserve the meaning of the specification in the mapping to the implementation. The ability to automatically generate protocol implementations from their specifications will also facilitate analysis such as comparing the performance of protocols with the same goals and testing the scalability of protocols for secure group communication, as well verification of other existing implementations of protocols.

KEY WORDS

code generation, cryptographic protocol, cryptographic protocol analysis, network security

GENERATION, ANALYSIS AND VERIFICATION OF CRYPTOGRAPHIC PROTOCOL IMPLEMENTATIONS

1 INTRODUCTION

Security in networked environments is an area of increasing importance, as it makes electronic commerce, secure personal communications and other important activities possible. Much research has been conducted in the area of design and analysis of cryptographic protocols which provide secure communication with one or more of confidentiality, authentication, integrity and non-repudiation. This research has resulted in design guidelines [1], belief logics such as GNY [8] and BAN [4] for analysing the security properties of protocols and attack construction (e.g. using strand spaces [7] to determine if the protocol is vulnerable to attacks such as replay or man in the middle attacks).

While this helps the development of cryptographic protocols without vulnerabilities, there has not been much focus on the correct implementation of cryptographic protocols. As is repeatedly shown, cryptographic protocol implementations coded by human programmers are often error prone. Implementation flaws have been found in implementations of PPTP [13], SSL [14], RADIUS [10] and many other cryptographic protocols. While some security flaws are a result of poor coding practices, e.g. unchecked buffer access, many are a result of the protocol being incorrectly implemented or misunderstood.

Depending on the language used to specify the protocol, many of the actions required to verify that protocol run may progress from the current round to the next, are implicit and easily missed by the protocol implementer. Even if the specification language is formally defined and explicitly states each action to be performed, it is not unusual for aspects to be overlooked resulting in an incorrect and/or vulnerable implementation.

The initial aim of this research project is develop a code generation tool that will be able to automatically generate code that implements a cryptographic protocol, given the protocol's formal specification. In addition, the implementation must be able to be proven to be correct according to that specification. In this context we define a correct implementation as one that can be proven to perform all the steps required by the specification in order for a protocol run to complete correctly. This means that for each round in the protocol run, the implementation must be guaranteed to perform every action specified (e.g. verification of components in the received message) in order for the implementation to progress to the next round.

Proving the absence of flaws resulting from poor coding practice, such as unchecked buffer access - which may allow buffer overflow attacks against the implementation, does not fall within the scope of this project. Never the less, by choosing an implementation language that runs in a managed

environment and has built-in checked array types (e.g. Java), we hope to avoid most vulnerabilities of this type. Proving the correctness of the cryptographic algorithm implementations also falls outside the scope of this project.

The ability to automatically generate protocol implementations will facilitate analysis such as comparing the performance of protocols with the same security goals, testing the scalability of protocols for secure group communication and comparing the performance of various cryptographic library implementations. It will also provide a way to test other cryptographic protocol implementations, by automatically generating attack programs that check that the implementation correctly verifies each message before proceeding to the next round of the protocol. In addition it will allow us to demonstrate and verify the feasibility of protocol level attacks by generating code that implements them.

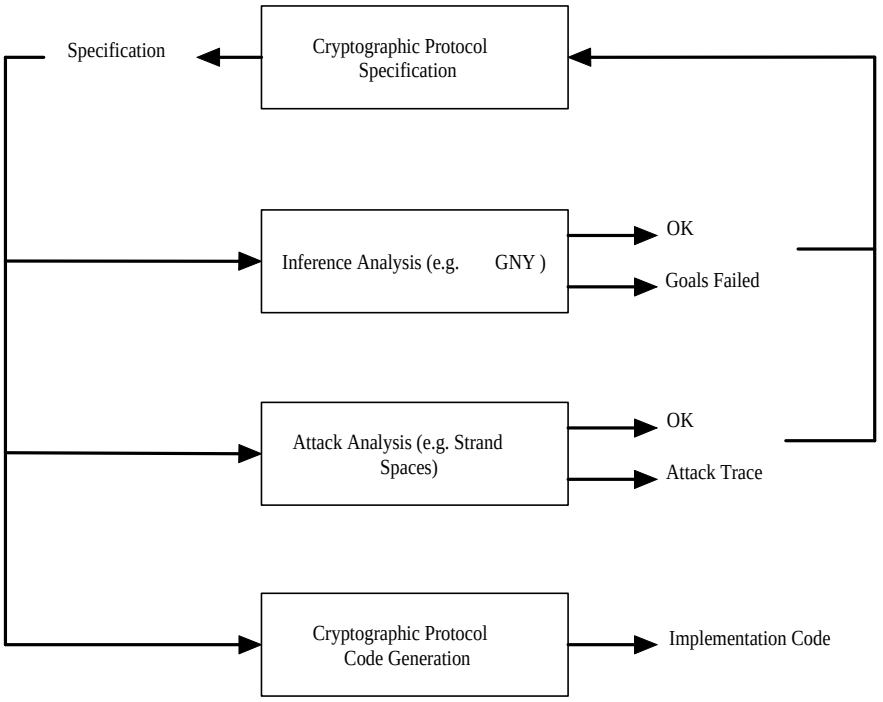

Figure 1: Overview of Cryptographic Protocol Development

Figure 1 places this project in context given the wider area of cryptographic protocol development. Previous, and current projects at the University of Cape Town, e.g. the SPEAR II [12] modeling and analysis tool, have addressed the areas of protocol specification and inference analysis using the GNY logic. This project will hopefully complement them by providing the final phase of protocol development, the actual implementation.

2 CODE GENERATION REQUIREMENTS

In order to meet the aims of this project, some guidelines or methods need to be followed when developing the code generation tool. Requirements for generating code for use in safety critical en-

vironments, by means of translating from a source specification language to a target implementation language, have been discussed by Whalen and Heimdahl in [15]. As code for safety critical environments and code that implements cryptographic protocols should have similar properties, i.e. a high level of integrity is required, the discussion in [15] is relevant to the development of the code generation tool.

They identify the five following requirements for high integrity code generation:

1. "*The source and target languages must have formally well-defined syntax and semantics.*" [15, page 2]
2. "*The translation between a specification expressed in a source language and a program expressed in a target language must be formal and proven to maintain the meaning of the specification.*" [15, page 3]
3. "*Rigorous arguments must be provided to validate the translator and/or generated code.*" [15, page 4]
4. "*The implementation of the translator must be rigorously tested and treated as high assurance software.*" [15, page 4]
5. "*The generated code must be well structured, well documented, and easily traceable to the original specification.*" [15, page 4]

Although we do not expect to be able to meet the 4th requirement, due to time and resource constraints, fulfilling the other requirements seem a reasonable minimum target, and they will be referred to in the following sections where the choice of specification language and implementation language is discussed.

3 CRYPTOGRAPHIC PROTOCOL SPECIFICATION LANGUAGE

3.1 Options

In choosing a specification language the 1st requirement of high integrity code generation should be born in mind, i.e. that it has formal semantics. As we want the translation processes to map from constructs in the specification language to constructs in the implementation language that have the same meaning, we need to know the formal meaning of the constructs in the specification language, and having formally defined semantics provides exactly this.

The Standard Notation (SN), often used in texts on cryptographic protocols, does not meet this criterion. While SN is fairly intuitive and easy to read, it does not provide a means to specify the goal

of the protocol, the initial assumptions or beliefs of the principals. As it lacks formal semantics, the assertions that need to be made at each round of a specified protocol are implied and may be missed by the reader. To address exactly these shortcomings, languages such as the Common Authentication Protocol Specification Language (CAPSL) [5] and Cryptographic Protocol Analysis Language (CPAL) [16] have been developed. They both have formally defined semantics and incorporate ways to specify assumptions, beliefs, initial possessions and goals of protocols.

3.1.1 CAPSL and CIL

CAPSL has been developed specifically for use with cryptographic protocol analysis tools. There are already a number of tools that use it to specify input, such as those described in [3] and [11]. Among its strengths are that it uses a syntax that closely resembles SN and is thus easily humanly writable and readable. It also has an intermediate form, CAPSL Intermediate Language (CIL), that is based on re- writing logic and also provides the formal semantics for CAPSL. This form of the language has been specifically designed to be easy to translate to other representations that analysis tools may use internally.

Unfortunately it seems that currently the only way to translate from CAPSL to CIL is by use of a translation program that uses Maude, a system that supports equational and re-writing logic by the CAPSL developers. The translation process does not seem to be well defined outside this environment and the translator itself is described as experimental.

3.1.2 CPAL

Where CAPSL is relatively widely used, CPAL does not seem to have enjoyed the same success. While it provides the same advantage as CAPSL, it is simpler and more concise and does not require an intermediate form that formally specifies its meaning. Its formal semantics are defined by means of pre and post conditions.

3.2 Chosen Specification Language: CAPSL

As both languages meet the desired requirements, the deciding factor is the number of existing tools for working with the language, e.g. parsing, analysis and so on. CAPSL is definitely stronger here and is thus the specification language of choice.

4 IMPLEMENTATION LANGUAGE

4.1 Options

The first requirement for high integrity code generation states that the target or implementation language needs to have its semantics formally defined. Unfortunately formal specifications for programming languages, especially the commonly used procedural and object orientated ones such as C, C++ and Java, are rare. This is largely due to these languages having been designed without the formal specification of semantics in mind, thus making the task of formally defining them retroactively complex and lengthy. There are some exceptions, and also some projects that have looked to define semantics formally for subsets of the Java language. One of the aforementioned exceptions is Scheme, a functional, LISP like language which has formally defined denotational semantics for the whole language.

4.1.1 Java

Java is a widely adopted object orientated language with a C++ like syntax. Although it runs in a managed environment, with resulting advantages such as memory management, safe array types and no pointer arithmetic, it has fairly strong performance and is widely used to implement OLTP servers and the like. Java also has a large collection of well defined APIs for all sorts of tasks, including cryptographic operations. However, formal semantics have only been defined for a subset of the language [2], [9].

This lack of formal semantics for all of the Java language is a major drawback. It makes it difficult to prove that the translation process maps the specification language constructs to correct Java language constructs. This is because in order for the mapping from a specification construct to an implementation construct to be defined, the meaning of both constructs needs to be clear, unambiguous and of course equivalent, something which could be determined given formal semantics for both languages.

One option is to relax an aspect of the 1st requirement and translate from the specification language to a set of well defined Java objects and methods that are assumed to correctly implement the data and actions they represent. In this case it could still be shown that the correct Java code is called for each action, assertion and transition in the protocol, but there would be no guarantee that the actions, assertions and transitions were implemented correctly.

4.1.2 Scheme

Scheme is a LISP like functional language with concise formally defined semantics [6]. As an interpreted functional language it does not have the same performance of commonly used procedural languages like C, C++ and to a lesser extent Java. Also, although most Scheme implementations have libraries for console, file and network IO, cryptographic libraries do not seem to be readily available. It may, however, be possible to work around this by making calls to native code that implements cryptographic algorithms.

4.2 Chosen Implementation Language: Scheme or Java

At this stage it seems as if Scheme is the more appropriate choice. Provided the work around to use native cryptographic libraries mentioned previously is not a significant technical hurdle, the simplicity and the fact that the language is formally defined are strong advantages. That said further investigation into the projects to formally specify semantics for Java is justified, and it may yet prove to be the better choice.

5 TRANSLATION

Considering requirements 2 and 3 for high integrity code the code generation tool is probably best implemented in Prolog or a similar rules based declarative language. This would allow the rules for translation, or mapping, from the specification language constructs to the implementation language constructs to be clearly defined. A language such as Prolog can then apply these defined rules to perform the mapping from specification to implementation.

6 ANALYSIS AND VERIFICATION OF PROTOCOLS

Upon completion of the code generation tool, we would like to use the ability to generate protocol implementations to conduct various forms of analysis.

6.1 Managed Environment for Protocol Runs

We propose the development of an environment to execute controlled and monitored protocol runs. This will facilitate protocol analysis by allowing a protocol run to be traced, monitored for specified events and even manually interacted with by injecting and/or intercepting messages in the message flow. The environment will provide a way for the principles in a protocol run to communicate by

bypassing the network stack, allowing performance measurement to be made that aren't subject to fluctuations of network traffic.

6.2 Protocol Analysis

6.2.1 Comparison of Protocols

We would like to be able to compare the performance of cryptographic protocols that have the same security goals. Being able to automatically generate implementations for protocol will give us the ability to do this kind of performance analysis based on empirical measurements, as opposed to estimating the performance based counting the number of protocol rounds and the type of cryptographic algorithms employed.

6.2.2 Group Protocol Scalability

In a similar vein, testing the scalability of protocols that allow communication between large numbers of principles would also be possible.

6.3 Verification of Protocols Implementations

Given the ability to automatically generate an implementation for a protocol, we would like to extend the code generation tool to automatically generate variations on protocol implementations that test other existing implementations of the same protocol. For example, by generating an implementation for the initiating principle of a protocol that sends certain corrupted or specifically selected invalid message components, one could test that the implementation of the responding principle correctly verifies the messages it receives before preceding to the next round of the protocol.

6.4 Implementing Protocol Level Attacks

As protocol level attacks can be specified in a similar way to protocols themselves, the code generation tool could also be used to demonstrate existing attacks against protocols, as well as experiment with new attacks.

7 CONCLUSION

In this paper we have discussed our intended approach to developing a code generation tool for implementing cryptographic protocols and the possible uses for this in the analysis and verification of protocols. We have also covered some of the issues that will need to be addressed as far as delivering a high level of confidence in the output of the code generation tool. Finally, we have listed some of the analysis and verification we would like to perform once the code generation tool is working.

REFERENCES

[1] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

[2] Stefan Berghofer and Martin Strecker. Extracting a formally verified, fully executable compiler from a proof assistant. In *Proc. 2nd International Workshop on Compiler Optimization Meets Compiler Verification (COCV'2003)*, Electronic Notes in Theoretical Computer Science, 2003. To appear.

[3] S. Brackin, C. Meadows, and J. Millen. CAPSL interface for the NRL protocol analyzer. In *2nd IEEE Workshop on Application-Specific Software Engineering and Technology (ASSET '99)*. IEEE Computer Society, 1999.

[4] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication, from proceedings of the royal society, volume 426, number 1871, 1989. In *William Stallings, Practical Cryptography for Data Internetworks, IEEE Computer Society Press, 1996*. 1996.

[5] G. Denker and J. Millen. CAPSL and CIL language design. Technical Report SRI-CSL-99-02, SRI International Computer Science Laboratory, 1999.

[6] H. Abelson et al. Technical report, March 2001.

[7] Jonathan C. Herzog F. Javier Thayer Fabrega and Joshua D. Guttman. Strand spaces: Why is a security protocol correct? 1998.

[8] Li Gong, Roger Needham, and Raphael Yahalom. Reasoning About Belief in Cryptographic Protocols. In Deborah Cooper and Teresa Lunt, editors, *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society, 1990.

[9] Denis Caromel Isabelle Attali and Marjorie Russo. A formal executable semantics for java.

[10] Shawna McAlearney. Patches issued for multiple radius implementation flaws.

[11] J. Millen and F. Muller. Cryptographic protocol generation from CAPSL. Technical Report SRI-CSL-01-07, SRI International, December 2001.

[12] E. Saul. Facilitating the modelling and automated analysis of cryptographic protocols. Master's thesis, University of Cape Town, July 2001.

[13] Bruce Schneier. Cryptography: The importance of not being different. March 1999.

[14] David Wagner and Bruce Schneier. Analysis of the ssl 3.0 protocol.

[15] Michael W. Whalen and Mats P.E. Heimdahl. On the requirements of high-integrity code generation. In *Proceedings of the Fourth IEEE High Assurance in Systems Engineering Workshop*, November 1999.

[16] Alec F. Yasinsac. *Evaluating Cryptographic Protocols*. PhD thesis, 1996.