Specifying Sharemind's Arithmetic Black Box

Peeter Laud
Cybernetica AS
peeter.laud@cyber.ee

Alisa Pankova
Cybernetica AS
Software Technologies and Applications Competence Centre
University of Tartu, Institute of Computer Science
alisa.pankova@cyber.ee

Martin Pettai
Cybernetica AS
Software Technologies and Applications Competence Centre
University of Tartu, Institute of Computer Science
martin.pettai@cyber.ee

Jaak Randmets
Cybernetica AS
University of Tartu, Institute of Computer Science
jaak.randmets@cyber.ee

ABSTRACT

In this paper, we discuss the design choices and initial experiences with a domain-specific language and its optimizing compiler for specifying protocols for secure computation. We give the rationale of the design, describe the translation steps, the location of the compiler in the whole Sharemind protocol stack, and the results we have obtained with the system.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Cryptographic controls*; D.3.4 [**Programming Languages**]: Processors—*Code generation*

Keywords

Secure Multiparty Computation; Protocol optimization

1. THE PROTOCOLS OF SHAREMIND

Existing secure multiparty computation (SMC) frameworks use different protocol sets for achieving privacy. Several frameworks implement the *arithmetic black box* (ABB) [4], the methods of which are called during the runtime of a privacy-preserving computation by the SMC engine in the order determined by the specification of the computation. An ABB must at least contain the methods for linear combination and multiplication of private integers, but it contains more in typical implementations.

Sharemind SMC framework [2] features an exceptionally large ABB. Besides the operations listed above, it also contains comparison, bit extraction, widening, division of arbitrary-width integers [3], as well as a full set of floating-point [5] and fixed-point operations, including the implementations of elementary functions. More protocol sets on top of different SMC methods are planned.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PETShop'13, November 4, 2013, Berlin, Germany.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2489-2/13/11 ...$15.00.
http://dx.doi.org/10.1145/2517872.2517874.

Different protocols of the ABB form a hierarchy, with more complex protocols invoking simpler ones (multiplication, widening, certain bit-level operations) for certain tasks [3].

The implementation of protocols for ABB operations is an error-prone and repetitive task. Repetition is caused by implementing the protocols to work with values of different bit-length. Attempts to optimize complex protocols over the composition boundaries introduce errors and make the library of protocols unmaintainable. The task of building and maintaining implementations of protocols is naturally answered by introducing a domain-specific language (DSL) for specifying them.

The DSL allows us to specify the protocols in a manner similar to their write-up in papers on SMC protocols. This specification is compiled and linked with the Sharemind platform. There is a different language [1] for specifying the privacy-preserving applications as a composition of these protocols. Having different languages for implementing different levels of the privacy-preserving computation allows us to apply optimizations most suitable for each level, and improves the user experience by allowing us to tailor the languages for the specific domain. Protocols are specified and implemented in a declarative style, but applications are implemented in imperative style as a sequence of protocol invocations.

2. THE LANGUAGE FOR PROTOCOLS

Our protocol DSL is a functional language, mimicking the style of the pseudocode used to present protocols. A program in this language states, which party computes which values from which previously available values. Computations used several times can be abstracted as functions.

The language allows to state only once similar computations performed by different parties. Actually, this is the default mode and each variable `x` in the program denotes a separate value at each party. Defining `x=f(y1,...,yn)` causes each party to apply `f` to its own values of `y1`,...,`yn` and denote the result as `x`. To access the value of `x` at a particular party no. i, one may write `x from` i. In party i's code, the pseudo-numbers `Prev` and `Next` denote the parties no. $(i-1)$ and $(i+1)$ (*modulo* the number of parties). There are specific syntactic constructs to state that certain computations have to be made only by a subset of parties.

Fig. 1 gives the specification for multiplying numbers $u, v \in \mathbb{Z}_{2^n}$, additively shared between three parties [2] (i.e. a private value $x \in$

```
parties 3

def reshare (u : uint[n]) : uint[n] = {
  let r = rnd (),
      w = u - r + (r from Prev);
  return w;
}

def mult (u : uint[n], v : uint[n]) : uint[n] = {
  let u' = reshare (u),
      v' = reshare (v),
      w  = u' * v' + u' * (v' from Prev) + (u' from Prev) * v';
  return reshare (w);
}
```

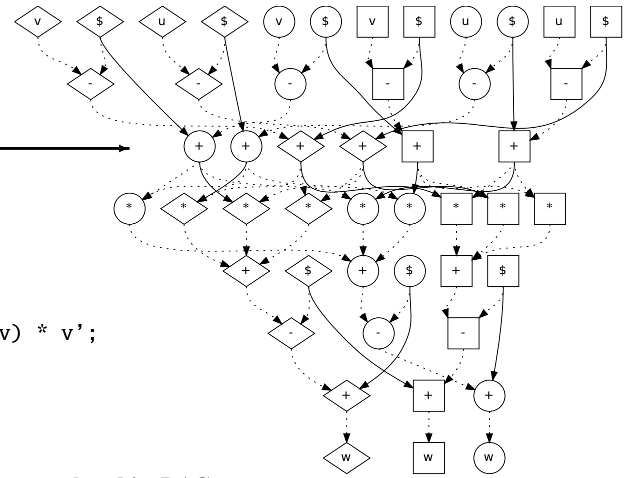

Figure 1: Multiplication protocol and its DAG

$\mathbb{Z}_{2^n}$ is represented as each party i holding $x_i \in \mathbb{Z}_{2^n}$, satisfying $x_1 + x_2 + x_3 \equiv x \pmod{2^n}$)). We see the similarity with Alg.s 1&2 in [3].

The type system of our DSL is inspired by Cryptol [7], and at its core the type system is Hindley-Milner extended with constraints. There is one basic type — `bit` — and one data type constructor for arrays. `uint[n]` is a synonym for an array of n bits where n is a type-level size variable. Size polymorphism allows the protocols to be specified once for any input length. Similarly to Cryptol, our types can be refined with linear constraints over type variables.

Integration with Sharemind.

The specified protocols are used to generate protocol implementations for the Sharemind platform. While the specifications have been polymorphic in the bit-width of the arguments and the result, the Sharemind protocols work with bit-strings of fixed length. Hence, together with our protocols we also specify, for which protocols we want the implementations to be generated, and what should be the values of the type variables in the implementations. E.g., Sharemind currently has protocols for multiplying 8-, 16-, 32-, and 64-bit integers.

The protocols are first translated to an intermediate representation, described below, and then to implementations in C++. The implementations make use of the features offered by the platform, in particular the primitives for communicating between different parties. The necessary communication between parties is derived from the accesses of a party's values from the code executed by a different party; the communication is realized with the help of Sharemind's networking API, packing all values communicated at the same round into a single, or a few messages of suitable length. During the translation to the intermediate representation, all polymorphism is resolved, hence each compiled protocol is used with values of a particular length and lengths of all exchanged messages are known at compile-time.

3. ARITHMETIC CIRCUITS

Arithmetic circuits are the intermediate representation in our protocol compiler; this representation is used for optimizations. An arithmetic circuit is a directed acyclic graph (DAG), where the vertices are labeled with operations and the incoming edges of each vertex are ordered. The input nodes of the circuit correspond to the representation of the inputs to the ABB operation that this protocol implements; in case of protocol sets based on secret sharing, each input is represented by a number of nodes equal to the number of the protocol parties. Similarly, the output nodes correspond to shares of the protocol output.

Communication between parties is expressed implicitly: each node of the circuit is annotated with the executing party, and an edge between nodes belonging to different parties denotes communication. Such representation makes both the aspects of computation (relationships between values) and communication (how many bits are sent in how many rounds?) in the protocol easily accessible for analyses and optimizations.

To compile the protocols specified in our protocol DSL to circuits, the loops have to be unrolled, function calls inlined, etc. The type system and the compiler of the DSL ensure that loop counts and function call depths (even for recursive functions) are known during the compile time. If the control flow of a protocol requires the knowledge of (public) data known only at runtime (e.g. the length of an array), then this protocol cannot be fully specified in the protocol DSL and the language of [1] has to be at least partially used. The circuit corresponding to the multiplication protocol is shown in the right of Fig. 1. Different parties are identified by different node shapes. A solid edge denotes communication. We see that this protocol requires two rounds, because there are paths in this graph that contain two solid edges.

The intermediate representation is used to optimize the protocols. Due to the compositional nature of specification, the protocols typically contain constants that can be folded, duplicate computations, etc. So far, we have implemented all optimizations analogous to the ones reported in [6] for Boolean circuits (constant propagation, merging of identical nodes, dead code removal). But as our circuits are much smaller (the biggest ones corresponding to protocols in [3] have tens of thousands of nodes), and the arithmetic operations allow much more information about the computation to be easily gleaned, we have also successfully run more complex optimizations. We can simplify certain arithmetic expressions (e.g. linear combinations), also if communication is involved inbetween. Interestingly, we can move certain computations from one party to another, or even duplicate computations, if it results in the decrease of communication (which is the bottleneck for current protocols of Sharemind). In the multiplication protocol in Fig. 1, we can reduce the number of rounds to 1 by duplicating the six addition nodes and assigning them to different parties (duplicated "circles" become "diamonds", "diamonds" become "boxes" and "boxes" become “circles”). This does not make a secure protocol insecure because it does not make the view of any party richer than it was. We are currently developing a comprehensive library of optimizations for such distributed arithmetic circuits.

We have tried out the optimizations on certain protocols described in [3]. We have managed to reduce the amount of communication of the largest protocols by around 4% (7% when not considering randomness that could be predistributed). Also, the number of rounds the protocols need is reduced by 1–2, compared to [3]. The composition of protocols also creates many places where constant propagation and merging are useful.

4. ACKNOWLEDGEMENTS

This work was supported by the European Social Fund through the ICT Doctoral School programme, and by the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS, and through the Software Technologies and Applications Competence Centre, STACC. It has also received support from Estonian Research Council through project PUT2.

5. REFERENCES

[1] D. Bogdanov, P. Laud, and J. Randmets. Domain-polymorphic programming of privacy-preserving applications. Cryptology ePrint Archive, Report 2013/371, 2013. `http://eprint.iacr.org/`.

[2] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In S. Jajodia and J. López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.

[3] D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec.*, 11(6):403–418, 2012.

[4] I. Damgård and J. B. Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In D. Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2003.

[5] L. Kamm and J. Willemson. Secure Floating-Point Arithmetic and Private Satellite Collision Analysis. 2013. Submitted.

[6] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 285–300. USENIX Association, 2012.

[7] J. Lewis. Cryptol: specification, implementation and verification of high-grade cryptographic applications. In P. Ning, V. Atluri, V. D. Gligor, and H. Mantel, editors, *FMSE*, page 41. ACM, 2007.