Efficient Java Code Generation of Security Protocols specified in *AnB*/*AnBx*

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.

Keywords: security protocols, code generation, applied formal methods

1 Introduction

The implementation of security protocols is challenging and error-prone, as experience has shown [1] that even widely used and heavily tested protocols like TLS and SSH need to be patched every year due to low-level implementation bugs. 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, in this work we consider a model-driven development approach that allows automatic generation of an application, from a simpler and abstract model that can be formally verified. We present the *AnBx Compiler and Code Generator*[1], a tool for automatic generation of Java code of security protocols specified in the simple Alice&Bob (*AnB*) notation [2], suitable for agile prototyping. Despite being intuitive, protocol narrations are in general semi-formal because they contain a lot of implicit concepts. In particular, it does not say 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 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 [3]. This allows

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

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 [4], an extension of *AnB* to be employed for a purely declarative modelling of distributed protocols.

2 The *AnBx* 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 [5] and can be summarized as follows:

Pre-Processing and Verification *AnBx* $\rightarrow$ *AnB* $\rightarrow$ (verification)
The *AnBx* protocol is lexed, parsed and then compiled to *AnB*, a format suitable for verification with the external tool OFMC [6], a state of the art model checker. The compiler can also read protocols directly in *AnB*. *AnBx* and its translation to *AnB* have already been described in [4,7].

Front-end *AnB* $\rightarrow$ *ExecNarr* $\rightarrow$ *Opt-ExecNarr*
If the protocol is deemed safe by the model checker, the *AnB* specification can be compiled into an *executable narration* (*ExecNarr*), a set of action that gives an interpretation on 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. The *optimized executable narration* (*Opt-ExecNarr*) goes further in this direction and applies some optimization techniques, including common subexpression elimination (CSE), which in general are useful to generate efficient code.

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 do not require any adaptation in case another programming language is used. But even in the back-end we postponed any language dependent decision in order to increase the compiler's portability and simplify the re-targeting.

3 Executable Narrations and Optimization

The computation of the checks on reception is done by extending and refining the ideas proposed in [3]. In short, three kinds of checks are considered in formulas on received messages: *equality* $[E = F]$ on expressions denoting the comparison of two bit-streams, E and F; *well-formedness* $[E]$ denoting the verification of whether the projections and decryption contained in E are likely to succeed; *inversion* $inv(E, F)$ denoting the verification that E and F evaluate to inverse messages. Since consistency checks operate on *(message,expression)* pairs, the representation of the agent's knowledge must be generalized. The underlying idea is that a pair (M, E) denotes that an expression E is equivalent to the message M. For this reason is it necessary to introduce the notion of *knowledge sets*, and two operations on them: *synthesis,* reflecting the closure of knowledge sets using message constructors, and *analysis,* reflecting the exhaustive recursive decomposition of knowledge pairs as enabled by the currently available knowledge.

The compilation of an action $A \to B : M$ checks that M can be synthesized by the agent A, instantiate a new variable x and adds the pair (M, x) to the knowledge of agent B. The consistency formula $\Phi(\mathcal{A}(K'_B))$ of the analysis of the updated knowledge K'_B defines the checks to be performed by B at run-time.

Performance issues A preliminary version of our compiler [7] implemented verbatim the method proposed in [3], extending only the *analysis* and *synthesis* rules in order to support cryptographic functions which were not available in [3], HMAC and Diffie-Hellman key agreements in particular. Unfortunately, especially with some industrial-size protocols, it turned out to be very inefficient. In our experiments [Table 1 (a)], we found it challenging to work with the original specification of the e-commerce protocols SET [8] (we considered the unsigned variant denoted SETv2 in which the customer does not possesses asymmetric keys) and 3KP [9]. A clear symptom of inefficiency was the fact that these protocols required very long time to be compiled (around 1 hr and 55 min for 3KP and almost 6 days for SETv2 on a Windows 7 64-bit machine, CPU Intel Core i7-3770 3.40 GHz, 8 GB RAM, JDK 7.0.45 64-bit, Haskell Platform 2013.0.0.2). In contrast, the revised versions specified in *AnBx* performed better, thanks to the fact that *AnBx* can express security properties at the channel level and this implies, in the concrete implementation, a reduced number of nested encryption layers and MACs, compared to the original specification.

Moreover, we noticed that some of the computed checks were failing anyway. It turned out that the reason of this discrepancy was the different behavior, in the abstract and concrete model, of the cryptographic primitives. In fact, given two identical messages and encryption keys, a non-deterministic encryption scheme returns two different ciphertexts which are indistinguishable by an observer. This nice (in the real world) property was not properly captured by the model.

Optimization To address this issue, the key point is to observe that, if an agent knows the correct decryption key, he will deconstruct the ciphertext using the *analysis* rules. In this case, he will only store the decrypted message in his own knowledge, forgetting the ciphertext. A significant exception is represented by forwarding channels. For example, in SET and iKP the merchant must forward a message originating from the customer but secret for the acquirer, and such expression is stored in the knowledge as an encrypted term. Assuming a non-deterministic cipher scheme, when an agent will need to build a new ciphertext involved in an equality check, the check will fail because the term will differ from the one to be compared with. Therefore, if we prevent the agent from using the *synthesis* rules modelling encryption when computing the checks, we basically avoid computing any check which requires synthesizing new terms using symmetric and asymmetric encryption. It is important to underline that this does not undermine the robustness of the application because we just prune checks failing due to the over approximation of the abstract model.

Protocol	*Compile Time (sec)*			*Exec. Time (sec)*			*Mem. usage* (MB)*		
	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)
2KP (orig)	97.75	2.80	2.25	1.08	0.97	0.81	3.46	4.80	8.24
2KP (AnBx)	1.03	0.17	0.13	1.01	1.17	0.95	0.91	1.06	0.88
3KP (orig)	6,945.43	13.23	13.20	100.85	1.34	0.83	411.52	16.08	33.64
3KP (AnBx)	8.88	0.16	2.25	1.26	1.44	0.91	1.05	1.03	1.04
SETv2 (orig)	513,827.20	2.89	3.04	4.05	3.94	1.11	262.07	5.93	7.91
SETv2 (AnBx)	0.84	0.06	0.05	1.08	1.25	0.89	0.74	0.80	0.83
H530	1.89	1.75	2.70	1.96	1.88	1.79	10.14	9.12	5.31
Google SSOv2	1.33	0.03	0.02	1.07	0.95	0.90	0.68	0.71	0.75
Kerberos PKinit	0.37	0.36	0.36	1.32	1.12	0.94	1.35	1.53	1.88

Table 1. Exp. results: (a) as in [3] w/o opt (b) opt (c) opt+cse (*at compile time)

From the practical point of view [Table 1 (b)], we were able to reduce the compile time for 3KP, from 1 hr 55 min to just 13 sec, decreasing the peak memory usage (measured by the Profiler of the Haskell Compiler 7.6.3) from 411 MB to just 16 MB. The execution time was also cut from 100 sec to just 1.34 sec. This large difference is explained by the fact that the standard algorithm [3] computes more than 10,000 checks (but most of them are failing), while our version generates about 50 checks. For the protocol SETv2, the good news is that it can now be compiled in 3 sec while before it required almost 6 days. Peak memory usage was also decreased from 262 MB to just 6 MB. The execution time diminished just from 4.05 sec to 3.94 sec. This is interesting because in this case the check set is not heavily pruned as in 3KP. Indeed our changes allows detecting the checks more efficiently than before.

Common Subexpression Elimination (CSE) We identify the set of cryptographic operations, which in general are computationally expensive, and optimize the code to reduce the overall execution time, introducing variables storing partial results, and making a reordering with the purpose of minimizing the number of cryptographic operation performed. In terms of performance gain [Table 1 (c)], CSE and reordering allowed us to cut further the execution time by 72% for SETv2, 38% for 3KP, but less than 5% for H530.

4 Related Work and Conclusions

With respect to the tools proposed in the past, our compiler generates Java code which includes the checks on reception. We think this is very important for building defensive implementations of security protocols and it has a practical impact. However, this makes difficult to compare the compile time performance with other tools because in [10,11,12] the checks must be written manually. In

[13] a notion of "receivability" is used, which only models the ability to decrypt the received messages but does not compute other checks. In contrast to the tools that require process calculi as input language [10,11,12], we use a more intuitive language *AnB*, making our tool suitable for a larger audience of developers. In addition, our abstract specification is the most compact. Using SPI requires long specification files [11] and type annotations, [13] requires type annotations as well. Instead, we use a simple naming convention to make the protocol specification extremely succinct and the tool delegates the duty to generate well-typed code to the type system. Future work could take several directions. It would be important to make a formal proof of the soundness of the translation process along with extending the tool to generate interoperable code.

Acknowledgments Part of this work was carried out while the author was a Ph.D. candidate at Università Ca' Foscari Venezia, under the valuable supervision of Prof. Michele Bugliesi. This work was partially supported by the EU FP7 Project n. 318424, "FutureID: shaping the Future of Electronic Identity". The author wishes to thank Thomas Groß for his helpful discussions and comments.

References

1. Avalle, M., Pironti, A., Sisto, R.: Formal verification of security protocol implementations: a survey. Formal Aspects of Computing **26**(1) (2014) 99–123
2. Mödersheim, S.: Algebraic properties in Alice and Bob notation. In: International Conference on Availability, Reliability and Security (ARES 2009). (2009) 433–440
3. Briais, S., Nestmann, U.: A formal semantics for protocol narrations. Theoretical Computer Science **389** (2007) 484–511
4. Bugliesi, M. and Modesti, P.: AnBx-Security protocols design and verification. In: ARSPA-WITS 2010. (2010) 164–184
5. Paolo Modesti: Efficient Java code generation of security protocols specified in AnB/AnBx. Technical Report CS-TR-1422, Newcastle University (2014)
6. Basin, D., Mödersheim, S., Viganò, L.: OFMC: A symbolic model checker for security protocols. Int. Journal of Information Security **4**(3) (2005) 181–208
7. Paolo Modesti: Verified Security Protocol Modeling and Implementation with AnBx. PhD thesis, Università Ca' Foscari Venezia (Italy) (2012)
8. Bella, G., Massacci, F., Paulson, L.: Verifying the SET purchase protocols. Journal of Automated Reasoning **36**(1) (2006) 5–37
9. Bellare, M., et al.: Design, implementation, and deployment of the iKP secure electronic payment system. IEEE JSAC **18**(4) (2000) 611–627
10. Pozza, D., Sisto, R., Durante, L.: Spi2Java: Automatic cryptographic protocol Java code generation from spi calculus. In: Proceedings of the 18th AINA, IEEE (2004)
11. Backes, M., Busenius, A., Hriţcu, C.: On the development and formalization of an extensible code generator for real life security protocols. In: NASA Formal Methods. Springer (2012) 371–387
12. Tobler, B., Hutchison, A.: Generating network security protocol implementations from formal specifications. Cert. and Security in Inter-Org. E-Service (2005) 33–54
13. Millen, J., Muller, F.: Cryptographic protocol generation from CAPSL. Technical Report SRI-CSL-01-07, SRI International (2001)