Automatic Generation of Security Protocols Implementations (Extended Abstract)

Paolo Modesti

School of Computing Science, Newcastle University, UK
paolo.modesti@newcastle.ac.uk

Abstract. The implementation of security protocols is challenging and error-prone. A model-driven development approach allows the automatic generation of an application, from a simpler and abstract model that can be formally verified. Our AnBx compiler is a tool for automatic generation of Java code of security protocols specified in the Alice&Bob notation. In contrast with existing tools, it uses a simpler specification language and computes the consistency checks that agents have to perform on reception of messages. Moreover, the tool applies various optimization strategies to achieve efficiency both at compile and run time.

The implementation of security protocols is challenging and error-prone, as experience has shown that even widely used and heavily tested protocols like TLS and SSH need to be patched every year due to low-level implementation bugs. Moreover, bugs like *"Heartbleed"* (OpenSSL) [1] and "*goto fail*" (Apple TLS implementation) [2] have shown that missing (or untested) checks, hidden deep in the code, may have a severe impact. The critical aspect is that the high-level security properties of a protocol must be hard-coded explicitly, in terms of low-level cryptographic operations and checks of well-formedness.

To counter this problem, we propose a model-driven development approach that allows for automatic generation of an application, from a simpler and abstract model that can be formally verified. Our *AnBx Compiler and Code Generator*[1] [3], is a tool for automatic generation of Java code of security protocols specified in the popular Alice & Bob notation, suitable for agile prototyping. From the design perspective, working on a simplified abstract model has proven to be very effective. It not only allows reasoning about the high-level security property, abstracting from the low-level details of the cryptographic implementation, but it also helps to reduce the problem to a size that can be handled efficiently by automatic verification tools.

However, in order to build robust implementations, it is necessary to define explicitly which (defensive) consistency checks on the received data need to be performed to verify that the protocol is running according to the specification. It is important to recognize that while some checks on reception are trivially derived

[1] Available at `http://www.dais.unive.it/~modesti/anbx/`

from the narrations (verification of a digital signature, comparison of agent's identities), others are more complex and managing them can be a challenging task even for an expert programmer.

In addition to the main contribution of an end-to-end *AnB* to Java compiler, we also present an improved way to compute the checks on reception with respect to a previous solution proposed by Briais and Nestmann [4]. This allows reducing the compilation time (in one case even from days to seconds), preventing space state explosion problems in the optimization phase, and increasing the execution speed. The tool also supports the *AnBx* language [5], an extension of *AnB* to be employed for a purely declarative modelling of distributed protocols.

Architecture of the Compiler

The automatic Java code generation of security protocols comprises several phases. A detailed description of the architecture of the tool, which is developed in Haskell, is given in [6] and can be summarized as follows:

Pre-Processing and Verification *AnBx* → *AnB* → (verification)

The *AnBx* protocol is lexed, parsed and then compiled to *AnB* [7], a format suitable for verification with the external tool OFMC [8], a state of the art model checker which is part of the AVISPA [9] platform. The compiler can also directly read protocols in *AnB*. *AnBx* and its translation to *AnB* have already been described in other works [5,6,10], but we point out that translation from *AnBx* to *AnB* can be parametrized using different channel implementations that realize the security properties, specified at the channel level, by means of different cryptographic operations.

Front-end *AnB* → *ExecNarr* → *Opt-ExecNarr*

After verification, if the protocol is deemed safe, the *AnB* specification can be compiled into an *executable narration* (*ExecNarr*), a set of actions that gives an interpretation of how the protocol participants are expected to execute the protocol. The core of this phase is the automatic generation of the consistency checks derived from the static information of protocol narrations. We build our generation of the checks on the method proposed by Briais and Nestmann [4], extending the language making possible to model a wider and more realistic range of applications, and improving the algorithm in order to improve dramatically the performance for large e-commerce protocols like SET [11] and iKP [12], as detailed in [3].

The *optimized executable narration* (*Opt-ExecNarr*) goes further and applies some optimization techniques, including common subexpression elimination (CSE), which in general are useful to generate efficient code. We identify the set of cryptographic operations, which are computationally expensive, and optimize the code, in order to reduce the overall execution time. To this end, we instantiate variables to store partial results, and reorder assignment instructions with the purpose of minimizing the number of cryptographic operation performed.

Back-end *Opt-ExecNarr* $\rightarrow$ (protocol logic) + (application logic) $\rightarrow$ *Java*

The final result of the compilation is the generation of the Java source code from the *Opt-ExecNarr*. The previous phases are fully language independent and therefore does not require any adaptation whatever target programming language is considered. In addition, in the back-end we have postponed any language dependent decision in order to increase the compiler's portability and simplify re-targeting, as long as other object oriented and procedural programming languages are used. We summarize the main features and components below:

Code generation strategy We make a distinction between the *protocol logic* and the *application logic.* The latter is implemented by means of parametrized application template files written in the target language. The templates are instantiated with the information (the *protocol logic*) derived from the optimized executable narration. We model the *protocol logic* by means of a language independent intermediate format called *Typed-Opt-ExecNarr*, which is in essence a typed representation of the *Opt-ExecNarr*. This is useful to parametrize the translation and to make it easier to emit code in other programming languages.

Type System Building the *Typed-Opt-ExecNarr* requires a type system modeling a typed abstract representation of the security-related portion of a generic procedural language supporting a rich set of abstract cryptographic primitives. The type system infers the type of expressions and variables and insures that the generated code is well-typed. It has the additional benefit of detecting at run time whether the structure of the incoming messages is coherent with the one specified by the narration.

Verification of the implementation The *Typed-Opt-ExecNarr* can be translated in applied-pi, and then verified with Proverif [13]. However, this requires that the *AnB* security goals are analyzed and specific facts, modeling the security properties, are generated along the compilation chain.

Code emission The code emission is performed by instantiating the protocol templates, i.e., the skeleton of the application, with the information derived from the protocol logic. It is worth noting that only at this final stage the language specific features and their API calls are actually bound to the protocol logic. To this end two mappings are required. One between the abstract and the concrete types; another one between the abstract actions and the concrete API calls. It is important to underline that the application templates are generic, i.e., independent from the specific protocol, and can be modified by the user in order to fit his application domain.

Security API The run-time support relies on the cryptographic services offered by the Java Cryptography Architecture (JCA) [14,15]. In order to connect to the JCA, we designed an API for security which wraps, in an abstract way, the JCA interface and implements the custom classes necessary to encode the generated programs in Java. The *AnBxJ* library offers a high degree of generality and customization, since the API does not commit to any specific cryptographic solution (algorithms, libraries, providers). Moreover, the library provides access in an abstract way to the communication primitives used to exchange messages in the standard TCP/IP network environment. The generated code comes along with a configuration file that allows the developer to fully customize the deployment of the application at the cryptographic (keystore location, aliases, cipher schemes, key lengths, etc) and network level (IP addresses, ports, etc) without requiring to regenerate the application.

In summary, the tool allows for a one-click code generation of widely configurable and customizable ready-to-run Java applications from an *AnBx* or *AnB* specification.

References

1. Cassidy, S.: Existential type crisis : Diagnosis of the OpenSSL Heartbleed Bug (2014) http://blog.existentialize.com/diagnosis-of-the-openssl-heartbleed-bug.html.
2. Ducklin, P.: Anatomy of a "goto fail" - Apple's SSL bug explained, plus an unofficial patch for OS X! (2014) http://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/.
3. Modesti, P.: Efficient Java code generation of security protocols specified in AnB/AnBx. In: Security and Trust Management - 10th International Workshop, STM 2014, Proceedings. (2014) 204–208
4. Briais, S., Nestmann, U.: A formal semantics for protocol narrations. Theoretical Computer Science **389** (2007) 484–511
5. Bugliesi, M. and Modesti, P.: AnBx-Security protocols design and verification. In: ARSPA-WITS 2010. (2010) 164–184
6. Paolo Modesti: Efficient Java code generation of security protocols specified in AnB/AnBx. Technical Report CS-TR-1422, Newcastle University (2014)
7. Mödersheim, S.: Algebraic properties in Alice and Bob notation. In: International Conference on Availability, Reliability and Security (ARES 2009). (2009) 433–440
8. Basin, D., Mödersheim, S., Viganò, L.: OFMC: A symbolic model checker for security protocols. Int. Journal of Information Security **4**(3) (2005) 181–208
9. Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuéllar, J., Drielsma, P.H., Héam, P.C., Kouchnarenko, O., Mantovani, J., et al.: The AVISPA tool for the automated validation of internet security protocols and applications. In: Computer Aided Verification, Springer (2005) 281–285
10. Paolo Modesti: Verified Security Protocol Modeling and Implementation with AnBx. PhD thesis, Università Ca' Foscari Venezia (Italy) (2012)
11. Bella, G., Massacci, F., Paulson, L.: Verifying the SET purchase protocols. Journal of Automated Reasoning **36**(1) (2006) 5–37
12. Bellare, M., et al.: Design, implementation, and deployment of the iKP secure electronic payment system. IEEE JSAC **18**(4) (2000) 611–627
13. Blanchet, B.: An efficient cryptographic protocol verifier based on Prolog rules. In: Computer Security Foundations Workshop, IEEE, IEEE Computer Society (2001) 0082–0082
14. Gong, L., Ellison, G., Dageforde, M.: Inside Java 2 Platform Security: Architecture, Api Design, and Implementation. Addison-Wesley (2003)
15. Pistoia, M., Nagaratnam, N., Koved, L., Nadalin, A.: Enterprise Java 2 Security: Building Secure and Robust J2EE Applications. Addison Wesley (2004)