Compiling CAO: from Cryptographic Specifications to C Implementations

Manuel Barbosa, David Castro, and Paulo F. Silva

HASLab/INESC TEC — Universidade do Minho, Portugal
{mbb, dcastro, paufil}@di.uminho.pt

Abstract. We present a compiler for CAO, an imperative DSL for the cryptographic domain. The tool takes high-level cryptographic algorithm specifications and translates them into C implementations through a series of security-aware transformations and optimizations. The compiler back-end is highly configurable, allowing the targeting of very disparate platforms in terms of memory requirements and computing power.

1 Introduction

The development of cryptographic software poses a set of challenges that differ from general-purpose software. Producing cryptographic code requires a set of skills related to mathematics, electrical engineering and computer science. Moreover, performance is usually critical and aggressive optimizations must be performed without altering the security semantics. It is common to find cryptographic software directly implemented in assembly because this permits a more efficient implementation, whilst ensuring that low-level security policies are satisfied. Hence, the development of cryptographic software is often an error-prone and time consuming task that only experts can be trusted to carry out.

The CAO language [1] aims to change this. It is a domain specific language (DSL) tailored for the implementation of cryptographic software. In this paper we present a tool for compiling CAO programs into C libraries, i.e., cryptographic components that can then be integrated into more complex software projects. Although at the high-level it appears similar to that of a standard compiler, the architecture of the CAO compiler has been tailored to cater for the widely different scenarios for which cryptographic code may need to be produced, with two main design goals: i. to create a compilation tool that is flexible and configurable enough to permit targeting a wide range or computing platforms, from powerful servers to embedded microcontrollers; and ii. to incorporate, whenever possible, domain-specific transformations and optimizations early on in the compilation process, avoiding platform-specific variants of these transformation stages. One

This work was made in the framework of the BEST CASE project ("NORTE-07-0124-FEDER-000056") financed by the North Portugal Regional Operational Programme (ON.2 – O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), and by national funds, through the Foundation for Science and Technology (FCT).

example of this is the generation of indistinguishable operations needed in the deployment of countermeasures against side-channel attacks.

CAO Language. CAO is an imperative language that supports high-level cryptographic concepts as first-class features, allowing the programmer to focus on implementation aspects that are critical for security and efficiency. In particular, CAO has call-by-value semantics and does not provide any language construct to dynamically allocate memory nor input/output support, as it is targeted at implementing the core components of cryptographic libraries. The native types and operators in the language are highly expressive. The CAO native types are: booleans, arbitrary precision integers, machine integers, signed/unsigned bit strings of a given length, rings or fields defined by an integer, extension fields defined by a type and a polynomial, vectors of elements of a type and a given length and matrices of elements of a type and a given size. There is a number of built-in operators and expressions which deal with values of these types. The operators include: arithmetic binary/unary operators, operators for comparing elements, bitwise operators for bit-strings and shift, rotate and concatenation operations on bit-strings. CAO is strongly typed, and the type system provides a powerful mechanism for implementing templates of cryptographic programs by using *symbolic constants* and a limited form of dependent types. A detailed description of (an earlier version of) the CAO language, type checking rules and a proof of their soundness can be found in [1].

In addition to the CAO compiler described in this paper, CAO is supported by two other tools: the CAO interactive interpreter and the CAOVerif tool [3], a deductive verification tool inspired by the Frama-C platform.

2 Compiler Architecture

The CAO compiler is logically divided in classical *front-end*, *middle-end* and *back-end* structure. The front-end parses the input file and produces an abstract representation, or *Abstract Syntax Tree* (AST), which is then checked against the typing rules of the language. This results in an annotated AST which is used in subsequent stages. The most distinctive parts of our compiler are the middle-end and the back-end which we will describe in more detail in the following.

2.1 Middle-end

In addition to generating C code, the CAO compiler is also intended to perform meaningful CAO-to-CAO transformations. The middle-end takes the annotated AST and applies a sequence of such transformations towards a CAO format suitable for easy translation to C. The most interesting steps are the following.

Expansion. This optional transformation follows from the fact that most cryptographic algorithms use iterative structures with statically determined bounds. The body of the iteration is unrolled and the loop variables are instantiated.

Evaluation. This transformation evaluates the statically computable expressions, possibly instantiated in the previous step. Operator properties such as idempotence and cancellation are also used to simplify expressions.

Simplification. This transformation is in charge of reducing the mismatch between CAO and C. Compilers that generate assembly code traditionally use an intermediate representation known as three-address code, in which every instruction is in its simpler form with two operand addresses and one result address. Our format shares some of the same principles and, looking ahead, it is consistent with the syntax adopted in the construction of the supporting static libraries.

Optimization. At this stage, the *Control Flow Graph* (CFG) of the CAO code is inferred and transformations to and from *Static Single Assignment* (SSA) form are implemented using adaptations of the algorithms described in [4] and [5]. We provide a set of functions to manipulate the CFG (and CFG in SSA form), to ease the task of implementing (domain-specific) optimization passes.

Side-channel countermeasures. The CAO compiler incorporates a popular software countermeasure against side-channel attacks [2]. The compiler ensures that the code generated for two potentially vulnerable functions (specified by the user) is indistinguishable: both functions execute the same sequence of native CAO operations. To this end, it reorders instructions and, if necessary, introduces dummy operations. The resulting code is kept as efficient as possible by heuristic optimization. This is done after the optimization stage, since optimization could break this security-critical protection. We note that such countermeasures do not guarantee security against side-channel attacks, but are commonly used to increase the resilience of implementations.

2.2 Back-end

Targeting a language like C poses different challenges than translating code to assembly. One of the reasons for this is that the design space is much larger and the C code can be compiled to very disparate platforms. We tackle this problem using a two-layer approach: the CAO code is translated to a specific C format, which is then linked with a static library where the semantics of the CAO operations is implemented and the data types are defined. This allows adjusting the C data type definitions and the implementation of the operations to the characteristics of the target platform. We identified the following variants of static library implementations that may be preferable depending on the target:

- native variable declarations versus complex declarations using C macros;
- automatic static allocation of memory versus explicit dynamic allocation;
- implementing operations using C functions versus using C macros;
- returning results by value versus returning results by reference;
- calling a function by passing values versus passing references;
- translating literals to constants versus initializing auxiliary variables;

– implement operators so as to preserve the input values in arguments versus unsafe implementations.

For each target platform, our back-end takes a configuration file that describes the specific implementation choices adopted for the static library and generates the C code accordingly with the definitions. For example, in the case of variables of a given type use explicit allocation, the compiler will know to call a memory allocation routine. Similarly, if operations over a given type take parameters by reference, then the code generator will make sure the routine receives a pointer to the input parameter.

An important point is that the target platform specification also declares which operations are defined in the static library allowing for incomplete implementations. Therefore, the compilation may fail with an error when the translation is not possible because an operation or data type is not supported.

3 Conclusions and directions for future work

The CAO compiler has been successfully used to implement different cryptographic functions and algorithms, targeting both powerful computational platforms and constrained embedded devices. Example implementations include the SHA family of hash functions, HMAC authentication algorithms, RSA-OAEP encryption and Rabin-Williams signatures. The compiler code is reasonably stable and the current release can be used in real-world contexts. It is available from `http://crypto.di.uminho.pt/CAO` and will soon be published as an open-source project in the Hackage repository.

So far we have only preliminary results regarding a comparative analysis of the tradeoff between the reduction in development time and the performance penalty incurred by using the CAO compiler. Future work will include a more detailed analysis of these trade-offs. Nevertheless, these results indicate that a highly optimized CAO back-end can lead to C implementations with analogous performance to those offered by open-source off-the-shelf cryptographic packages. This is because the output of the CAO compiler is essentially a sequence of calls to an underlying static library, which can incorporate state-of-the-art optimizations, with the extra advantage that these can be transparently reused from one CAO program to another.

Additional directions for future work include improving the compiler efficiency, supporting additional countermeasures against side-channel attacks, and supporting novel cryptographic constructions, namely those based on lattices.

References

1. Barbosa, M., Moss, A., Page, D., Rodrigues, N., Silva, P.: Type checking cryptography implementations. In: FSEN'11. LNCS, vol. 7141, pp. 316–334. Springer (2012)
2. Barbosa, M., Page, D.: On the automatic construction of indistinguishable operations. In: Cryptography And Coding. LNCS, vol. 3796, pp. 233–247. Springer (2005)

3. Barbosa, M., Pinto, J., Filliâtre, J.C., Vieira, B.: A deductive verification platform for cryptographic software. Electronic Communications of the EASST 33 (2010)
4. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. (TOPLAS) 13(4), 451–490 (1991)
5. Sreedhar, V., Ju, R.C., Gillies, D., Santhanam, V.: Translating out of static single assignment form. In: Static Analysis, LNCS, vol. 1694, pp. 194–210. Springer (1999)