Provable Implementations of Security Protocols

Andrew D. Gordon
Microsoft Research

Proving security protocols has been a challenge ever since Needham and Schroeder threw down the gauntlet in their pioneering 1978 paper:

> Protocols such as those developed here are prone to extremely subtle errors that are unlikely to be detected in normal operation. The need for techniques to verify the correctness of such protocols is great, and we encourage those interested in such problems to consider this area. [12]

This may not seem such a grand challenge, at first, as most cryptographic protocols can be written in half a dozen lines and involve even fewer players; the twist is the intruder, who actively interferes with proceedings:

> We assume that an intruder can interpose a computer on all communication paths, and thus can alter or copy parts of messages, replay messages, or emit false material. [12]

Almost all of the many papers addressing the problem of attacks on security protocols fall into two groups. They either develop informal, though empirically based, methods for designing robust protocols, or they develop formal methods for reasoning about abstract models of protocols.

The work on informal methods attempts to discern common patterns in the extensive record of flawed protocols, and to formulate positive advice for avoiding each pattern [1, 2]. For example, Anderson and Needham propose an *Explicitness Principle*, that covers many classic errors:

> Robust security is about explicitness. A cryptographic protocol should make any necessary naming, typing and freshness information explicit in its messages; designers must also be explicit about their starting assumptions and goals, as well as any algorithm properties which could be used in an attack. [2]

Much of the work on formal methods has depended on the powerful idea, introduced by Dolev and Yao [9], of idealizing cryptographic operations such as encryption and decryption as a symbolic algebra. After intense effort on symbolic reasoning, there are now several techniques [8, 7] for automatically proving properties of protocols represented within a symbolic, algebraic model. On the other hand, the formal proofs used by cryptographers tend to rely on probabilistic computational models, which make fewer unwarranted assumptions than symbolic models, but typically lack automation. Justifying symbolic models via computational models (where possible), or simply developing automation for the latter, is a growing research area.

Although either the informal design principles or the automated formal tools would catch most design errors, they clearly only work if applied. As in other areas of software, the trouble is that while practitioners are typically happy for researchers to write formal models of their natural language specifications and to apply design principles, they are reluctant to do so themselves. In practice, specifications tend to be partial and ambiguous, and the implementation code is the closest we get to a formal description of most protocols.

This motivates the subject of my talk: the relatively new enterprise of adapting formal methods for security to work on code instead of abstract models. The goal is to lower the practical cost of security protocol verification by eliminating the need to write a separate formal model.

A step in this direction is to extract formal models from the configuration and policy files that govern security processing in some implementations. For example, we built a policy analyzer [3] for web services policy files; it compiles XML policies found in actual implementations into the TulaFale modelling language [5] for analysis. We found many bugs in user written policies; a simplified version of the research tool now ships in a product [4].

A step in a different direction is to compile formal models into implementation code. Several tools have successfully demonstrated this idea [13, 14]. The difficulty is that this approach involves growing a formal model into a full programming language, building a compiler, educating developers, and so on; overall, it seems cheaper to compile from an existing programming language into a formalism.

Goubault-Larrecq and Parrennes [10] were the first to build a tool to extract a formal model from the actual implementation code (in C) of a cryptographic protocol. They apply a pointer analysis together with an analysis of the messages the intruder can send and receive to construct a

logical model of the program as a set of Horn clauses; these clauses can then be analyzed by other tools.

The main technical content of my talk will be recent work with Bhargavan, Fournet, and Tse [6] on extracting π-calculus models from protocol implementation code. Our software is developed in the functional language F# [15], a dialect of ML. A central idea for structuring the code to be analyzed, is to define typed interfaces for the low-level cryptographic and communication services used by the protocol. These interfaces include operations such as performing an encryption, verifying a signature, or sending a network message. We provide dual *concrete* and *symbolic* library implementations of these interfaces, against which we can build and run our protocol code. The concrete implementation of the cryptographic interface relies on the real cryptographic operations provided by the operating system, while the symbolic implementation is a Dolev-Yao style algebraic idealization. Binaries built from the concrete libraries are for interoperability testing with production implementations, or indeed for production use. Binaries built from the symbolic libraries are for initial symbolic debugging. For verification, we stipulate protocol properties such as authentication and secrecy in terms of correspondences between events. These correspondences are to hold in the presence of an intruder, much as described by Needham and Schroeder, that is able to initiate multiple sessions, to select protocol parameters, and to compromise key material. To prove these properties, we first compile the F# protocol code and symbolic libraries to the π-calculus. The core of the translation is Milner's interpretation of λ-calculus functions as π-calculus processes [11]. Given a π-calculus model of the protocol, we apply Blanchet's ProVerif theorem prover [7] to attempt to verify or refute each correspondence property automatically. Soundness of the method relies on a proof that the compilation from F# to the π-calculus preserves attacks. Hence, if ProVerif shows there is no attack at the π-calculus level, there can be no attack at the F# level. We had to implement several optimizations of Milner's compilation strategy for the translated π-calculus to be efficiently analyzed by ProVerif. For a suite of multi-message protocols, we have both demonstrated interoperability with existing production implementations, and successfully verified expected security properties. We verify all the protocol code written against the low-level cryptographic and communications interfaces. On the other hand, we trust but do not formally verify that the symbolic libraries are correct abstractions of the concrete libraries.

In both these projects on verifying implementation code [10, 6], the verified code has been written within the research group. A challenge remaining for further work, then, is to verify independently written implementations of cryptographic protocols, such as those in common operating systems and platforms.

References

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

[2] R. Anderson and R. Needham. Programming Satan's computer. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, pages 426–440. Springer, 1995.

[3] K. Bhargavan, C. Fournet, and A. D. Gordon. Verifying policy-based security for web services. In *11th ACM Conference on Computer and Communications Security (CCS'04)*, pages 268–277, Oct. 2004.

[4] K. Bhargavan, C. Fournet, and A. D. Gordon. Policy advisor for WSE 3.0. In *Web Service Security: Scenarios, patterns, and implementation guidance for Web Services Enhancements (WSE) 3.0*. Microsoft Press, 2006.

[5] K. Bhargavan, C. Fournet, A. D. Gordon, and R. Pucella. TulaFale: A security tool for web services. In *International Symposium on Formal Methods for Components and Objects (FMCO'03)*, volume 3188 of *LNCS*, pages 197–222. Springer, 2004.

[6] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, 2006. To appear.

[7] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 82–96, 2001.

[8] E. Cohen. TAPS: A first-order verifier for cryptographic protocols. In *13th IEEE Computer Security Foundations Workshop*, pages 144–158. IEEE Computer Society Press, 2000.

[9] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, 1983.

[10] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 363–379. Springer, 2005.

[11] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[12] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[13] A. Perrig, D. Song, and D. Phan. AGVI – automatic generation, verification, and implementation of security protocols. In *13th Conference on Computer Aided Verification (CAV)*, LNCS, pages 241–245. Springer, 2001.

[14] D. Pozza, R. Sisto, and L. Durante. Spi2Java: automatic cryptographic protocol Java code generation from spi calculus. In *18th International Conference on Advanced Information Networking and Applications (AINA 2004)*, volume 1, pages 400–405, 2004.

[15] D. Syme. *F#*, 2005. Project website at `http://research.microsoft.com/fsharp/`.