Issues on Designing a Cryptographic Compiler

Stefan Lucks, Nico Schmoigl, Emin İslam Tatlı
Department of Computer Science
University of Mannheim

Abstract: Flawed implementations of security protocols is a major source of real world security problems. Typically, security protocols are specified in some "high-level" way and may even be formally proven secure. Implementing them in practical (and comparatively low-level) source code has turned out to be error-prone. This paper introduces an experimental language for high-level protocol specifications and describes a tool to automatically compile source code from these specifications.

C. Wolf, S. Lucks, P.-W. Yau (Eds.): WEWoRC 2005, LNI P-74, pp. 109–122, 2005.

1 Motivation: Implementing Security Protocols

The design of cryptographic security protocols is tricky and error-prone, and many such protocols have been broken by cryptanalysts. To defend against surprises, it has become good practice to formally prove the security of a protocol (often under some unproven but well-studied and reasonable assumptions).

In practice, however, the designers of cryptographic security protocols should fear their friend, the implementor of their cryptosystem, as much as their foe, the cryptanalyst. A huge number of security flaws in applications is due to implementation flaws rather than successful attacks against the specified scheme. Implementors often miss some subtle issues of the protocol, perhaps when committing last-minute changes or quickly fixing bugs discovered before. Sometimes, the implementor may even deliberately deviate from the specification. Even if the implemented protocol still passes all functional tests and runs perfectly well in a non-adversarial environment, the difference between specification and implementation may still be exploitable by adversaries.

This paper wants to give a snapshot at our research in progress. At the time of writing, our specification language for security protocols is still in flux, and the compiler is still a prototype, though already able to compile some protocol specifications. The current prototype is still restricted to Java as the only target language.

This paper is organized as follows. Section 2 describes goals and requirements for the cryptographic compiler. Section 3 discusses available languages for specifying security protocols and tools to ensure the correct implementations. The compiler architecture is explained in Section 4, Section 5 provides two example protocols. In Section 6, we discuss our results together with further goals and conclusions.

2 The Idea of the Cryptographic Compiler

As pointed out in the motivation, bridging the semantic gap from an abstract specification of a protocol to a concrete implementation in some programming language is dangerous and error-prone. Thus, we would like to have a tool which takes some more or less abstract specification of a security protocol as input and produces an implementation of the protocol in some programming language. In a nutshell, this is what our cryptographic compiler will do. Of course, finding a suitable input language is a precondition for implementing the compiler. In the following, we list the major requirements for compiler and language in more detail:

High Abstraction Level. The compiler's input language should support an appropriate abstraction level for cryptographers specifying cryptographic protocols. In other words, the semantic gap between the specification of the protocol in some verification language (if it is proven secure by the means of automated reasoning) or in the context of some theorem (if proven secure manually) and its specification for source code generation must be minimized.

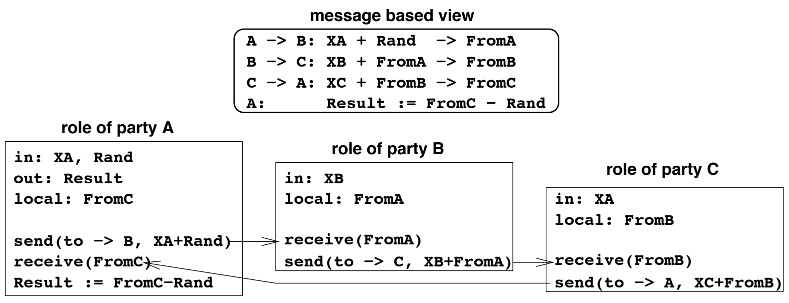
Flexibility and protocol constraints. Language and Compiler shall support flexible protocols. E.g., when specifying protocols employing a cryptographic hash function, the designers shall be allowed to leave the concrete choice of the hash function open. On the other hand, the designers shall be able to impose constraints, e.g., on the minimum length of a hash value.

Avoid Dependence on a particular Programming Language. The compiler shall be able to generate source code in different programming languages. Additionally, only one protocol definition should be created by the designer, such that the definition can both be verified and transformed into source codewithout having to adjust the input file(s). More importantly, extending the compiler (or rather its "back-end", see below) to support additional languages shall be easy.

Message-based $\leftrightarrow$ Role-Based. Typically, protocols are specified in a message-based way, like, e.g.: *First, a message is sent from A to B, second, a message is sent from B to C, third, another message is sent from C to A, . . .* On the other hand, protocol implementations are typically role-based with different program fragments (threads, methods, . . .) for each party, e.g.: *A sends a message to B and then waits for a message from C . . .* See also Figure 1.

We believe that the above requirements for compiler and protocol specification language are very reasonable.

As the logical consequence of the high-level approach we are taking, our compiler is not meant for writing low-level specifications:



```
A -> B: XA + Rand  -> FromA
B -> C: XB + FromA -> FromB
C -> A: XC + FromB -> FromC
A:      Result := FromC - Rand
```

```
in: XA, Rand
out: Result
local: FromC

send(to -> B, XA+Rand)
receive(FromC)
Result := FromC-Rand
```

```
in: XB
local: FromA

receive(FromA)
send(to -> C, XB+FromA)
```

```
in: XA
local: FromB

receive(FromB)
send(to -> A, XC+FromB)
```

Figure 1: Message- and role-based view at a joint three-party computation of `Result:=XA+XB+XC`.

No implementation of cryptographic building blocks: We assume that there is a cryptographic library which implements the required building blocks (e.g. block ciphers, public key encryption atomics, ...).

No low-level specification of data in transmission: Objects sent between two parties must know how to serialize and unserialize themselves. This implies that if different implementations (for example different libraries) are used, programmers must (un)serialize in a compatible way (for example take care of the endianess). A similar issue arises when implementing standardized protocols such as TLS [Die]. This has to be considered by a layer below our protocol language.

2.1 Example Application: A Mobile Business Project

Developing a compiler never should end in itself, so we aim to use it for the implementation of secure applications in the context of a mobile business (m-business) project at the University of Mannheim [mbu]. The project's goal is to develop a generic framework to enable context-aware mobile services. Handling security issues diligently is of vital importance for the project's success.

The need to integrate the compiler's output (i.e., the security logic) into the remaining parts of the applications (the application logic),[1] has been a major source of inspiration for our design decisions. In fact, some of our major requirements mentioned above are inspired by necessities for the m-business project. For example, the requirement for flexibility corresponds to dynamically supporting different security policies in the m-business project.

[1]Defining the interface between application logic and security logic can be quite tricky. Even when the application just needs to send a signed or encrypted message to some communication partner, the application context determines which keys to use for signing/encrypting.

3 Previous Work: Specification Languages and Issues

At the beginning of this research, we considered using one of the existing languages for specifying security protocols. In general, these languages have been developed with automatic protocol analysis and protocol verification in mind, rather than defined for automatically generating implementations. Specifically, we considered CAPSL [Mil97], HLPSL [CCC$^+$04], CASPER [Low98] and LAEVA [JM01].

In the case of CAPSL, Millen and Muller (MM) even documented their approach to implementing a Java-targeted cryptographic compiler in [MM01]. At a first look, the MM compiler already seems to achieve what we actually want to implement. However, there are some severe issues.

- The implementation of the compiler only generates Java output. We need to support more than one target language.
- The MM compiler handles the connection between security logic and application logic by some assumed "environment server". While this may be a reasonable choice from a verification point of view, it introduces an additional layer of complexity for the implementor.[2]
- CAPSL is not quite expressive enough for us. E.g., it does not support loops.
- While CAPSL supports if-then-else, the MM compiler does not.

Nevertheless, the MM compiler was a source of inspiration for us. Understanding both its achievements and its limitations turned out to be quite useful for making our own design decisions.

Apart from CAPSL, we considered the specification languages mentioned above. All these languages have been designed and successfully used for the formal analysis of security protocols, but for source code generation, all of them would have to be extended by additional language constructs or enriched by annotations.

Such an attempt of extending a given specification language – namely CASPER – for source code generation is documented by Didelot in [Did, Section 3.2]. At first glance, this approach seems to implement an easy way of producing source code from a given protocol specification written for analysis. Currently, however, Didelot does not provide any tool or automated way to retrieve a CASPER style input file from a file of his extended language. If a protocol specification in CASPER is changed, the annotations have to be updated manually, which hinders flexibility.

Our main issues with existing specification languages are the following.

[2]Instead of an environment, our specification language introduces parameters. One has to explicitly specify which communication party knows which parameter. The output of our compiler will generate a "module" (or a "class", in object-oriented terminology) for each communication party participating in the protocol. Although the implementation of parameters is considered a design decision on the side of the connector's implementor, it is encouraged to either implement it as "property" in terms of an object-oriented target language or as global constant in terms of a functional target language.

- In almost all languages, the order of passing the variables in a message is not defined precisely. For automatic analysis, one can simply assume sender and receiver to use the same transmission order – but an implementation must take care of this.
- No language defines a way to explain where nonces, block ciphers or tag values are retrieved. Especially, assuming to have a "perfect block cipher" might be enough for protocol verification, but this aspect is vital for a real implementation.
- In most protocol specification languages, one defines the properties that have to hold after a successful protocol run. This is required for protocol analysis. Few languages, however, *explicitly* describe the interface between the protocol and its user (the application running the protocol). For practical implementations, interface specifications are a major issue and unclear interface specifications are known to be error-prone. The latter may easily lead to security leaks.
- Protocol specifications usually concentrate on the "normal protocol flow" and often lack explicit failure handling. We will discuss this in Section 6.

To summarize, a specification language written with automatic analysis in mind will lack features needed for source code generation. Extending an existing language by annotations or additional language constructs is possible, but appears to be tricky. After all, the specification should still be reasonably well human-readable.

Based on the aim of optimizing current high-level compilers to incorporate more cryptographic related patterns, M. Barosa, R. Noad, D. Page and N.P. Smart have written the CAO paper [BNPS05]. Instead of starting from the point of protocol verification, they focus on enhancing current functional programming languages in such a way that common cryptographic primitives become accessible in high-level programming languages more easily. Moreover, they discuss optimization approaches even in the light of side-channel attacks. This is a very interesting approach, aiming at an abstraction level for cryptographic building blocks. This is orthogonal to our approach, as we are seeking for a (high-level) language defining the flow of information between parties. It seems worth future research to use CAO for specifying the building blocks that will be used in our language.

4 The Compiler Architecture

Instead of modifying and extending an existing protocol specification language to suit our needs, we decided to define our own. The major task therefore was to find a language definition which is both suitable for the analysis of cryptographic protocol and immediately translating them into plain, non-interpreted source code. The current paper describes a snapshot of an ongoing research project. As mentioned in the motivation, not all details of the specification language's syntax and semantics are fixed yet. In Section 5, we illustrate some basic principles behind our language design by examples.

As we demanded in Section 2, the abstract protocol definition must be target language independent. Thus, if we change the target language, we have (of course) to modify the compiler, but we do not want to additionally have to change any protocol definition. It seems natural to separate the compiler into two parts (see [AA99, pp. 24–25]):

- The *front end* generates tokens from the abstract definition, parses them into a parse tree and stores this information into an external file.
- The *back end* takes the parse tree and forms the language dependent output. We split up the back end into the "Connector" and the "Target Translator" (often referred to as "Translator"). When choosing another target language, only the Translator needs to be adopted.

This is illustrated in Figure 2.

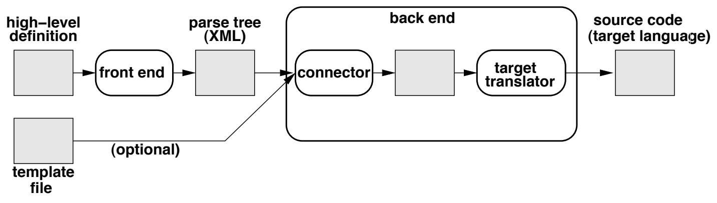


Figure 2: The information flow of our cryptographic compiler

4.1 The Front End

The front end takes the abstract protocol definition as input and generates a parse tree, in the syntax of the well-known Extensible Markup Language (XML). The benefits of this approach are:

- XML defines a language family of normally quite human-readable structures, thereby simplifying debugging.
- The structure of XML documents is already tree-like and therefore copes well with a parse tree.
- Combining the two arguments above, it is possible to use a DTD to only allow well-formed and valid XML files, which allows to a detect a lot of possible syntax errors very easily.
- Simple transformations and analysis can be done directly with XSLT [W3C99].

Almost all of the handling of user errors will be done by the front end – the back end is largely relieved from error handling.

4.2 The Back End

The back end takes the the parse tree as input and checks the consistency of its XML structure[3] and creates the desired output source code for the implementation. In the last stage, the validity of the parameters and variables is checked implicitly. Accordingly, the back end can be divided into two separate parts:

Connector: After loading a representation of the parse tree into memory, the connector enriches this data structure by the information from the template file and some "Cryptographic Object Information", which is part of the compiler's distribution.

Target Translator: The target-language specific part of the compiler is the Target Translator. For fast switches between different target languages, it can be changed every time the back end is started.

This approach of just changing the Translator allows code re-usage to a huge extent. It permits to port existing cryptographic systems simply by implementing a suitable Target Translator. By changing the parameter to the Back End's invocation, the full implementation of the system in the desired language can then be generated.

4.3 The Template File

The abstract protocol specification does not need to specify concrete instances of block ciphers, hash functions, asymmetric schemes and similar cryptographic building blocks. However, the compiler needs to replace, say, some abstract "block cipher" by a concrete one (AES, triple-DES, . . .). The template file defines these matchings. As there is only one template file for each compiler run at the same time, each concrete assignment is unique[4]

The usage of the template file provides a great deal of flexibility – if one distrusts the security of the SHA-1 hash function, one can easily replace it by RIPEMD-160, simply by changing the template file and recompiling the protocol specifications: there is no need to change the specifications themselves.

Hence, the template mechanism allows the protocol designer to delegate the choice of cryptographic primitives to "security architects", who are then responsible for selecting *good* primitives. Additional assertions can provide some guidelines for the architects – and even prevent them from generating "apparently insecure" protocol instantiations. E.g., a 64-bit block cipher, such as triple-DES, may be considered secure in the context of one protocol, but, simply due to its small block size, harmful for another protocol. Accordingly, the designer can assert, e.g., "the block size of block cipher E is at least 128 bit". Violating such a constraint results in an error message without generating any source code.

[3]Mainly a version check to the XML information is done and the XML structures is validated to ensure that no manual modification has broken the internal structure.

[4]Please note that it may be possible to compile the same protocol specification with different templates, thus resulting in distinct protocol implementations.

5 Examples

We demonstrate the basic idea of our specification language and the concept of the compiler by presenting the example of the encrypt-then-authenticate paradigm for a secure secret-key channel.

Secret-key cryptography provides techniques to ensure the confidentiality of a message ("encryption") and techniques for authenticity. Often, however, one needs both confidentiality and authenticity (so-called "authenticated encryption"). Thus, it seems natural to combine encryption and authentication. Surprisingly, the order in which this is done matters: Assume we are given a secure encryption and a secure authentication scheme and two independent secret keys: one for authentication and the other for encryption. If the plaintext is first authenticated and then encrypted, the resulting authenticated encryption scheme may be insecure – in spite of encryption and authentication being secure when used on their own. On the other hand, it is provably secure to first encrypt and then authenticate the ciphertext [Kra01].

Since this protocol is extremely simple, we use it as a first example. The protocol is described in Figure 3, using our abstract language. Note that both parties share *two* secret keys, namely one for encryption and a second one for authentication.

As the source code already shows many of our newly introduced elements and structures, we give a short outline of the purpose they fulfill[5]:

Line 20: We introduced the concept of parameters. Their values need to be known prior to the protocol invocation and are considered constant during the execution. Communication between different instances in general assumes matching parameters.

By specifying the parameters in the corresponding block, the designer is able to define which of the parties is aware of a given parameter. Please note that in the case of parameters, it is very common that in-parameters are defined for several involved parties. The usage of "shared parameters" (cf. Line 15) for this purpose is encourage, if the *values* will be logically the same. Line 22 should therefore be read in the sense that the declaration of the (shared) parameter is "imported" into the scope of party "A". Please note that the declaration of shared parameters is vital for the implementation of translators which generate their output in languages for protocol analysis.

Line 21: There are two types of parameters: in-parameters and out-parameters. As their names suggest, the prefix corresponds to the data flow direction: in-parameters are values which the protocol must know on startup of the protocol, whereas out-parameters are the values the protocol provides after successful execution.

Considering the entire approach of parameter declaration, it solves two major issues at once: It allows an easy way of specifying the interface of the protocol both for the inbound and outbound direction as well as a clear and compact description for parameters and their mapping into the party's scopes.

[5]the order of source code lines is changed intentionally

```
system EncryptThenAuthenticate {
    template {
        S : SymmetricCipherKey;
        M : MacKey;
        T : Tag;

        assert(S, key_size >= 80);

        assert(M, key_size >= 80);
        assert(M, tag_size >= 32);

        assert(T, size >= 32);
    }

    param shared {
        kEnc : S;
        KAut : M;
    }

    param A {
        in B : Party;
        shared kEnc;
        shared KAut;

        in send_message : BitStream;
    }

    param B  {
        in A : Party;
        shared kEnc;
        shared KAut;

        out rcv_msg : BitStream;
    }

    var A {
        cphrtxt : BitStream;
    }

    var B {
        c_txt : BitStream;
        aut_tag   : T;
    }

    A : ciphertext :=
        kEnc.encrypt( send_message );
    A -> B : cphrtxt -> c_txt;
    A -> B : KAut.MAC_generate(cphrtxt)
        -> aut_tag;
    B : if KAut.MAC_verify(c_txt,
        aut_tag) = 0 then
            fail(1);
        endif
    B : rcv_msg :=
        kEnc.decrypt(c_txt);
}
```

Figure 3: The abstract definition file of the encrypt-then-authenticate paradigm for authenticated encryption

Line 36: While the parameters specify the interface of the protocol, variables store internal values which are not bound to the interface. The most fitting analogon of this approach is the widely used "local variable" in modern programming languages. They should be used mainly for storing internal states during protocol execution.

Line 2: Both parameters and variables may have a special type: we call them "templated types". As already said in Section 4.3, one can think of them as aliases or copies of the original type, however, the designer may put constraints on their usage. Such a constraint can be found in Line 7 where the templated type "S" gets the requirement, that its property called "key_size" must be greater than or equal to 80 (in this case: bits). Please note that using an "assert" implies that this constraint must be checkable at compile-time. The check is executed against the information obtained from the template file. A single non-fitting constraint causes an error message during the execution of the back end, which will prevent the creation of the output.

The introduction of constraints on types therefore vitally contributes to enforcing the cryptographic demands of protocol design during the time of source code generation. This even allows the protocol designer to catch possible risks by flawed template specifications.

Line 46: As stated in our requirements (see Section 2), it is important to use a compact notation with a message-based view of the protocol during the design phase. There are two types of messages: internal and external messages.

Line 46, 51 and 55: For later use, a party may have to execute some command. E.g., in Line 46, a ciphertext is computed, which later is sent to another party. We view such commands as "internal" messages of a party to itself. Normally, internal messages are only required by back ends which compile into a non-verification-only language.

Line 47 and 49: These lines define (external) messages that are used for describing the communication between the parties of the protocol. The notation is roughly similar to the one used in protocol analysis languages such as CAPSL or HLSPL.[6]

Line 51: As we discussed earlier in the context of the MM compiler, our compiler supports the if-then(-else) construction. It is essential to be able to check the validity of some input if the back end creates the source code for a real protocol implementation.

Line 52: The issue of having a "fail" keyword is closely related to if-then(-else) statements. On detecting a non-fitting input value, it is important to stop protocol execution and return control to the protocol or application logic. It is the task of the layer above the protocol to decide what to do on a certain type of protocol failure. Therefore, the "fail" statement takes a failure code (currently just an integer) as argument.

[6]Note that we deprecate the popular *curly brackets*-notation "$\{\text{message}\}_K$". The meaning of "$\{\text{message}\}_K$" is not always obvious and may have to be derived from the protocol goals. If, e.g., K is a symmetric key, "$\{\text{message}\}_K$" could either mean "encrypt message under K", or "generate a message authentication code for message under K", or perhaps both, i.e., the authenticated encryption of the message under K ... The need to decide between such options would complicate our compiler a lot. The "object.message" notation we use instead has been inspired by common object-oriented programming languages.

The application logic then reacts on that value. We elaborate a bit more on this issue in Section 6.

Based on the specification of the protocol above, the front end parses it and generates a protocol representation in XML, which is then read by the back end. It is highly recommended not to alter this XML file manually, as most checking is already done during parsing the protocol specification by the front end. The front end already generates sectionized data structures like a symbol table and a type table. Combined with some data in the template file which is shown in Figure 4, the back end creates the desired source code. Figure 5 gives an outline of how the back end works when using the Java Target Translator.

```
S = AES
S.key_size = 128

M = OMAC
M.key_size = 128
M.tag_size = 64

T = StandardTag
T.size = 64
```

Figure 4: The contents of the template file, used by the back end

6 Discussion

As discussed in Section 3, there exist quite a few similar languages. But all of them have been developed to support automatic analysis and protocol verification. On the other hand, our language has been developed to be automatically compiled into a practical programming language, and the design of our language reflects this.

However, we are not at all willing to sacrify the option of automatic protocol analysis! It is possible to implement analysis tools for our language[7]. But doing so would duplicate the effort done for other specification languages. Our compiler architecture provides another option: We argued that it is rather simple to adapt the compiler back end to another target language. Instead of a practical programming language, the target language can be another specification language – which then allows us to use existing analysis tools for other specification languages. As part of the current research, we are writing a compiler back end for CAPSL.

Thus, we will be able to specify protocols, verify their correctness by automatic analysis techniques and automatically generate implementations. If an abstract protocol is proven secure, the implementation might still be insecure – our compiler could be buggy. We

[7]This implies a syntactic element to specify the security goals of the protocol, which is not shown in our examples because the information is not needed for code generation.

```
// import statements omitted for better readability

public class EncryptThenAuthenticate_B
    extends CryptoSystem {
    // empty constructor omitted

    private OMAC KAut;
    public void setKAut(OMAC ref) {
        this.KAut = ref;
    }

    private AES kEnc;
    // Setter method for kEnc
    // omitted for better readability

    private BitStream rcv_msg;
    // Setter method for rcv_msg also omitted

    private Party A;
    // Setter method for A also omitted

    public void doProtocol()
      throws java.beans.PropertyVetoException,
      AssertionFailedException,
      UnmarshallingException,
      FailException {
        StandardTag aut_tag;
        BitStream c_txt;
        this.setState(1);
        // nothing to do for this Party

        this.setState(2);
        BitStream bs_2 = this.getA().receive();
        c_txt = bs_2;

        this.setState(3);
        BitStream bs_3 = this.getA().receive();
        aut_tag = StandardTag.
            getFromBitStream(bs_3);

        if (!bs_3.isEmpty()) {
            throw new UnmarshallingException
                ("buffer overflow attack " +
                 "on BitStream bs_3");
        }

        this.setState(4);
        if (KAut.MAC_verify(c_txt, aut_tag)
            == 0) {
            throw new FailException(1);
        }

        this.setState(5);
        rcv_msg = kEnc.decrypt(c_txt);
    }
}
```

Figure 5: The output for Party B of the Compiler by using the Java Target Translator

take great care to ensure correct compilation but proving correctness is beyond our current capabilities and can only be the goal for a long-time research project. In the meantime, we recommend to review the source code generated by the compiler for security problems – as the code generated by a human programmer should be. But note that the compiler cannot deliberately deviate from the specifications, cannot "misunderstand" protocol descriptions, and does never behave hectically, not even under the pressure of close deadlines.

The abstract level of most protocol specification languages conveniently saves the protocol designer from dealing with failures during protocol execution (the connection is interrupted or timed out, a value is out of its expected range, a message authentication or signature verification fails, ...). This is a good thing – but not for deriving source code from the specifications. The implementation of a security protocol must report possible failures back to the caller (i.e., to the application logic employing the security protocol). This is part of the interface between the protocol and its caller, and we found it prudent to specify this interface *explicitly* for any implementation. For this, our specification language provides a "`fail`" command, not known to most other specification languages. However, using "`fail`" inaccurately, one could introduce a vulnerability to a variant of the Bleichenbacher attack [Ble98].

7 Conclusion

Working in an m-business project, we have to both verify and implement the security protocols in our framework. Implementing security protocols manually is a very slow and error-prone process, and therefore automatic code generation tools is something to aim for. Unfortunately, existing verification tools either do not offer the possibility of automatic code generation, or their code generation mechanism does not satisfy our requirements. We therefore have aimed to build an automatic code generation framework which can enable both fast development and error-free codes.

In this paper, we explained the proposed cryptographic compiler that can generate source code in different programming languages from our experimental high-level specification language for security protocols. In addition, with suitable connectors to verification tools, our compiler can bridge the gap between source code generating languages and automatic protocol verification.

In our framework, one can so far specify security protocols and generate source code in Java. Generating source code for different languages like C++, Ada and implementing connectors to verification tools are within our further goals.

References

[AA99] J. Ullmann A. Aho, R. Sethi. Compilerbau Teil 1 (Compiler's principles, Techniques and Tools), 1999.

[Ble98] Daniel Bleichenbacher. Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS#1. In *Advances in Cryptology – CRYPTO' 98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Lucent Technologies, 1998.

[BNPS05] M. Barbosa, R. Noad, D. Page, and N.P. Smart. First Steps Toward a Cryptography-Aware Language and Compiler. Cryptology ePrint Archive, Report 2005/160, 2005.

[CCC$^+$04] Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, J. Mantovani, S. Mödersheim, and L. Vigneron. A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols. In *Proceedings of Workshop on Specification and Automated Processing of Security Requirements (SAPS 2004)*, 2004.

[Did] Xavier Didelot. COSP-J: A Compiler for Security Protocols.
URL: http://web.comlab.ox.ac.uk/oucl/work/gavin.lowe/Security/Casper/COSPJ/secu.pdf.

[Die] T. Dierks. The TLS Protocol Version 1.0 (RFC 2246).
URL: http://www.faqs.org/rfcs/rfc2246.html.

[JM01] Florent Jacquemard and Daniel Le Métayer. Language de spécification de protocoles cryptographiques de EVA : syntaxe concrète.
URL: http://www-eva.imag.fr/bib/EVA-TR1.pdf, November 2001.

[Kra01] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: how secure is SSL?), 2001.

[Low98] Gavin Lowe. Casper: A Compiler for the Analysis of Security Protocols.
URL: http://web.comlab.ox.ac.uk/oucl/work/gavin.lowe/Security/Casper/casper.ps, July 1998.

[mbu] The Mobile Business Research Group.
URL: http://www.m-business.uni-mannheim.de.

[Mil97] J. Millen. CAPSL: Common Authentication Protocol Specification Language.
URL: http://www.csl.sri.com/users/millen/capsl, 1997.

[MM01] Jonathan Millen and Frederic Muller. Cryptographic Protocol Generation from CAPSL. *SRI Technical Report, SRI-CSL-01-07*, December 2001.

[W3C99] W3C. XSL Transformations (XSLT) Version 1.0.
URL: http://www.w3.org/TR/xslt, November 1999.