CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations*

Martin Franz[1], Andreas Holzer[2], Stefan Katzenbeisser[3], Christian Schallhart[4], and Helmut Veith[2]

[1] Deutsche Bank
[2] TU Wien
[3] TU Darmstadt & CASED
[4] Oxford University

Abstract. Secure two-party computation (STC) is a computer security paradigm where two parties can jointly evaluate a program with sensitive input data, provided in parts from both parties. By the security guarantees of STC, neither party can learn any information on the other party's input while performing the STC task. For a long time thought to be impractical, until recently, STC has only been implemented with domain-specific languages or hand-crafted Boolean circuits for specific computations. Our open-source compiler CBMC-GC is the first ANSI C compiler for STC. It turns C programs into Boolean circuits that fit the requirements of garbled circuits, a generic STC approach based on circuits. Here, the size of the resulting circuits plays a crucial role since each STC step involves encryption and network transfer and is therefore extremely slow when compared to computations performed on modern hardware architectures. We report on newly implemented circuit optimization techniques that substantially reduce the circuit sizes compared to the original release of CBMC-GC.

Keywords: Secure Computations, Privacy, Compilers, Circuit Optimization.

1 Introduction

Imagine Alice and Bob as two millionaires who want to determine the richer one among them – but *without* revealing how much they own, neither to the other millionaire nor to somebody else. This is the "millionaires' problem", first described by Yao [18], who thereby initiated research on secure two party computation (STC). Subsequently it has been shown that every computable function over two inputs is also computable in the framework of STC: Two players can evaluate the function on their respective private inputs so that the result of the computation is available to both, without needing to share the inputs with each other.

In modern information processing infrastructures, not only data but also code is becoming more mobile, e.g., in cloud services. Thus, with the increasing amount of sensitive information processed, and facing laws and regulations that are not only hard

[*] This work was supported in part by the Austrian National Research Network S11403 and S11405 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grant PROSEED, and by CASED.

A. Cohen (Ed.): CC 2014, LNCS 8409, pp. 244–249, 2014.

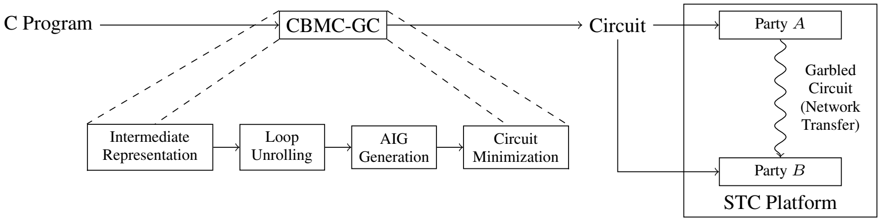

Fig. 1. STC Tool Chain

to understand but even harder to enforce across national boundaries, the demand for technical solutions is growing. These solutions, called Privacy Enhancing Technologies (PETs), assure data secrecy and privacy, even if data is processed on potentially untrusted platforms. The central cryptographic tool enabling such PETs is STC, allowing two distrusting parties to perform arbitrary computations on sensitive data without ever exposing their input in the clear. Hence, no information on the other party's input is revealed, beyond the information derivable from the commonly computed function output.

After 30 years of mainly theoretical studies, increased computational power and advanced cryptographic protocols make it feasible to evaluate reasonably large functions in an STC context [2,5,8,4,17]. The predominant approach to implement STC are Garbled Circuits (GCs), as originally proposed by Yao [19], working in two steps: First, Alice garbles a given circuit and hands this garbled circuit to Bob, together with a set of keys representing Alice's input. Using Oblivious Transfer [16], Bob obtains the set of keys corresponding to his own input, without obtaining any other key, and such that Alice does not know which keys Bob took. With these keys, Bob can evaluate the garbled circuit – unable to learn anything on Alice's input that is not implied by the final output. We refer to [12] for details.

One main obstacle for practical application of STC was the lack of support for general programming languages, as only circuit evaluation [7] or simplified programming languages [13] were supported. Recently at CCS [6], we presented CBMC-GC, the **first STC compiler for full ANSI C**.We argue that practical application of STC should be viewed as a combination of compiler and security research (cf. Figure 1): (i) **STC compilation**, i.e., the STC compiler translates the source code into a circuit that is optimized towards its use in STC and (ii) **STC interpretation**, i.e., the STC framework evaluates generated circuits in a way that ensures the STC guarantees. We believe that this separation of concerns is a crucial step towards broad practical use of STC.

```c
#include <cbmc-gc.h>
void millionaires() {
  int a, b, result;
  __CBMC_GC_INPUT_A(1, a);
  __CBMC_GC_INPUT_B(2, b);
  result = (a > b)?1:0;
  __CBMC_GC_OUTPUT(3, result);
}
```

Fig. 2. C code for Yao's millionaires' problem

Figure 1 shows CBMC-GC in the STC tool chain. CBMC-GC translates a C program into a circuit which is then deployed to the two STC parties A and B. The STC framework is essentially an interpreter for the circuit. In our current implementation, we use the GC construction proposed in [10] with optimizations from [9,15], allowing XOR-gates to be evaluated at essentially no cost. After compilation, party A garbles the circuit including party A's input and sends the resulting garbled circuit to party B. Due to the potentially huge size of garbled circuits, party B evaluates the circuit on-the-fly instead of storing it in memory. We refer to [12] for details and a security proof, only sketching the STC evaluation.

(1) *Garbling.* Party A assigns to each circuit wire w two random keys K_w^T and K_w^F, each representing one truth value of w (T = true, F = false). For all binary gates $G(u, v) = o$ with input wires u, v and output wire o, party A encrypts each entry $(\mathsf{val}(u), \mathsf{val}(v), \mathsf{val}(o))$ of G's truth table by computing

$$\mathsf{encrypt}_{K_u^{\mathsf{val}(u)}}(\mathsf{encrypt}_{K_v^{\mathsf{val}(v)}}(K_o^{\mathsf{val}(o)})),$$

i.e., $K_o^{\mathsf{val}(o)}$ gets encrypted using the keys $K_u^{\mathsf{val}(u)}$ and $K_v^{\mathsf{val}(v)}$. Therein, $\mathsf{val}(u)$ is the evaluation of u (i.e., the truth value T or F), and hence, if G is, say, an or-gate, party A garbles the entry (F, T, T) by encrypting K_o^T with K_u^F and K_v^T; finally, A permutes the resulting four encrypted keys so that the evaluating party does not see which encrypted key corresponds to which entry of the truth table. If G is an output gate, A encrypts no further key but the plain truth value from G's truth table.

(2) *Evaluation.* The garbled circuit is handed to party B together with the keys corresponding to party A's input. B obtains the keys corresponding to its own inputs with Oblivious Transfer [16], guaranteeing that B only obtains one key per input wire, and guaranteeing that A does not know which keys B has chosen. With these keys, B decrypts inductively the keys for the truth values corresponding to the valuation of the wires in the circuit under the combined inputs of A and B – and importantly, B can only decrypt those. More precisely, for each gate he tries to decrypt all four (permuted) truth table entries; only one decryption will succeed, giving him the necessary key for the subsequent gate.

CBMC-GC solves the millionaires' problem with the source code shown in Figure 2: The procedure `millionaires` is a standard C procedure, where only the input and output variables are specifically marked up, designated as input of party A or B (Lines 4 and 5) or as output (Line 7). But aside this input/output convention, arbitrary C computations are allowed to produce the desired result, in this case a simple comparison (Line 6). This paper presents CBMC-GC v0.9, an improved version of the compiler presented at CCS [6] which combines various techniques known from logic optimization to produce substantially smaller circuits.

2 CMBC-GC in a Nutshell

Our compiler CBMC-GC[1] is based on the software verification tool CBMC [3]. Since CBMC is a bounded model checker for ANSI C, it translates any given C program into

[1] `http://forsyte.at/software/cbmc-gc/`

Table 1. Circuit sizes produced by CBMC-GC v0.8 and v0.9

	#gates (v0.8)		#gates (v0.9)	
Benchmark	total	non-XOR	total	non-XOR
`Hamming distance, 320 bit`	19031	6038	4010	924
`Hamming distance, 800 bit`	47816	15143	10119	2344
`Hamming distance, 1600 bit`	95791	30318	20356	4738
`matrix multiplication, 5x5`	797751	221625	401250	148650
`matrix multiplication, 8x8`	3267585	907776	1636096	600768
`2000 arithmetic operations`	1531601	405640	938671	319584
`3000 arithmetic operations`	2298441	608668	1417684	479463
`median, merge sort, 21 elements`	750471	244720	210727	136154
`median, merge sort, 31 elements`	1840339	602576	550918	348761
`median, bubble sort, 21 elements`	346380	112800	67050	40320
`median, bubble sort, 31 elements`	1066470	349600	147600	89280

XOR gates are evaluated at essentially no cost and therefore non-XOR gates are mentioned explicitly. For details on the benchmarks see [6].

a Boolean constraint which represents the program behavior at a bit-precise level up to a bounded number of steps. In a nutshell, we adapted this capability of CBMC to provide the circuits needed for STC. The compilation is divided into four steps, where the first two steps are part of the standard CBMC processing and the second two are specific to STC tasks. For more details on the first two compilation steps, please see [6].

(1) *Intermediate Representation.* The C program gets translated into an intermediate representation—a so-called GOTO program. The only control structures remaining in a GOTO program are guarded GOTOs.

(2) *Loop Unrolling.* Loops and recursive function calls are unrolled up to a specific depth. CBMC-GC tries to compute this depth by a static analysis, but in case of failure, the depth can be specified by the user. After unrolling, we have a loop-free representation of the program.

(3) *AIG Generation.* It remains to translate each program statement into a circuit which encodes the bit-precise semantics of the computation the statement performs. CBMC-GC uses *and-inverter graphs* (AIGs) as an intermediate circuit representation. AIGs are directed acyclic graphs whose nodes represent logical AND gates. The edges of an AIG represent wires between gates. Some of these wires can negate the transmitted signal. Throughout the generation of this intermediate circuit, structural hashing, i.e., the removal of duplicated gates, and constant propagation are performed to keep the resulting circuit small [14]. CBMC-GC incorporates the ABC framework [1] to generate the intermediate representation.

(4) *Circuit Minimization.* XOR gates are preferable due to their small computation costs and therefore the circuit minimization step tries to maximize the number of XOR gates in the resulting circuit while keeping the overall circuit size small. Here, a repeated pattern based subcircuit rewriting is performed in combination with structural hashing, constant propagation, and a simplified version of SAT-sweeping [11].

By compiling the source code with CBMC-GC, we obtain a description of the circuit performing the computation and a mapping between in- and output identifiers and the corresponding circuit pins. Table 1 compares the circuit sizes produced by CBMC-GC v0.8 and CBMC-GC v0.9. The benchmarks were originally used to show the practicality of CBMC-GC v0.8 and are discussed in detail in [6]. We can observe a considerable reduction of circuit sizes when using CBMC-GC v0.9 instead of CBMC-GC v0.8.

References

1. Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 30916, `http://www.eecs.berkeley.edu/~alanmi/abc/`
2. Bogetoft, P., Damgård, I.B., Jakobsen, T., Nielsen, K., Pagter, J.I., Toft, T.: A Practical Implementation of Secure Auctions Based on Multiparty Integer Computation. In: Di Crescenzo, G., Rubin, A. (eds.) FC 2006. LNCS, vol. 4107, pp. 142–147. Springer, Heidelberg (2006)
3. Clarke, E., Kroning, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
4. Erkin, Z., Franz, M., Guajardo, J., Katzenbeisser, S., Lagendijk, I., Toft, T.: Privacy-Preserving Face Recognition. In: Goldberg, I., Atallah, M.J. (eds.) PETS 2009. LNCS, vol. 5672, pp. 235–253. Springer, Heidelberg (2009)
5. Goethals, B., Laur, S., Lipmaa, H., Mielikainen, T.: On secure scalar product computation for privacy-preserving data mining. In: ICISC 2004 (2004)
6. Holzer, A., Franz, M., Katzenbeisser, S., Veith, H.: Secure Two-Party Computations in ANSI C. In: CCS 2012 (2012)
7. Huang, Y., Evans, D., Katz, J., Malka, L.: Faster Secure Two-Party Computation Using Garbled Circuits. In: USENIX 2011 (2011)
8. Jagannathan, G., Wright, R.N.: Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In: KDD 2005 (2005)
9. Kolesnikov, V., Sadeghi, A.-R., Schneider, T.: Improved Garbled Circuit Building Blocks and Applications to Auctions and Computing Minima. In: Garay, J.A., Miyaji, A., Otsuka, A. (eds.) CANS 2009. LNCS, vol. 5888, pp. 1–20. Springer, Heidelberg (2009)
10. Kolesnikov, V., Schneider, T.: Improved Garbled Circuit: Free XOR Gates and Applications. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 486–498. Springer, Heidelberg (2008)
11. Kuehlmann, A.: Dynamic transition relation simplification for bounded property checking. In: ICCAD 2004 (2004)
12. Lindell, Y., Pinkas, B.: A Proof of Security of Yao's Protocol for Two-Party Computation. Journal of Cryptology 22, 161–188 (2009)
13. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay — A Secure Two-Party Computation System. In: SSYM 2004 (2004)
14. Mishchenko, A., Chatterjee, S., Brayton, R.: FRAIGs: A Unifying Representation for Logic Synthesis and Verification. Technical report (2005)
15. Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure Two-Party Computation Is Practical. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 250–267. Springer, Heidelberg (2009)

16. Rabin, M.O.: How To Exchange Secrets with Oblivious Transfer. IACR Cryptology ePrint Archive 2005, 187 (2005)
17. Smaragdis, P., Shashanka, M.V.S.: A framework for secure speech recognition. IEEE Transactions on Audio, Speech & Language Processing 15(4), 1404–1413 (2007)
18. Yao, A.C.-C.: Protocols for Secure Computations (Extended Abstract). In: FOCS 1982 (1982)
19. Yao, A.C.-C.: How to Generate and Exchange Secrets. In: FOCS 1986 (1986)