Secure Two-Party Computations in ANSI C

Andreas Holzer
TU Wien

Martin Franz
CrypTool Project

Stefan Katzenbeisser
TU Darmstadt & CASED

Helmut Veith
TU Wien

ABSTRACT

The practical application of Secure Two-Party Computation is hindered by the difficulty to implement secure computation protocols. While recent work has proposed very simple programming languages which can be used to specify secure computations, it is still difficult for practitioners to use them, and cumbersome to translate existing source code into this format. Similarly, the manual construction of two-party computation protocols, in particular ones based on the approach of garbled circuits, is labor intensive and error-prone.

The central contribution of the current paper is a tool which achieves Secure Two-Party Computation for ANSI C. Our work is based on a combination of model checking techniques and two-party computation based on garbled circuits. Our key insight is a nonstandard use of the bit-precise model checker CBMC which enables us to translate C programs into equivalent Boolean circuits. To this end, we modify the standard CBMC translation from programs into Boolean formulas whose variables correspond to the memory bits manipulated by the program. As CBMC attempts to minimize the size of the formulas, the circuits obtained by our tool chain are also size efficient; to improve the efficiency of the garbled circuit evaluation, we perform optimizations on the circuits. Experimental results with the new tool CBMC-GC demonstrate the practical usefulness of our approach.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*security and protection*

General Terms

Security, Programming Languages

Keywords

Secure Computations, Privacy, Model Checking, Compilers

1. INTRODUCTION

In the thirty years since Yao's seminal paper [31], Secure Multiparty Computation (SMC) and Secure Two-Party Computation (STC) have emerged from purely theoretic constructions to practical tools. In STC, two parties jointly evaluate a function f over two inputs A and B provided by the parties in such a way that either party keeps its input unknown to the other; SMC is an analogous method for multiple parties. STC and SMC enable the construction of Privacy-Enhancing Technologies which protect sensitive data during processing steps in untrusted environments.

The SMC/STC concept has been applied successfully to secure electronic auctions (the most famous being the sugar-beet auction of [9]), to conceal valuable biometric data while allowing authentication [16], to analyze and cluster private data on untrusted devices [18, 21] and to analyze medical signals [3]. Technically, the implementations either use the approach of "garbled circuits" introduced by Yao [31], where f is transformed into a Boolean circuit C_f and encrypted in a special way, or homomorphic encryption [15], where the inputs are encrypted and the operation f is directly performed on encryptions. A series of recent publications including [20, 28] demonstrated that secure computations are feasible on today's standard computing equipment.

In this paper we focus on the practical implementation of STC. Despite the versatile use and the obvious benefits of STC, it is currently rarely used in industrial large-scale applications. The main reason for the slow adoption of STC by practitioners is, besides performance problems for large-scale problem settings, the lack of a good tool chain. The implementation of a concrete new functionality f in STC is still cumbersome and error-prone, and even more so without a professional background in computer security. Indeed, most of the publications cited above introduced new protocols and primitives that were tailored towards improving the efficiency of one specific application scenario (in most cases, this included the design of efficient Boolean circuits for specific complex functions which are needed in the construction of garbled circuits).

During the last years we have witnessed the emergence of a number of frameworks such as Fairplay [29], VIFF [1], Tasty [19] and Sharemind [8] that allow to construct STC applications in a generic fashion by specifying the desired functionality f as a program in a tailor-made high-level language. The source code is subsequently transformed into code that runs on a special interpreter or in a dedicated runtime environment. Unfortunately, all existing frameworks are severely limited in their practical usability. Some frameworks, like the one by Huang et al. [20] or VMCrypt [28], do only offer basic (albeit highly optimized) libraries, which leaves the burden on the programmer to securely realize the desired complex functionality out of very basic blocks. For example [20] does not even provide a re-usable implementation for an integer multiplication operation. Most existing frameworks aiming at automatizing the construction of STC protocols [29, 1, 19] can only cope with fairly small programs and come with their own dedicated syntax; furthermore, code is meant to be used stand-alone and cannot easily be integrated with existing code or libraries.

As new technology, STC reminds in certain aspects of computer science in the fifties: there were no general-purpose programming languages, hardware and software were designed together, and it was hard to transfer existing solutions between hardware platforms. The picture changed dramatically when programming languages and compilers were introduced. While the first compilers did not produce very efficient code, they quickly reached the maturity to become the most important commodity tools of a programmer.

In this paper we argue that the technology is available to make the same step with STC:

- We view STC as a cryptographic analogue to a secure hardware platform which gives STC-specific security guarantees. For the working programmer, STC programming should be similar as programming a (quite slow) embedded processor or micro-controller.
- STC programming will become a normal programming task in a standard programming language that has a compiler and other productivity tools (e.g. for testing and debugging).
- The programming language should be standardized. This will allow developing and debugging STC programs on normal hardware before deploying it for STC. Moreover, standardization will aid security and safety certifications.
- The proposed approach will decouple cryptographic research in STC from compiler research for the programming language. The history of programming languages gives ample support for this separation of concerns.

In this paper, we provide a significant step towards this direction. We present CBMC-GC, the first compiler for secure computation of ANSI C programs. CBMC-GC allows a programmer to write the function f in the form of a C program. Thus, the programmer can view STC as a dedicated hardware platform, and compile standard C programs to this platform. To achieve this functionality, the CBMC-GC compiler transforms the C source code into an optimized Boolean circuit, which can subsequently be implemented securely utilizing the garbled circuit approach, for example by re-using VMCrypt, the fastest currently available implementation of garbled circuits [28].

At first sight, our choice of ANSI C may look surprising: While C has a standardized ANSI semantics (cf. Sec. 3.2.6), C is not the latest fashion in programming languages, it does not achieve the platform independence of Java, and it is also lacking advanced programming features such as object orientation. While these objections are valid, they are outweighed by technical and practical arguments in favor of C:

- The C language is quite close to the underlying hardware, and a competent programmer has good control of the actual computation on the processor. In particular, the programmer can control the memory and CPU use of his program better than in other high level languages. This proximity to the hardware platform has made C the language of choice for areas like embedded systems, device drivers and operating systems. For similar reasons, hardware vendors are customarily prototyping hardware components in C.
- As we demonstrate in this paper, this advantage of C translates to STC. The sizes of the circuits that we obtain from compiling C programs are surprisingly small. Since the practicality of STC is heavily dependent on the circuit size, we conclude that with current technology, C is a high level language very suitable for our task. As the history of compiler technology shows, a large number of optimizations can be made to improve our translation further. Thus STC will profit from advances in compiler technology.
- While the semantics of C may be less than pretty to the eye of programming language theorists, the conceptual simplicity of C makes automated safety analysis of C programs (e.g. by software model checking and static analysis) much simpler than for languages such as Java and C++. The last decade has seen dramatic improvements in practical source code analysis tools for C such as Microsoft's SLAM [2], the abstract interpreter ASTREE [14] and the bit-precise model checker CBMC [10]. Our compiler CBMC-GC will make use of existing CBMC technology in an atypical way.
- The availability of software verification tools for C has the additional advantage that we can address correctness of the program in a systematic way. We note that the limited size of programs realizable in STC makes them amenable to the most advanced safety analysis tools.
- C and Java are the two linguae francae of computer science: C still holds top positions in rankings of programming language popularity[1] and in the TIOBE programming community index[2]. Many programmers can write C programs at ease, and a large base of legacy code is available. Working programmers are supported by a multitude of productivity tools, e.g. for debugging, testing and code generation.

Technically, CBMC-GC is based on the software architecture of the model checker CBMC by Clarke et al. [10], which was designed to verify ANSI C source code. CBMC transforms an input C program f, including assertions that encode properties to be verified, into a Boolean formula B_f which is then analyzed by a SAT solver. The formula B_f is constructed in such a way that the Boolean variables correspond to the memory bits manipulated by the program and to the assertions in the program. CBMC is an example of a *bit-precise* model checker, i.e., the formula B_f is encoding the real life memory footprint of the analyzed program on a specific hardware platform under ANSI C semantics.

[1]http://langpop.com

[2]http://tiobe.com/index.php/content/paperinfo/tpci

(CBMC allows the user to configure the hardware platform, e.g. the word size.) The construction of the formula B_f moreover ensures that satisfying assignments found by the SAT solver are program traces that violate assertions in the program.

Thus, CBMC is essentially a compiler that translates C source code into Boolean formulas. The code must meet some requirements, detailed in Section 3, so that this transformation is possible in an efficient manner. In particular, the program must terminate in a finite number of steps; CBMC expects a number k as input which bounds the size of program traces (and CBMC also determines if this bound is sufficient). Our tool CBMC-GC inherits these constraints from CBMC. In practice, these limitations are not overly significant, as every program with bounded runtime can in principle be compiled to a circuit. To our knowledge, this class includes all problems that were considered so far in the context of STC. In fact, the same restrictions typically hold for software used in real-time and embedded systems, which needs to guarantee certain response times.

The architecture of our tool CBMC-GC builds on the core engine of CBMC. We modify CBMC to transform a C program into a Boolean circuit (rather than a formula). While CBMC optimizes the resulting formulas for easy solvability using a SAT solver, we changed the CBMC engine such that it outputs circuits optimized for STC performance, i.e., for garbled circuit evaluation. (Note that our tool CBMC-GC does not require a SAT solver.) The circuit obtained by CBMC-GC is subsequently ready for use in a garbled-circuit based STC framework. In our implementation, for simplicity we use the STC framework of [20], which provides security in the semi-honest attacker model. Nevertheless, the circuit generation process is completely decoupled from its evaluation; by choosing a different underlying STC framework that provides security against malicious adversaries, one can support the malicious case as well.

We evaluated the performance of CBMC-GC on a number of tests that are typically applied to assess the efficiency of STC; furthermore, we show the applicability of CBMC-GC to the problem of secure two-party computations with private functions (i.e., the case where the functionality to be computed is only available to one party). In particular, we consider the size of the circuits produced by CBMC-GC and evaluate their runtime when used in combination with the above mentioned highly efficient STC implementation [20]. We show that CBMC-GC allows STC on ANSI C programs to be deployed with good practical performance. Furthermore, we conclude that separating the problems of garbled circuit evaluation and efficient automatic circuit generation from a standard high-level programming language is a promising strand of research, enabling the practical uptake of STC.

2. RELATED WORK

2.1 Secure Two-Party Computation

Garbled Circuits (GC) have been introduced in [32] as a generic solution for the problem of securely evaluating a function. GC enable two parties to evaluate a Boolean circuit on their respective private inputs in a way that the computation reveals only the output of the function. In a nutshell, the function to be evaluated is represented by a Boolean circuit, where the input wires of the circuit represent the inputs of the respective parties. One party first garbles the circuit by assigning two random keys to each wire and encrypting the operation table of each gate: assuming that the gate has in-degree two, each entry in the table is replaced by the corresponding key and encrypted twice, namely with the keys that correspond to the truth values on the input wires, and all entries of the gate are permuted. Finally, the party hands the garbled circuit including the keys corresponding to his input values to the other party, who first obtains the keys to her input values using Oblivious Transfer and subsequently evaluates the garbled circuit. For details as well as a security proof we refer to [27].

Since the pioneering work of [32] garbled circuits have been frequently improved. Currently the two most advanced implementations of the garbled circuit approach are [20, 28]. The authors use the GC construction proposed in [26] combined with optimizations in [24, 30]. This allows XOR-gates to be evaluated at essentially no cost and each garbled circuit table consists of three entries instead of four. Furthermore, the implementation is tailored towards efficiency; thus, even circuits consisting of several million gates can be handled.

Several efforts have been made by different authors to improve the practical uptake of STC. Fairplay [29] was the first framework which allows to implement generic STC in a high level language. Later this was generalized to FairplayMP, a framework for secure multiparty computation [4]. Programs for Fairplay have to be specified in a high-level programming language, the Secure Function Definition Language (SFDL), which shares some similarity to the VHDL language, allowing basic support for integers and instructions for Boolean operations. The Fairplay compiler translates these programs into a garbled circuit, which is executed in a special runtime environment written in Java. Although SFDL 2.1 supports data types like structs and arrays, it lacks support for unsigned integer data types, enumerations, unions, and pointers. ANSI C also provides a richer set of statements like `while`-loops or recursive function calls. CBMC-GC strives to support these features (for limitations see Section 3.2.6). Tasty [19] was the first tool that allowed to combine SMC techniques from homomorphic encryption with garbled circuits. Programs are specified in a Python-like programming language, which allows for basic data types and arithmetic operations.

So far, none of the aforementioned tools provides the desired properties as mentioned in the introduction: A wide distribution and acceptance of the programming language; existing code which can be reused; books and tutorials which make the programming language broadly accessible; a large number of existing tools, e.g. for testing or verification.

2.2 Model Checking

The last decade has seen a revolution in practical safety analysis tools for software. Due to its simplicity and its relevance for safety-critical industries, C has become the primary target for these tools. The majority of the tools is based on overapproximation, e.g. by abstract interpretation [14] or by predicate abstraction and CEGAR abstraction refinement [2, 5, 13]. The alternative to overapproximation is bit-precise reasoning – the work relevant to our paper. In bit-precise reasoning, the program semantics is precisely modeled in a suitable logical formalism, most importantly in Boolean logic. While overapproximating model checkers often model e.g. integer variables by unbounded (mathematical) integer values, a bit-precise model checker will typically model them as a bit vector with real-life overflow behavior. The most well-known bit-precise model checkers are Kroening's tool CBMC [10] and NEC's tool DiVer [17].

For a given bound k, the model checker CBMC transforms the program into a Boolean constraint whose solutions are program traces of size at most k which violate one or more assertions in the program. These solutions are then determined by a Boolean SAT solver. Importantly, this transformation identifies the bits manipulated during program execution with Boolean variables in the formula. Notwithstanding the high theoretical complexity of SAT solving, modern SAT solvers [7] can often solve verification constraints with millions of clauses. The Boolean encoding also enables the SAT solver to determine whether the bound k was sufficient, i.e., whether the program terminates after k steps. Note that in common terminology, CBMC can also be classified as an instance of SAT-based model checking and bounded model checking [6].

The important property of CBMC for our purposes is its capability to generate bit-precise Boolean descriptions of the program execution from the source code. This capability forms the basis for the compiler which we describe in the next section.

3. FROM C PROGRAMS TO CIRCUITS

In this section, we describe our compiler, which takes C source code and generates a circuit representation by using techniques adapted from the software model checker CBMC [10, 11].

As explained above, CBMC reads a C program f containing assertions along with a bound k, and generates a Boolean constraint B_f in CNF such that the satisfying assignments of C_f encode program traces of size at most k that violate the assertions. Our goal is to reuse the functionality of CBMC to generate a Boolean circuit C_f which is bit-equivalent to the C program. We will first describe the functionality of CBMC, and then explain our modifications.

3.1 CBMC Architecture

We will now summarize the workflow inside the bit-precise bounded model checker CBMC.

On input of an ANSI C program f and a bound k, CBMC first translates the program into a cycle-free GOTO program. In a GOTO program, all control statements like while-loops are transformed into if-then-else statements with conditional jumps, similarly to assembler language. To make the program acyclic, the loops are replaced by a sequence of k nested `if` statements; the sequence is followed by a special assertion (called *unwinding assertion*) which can detect a missed loop iteration due to insufficient k. Similarly, recursive function calls are expanded k times. This process is called "unwinding" the program. For programs with at most k steps, unwinding preserves the semantics of the program.

Once the program is acyclic, CBMC turns it into single-static assignment (SSA) form. This means that each variable x in the program is replaced by fresh variables $x_1, x_2, \ldots$ where each of them is assigned a value only once. For instance, the code sequence `x=x+1; x=x*2;` is replaced by $\mathtt{x}_2$ `=` $\mathtt{x}_1$ `+ 1;` $\mathtt{x}_3$ `=` $\mathtt{x}_2$ `* 2;` SSA format has the important advantage that we can now view the assignments to program variables as mathematical equations. (Note that, as an equation, `x=x+1;` is unsolvable.) The indices of the variables essentially correspond to different intermediate states in the computation. It is thus possible to transform the program into a large quantifier-free formula involving equations and operations over the variables.

In the next step, CBMC replaces the variables by bit vectors. For instance, depending on the architecture, an integer variable will be represented by a bit vector of size 16 or 32. For more complex variables such as arrays and pointers, CBMC uses more advanced techniques [11, 12] whose presentation we omit for simplicity. (Note that in n program steps at most $O(n)$ memory cells can be accessed; this can be exploited by a clever Boolean encoding.) Correspondingly, the operations over the variables (e.g. arithmetic computations or comparisons) are naturally translated into Boolean functions over the corresponding variables. Internally, CBMC realizes these Boolean functions as *circuits* whose construction principles are inspired by methods from hardware design. In the default setting, CBMC translates the resulting circuit into a CNF formula B_f which conjoins the formula for the program semantics with the claim that an assertion is violated.

By construction, the circuit and hence the Boolean formula encode the semantics of the program exactly: the formula is satisfiable if and only if there is a program execution in the unwound program leading to an assertion violation. Due to the fixed unwinding of the program, the tool might miss a bug in case some loop was not unrolled sufficiently often. These cases however are detected by CBMC as explained above.

3.2 From CBMC to CBMC-GC

In this section, we show how to utilize the capability of CBMC to generate Boolean circuits; as explained in the previous section, these circuits express the computation of a C program in a bit-precise manner. Note that for our purposes, code assertions and SAT solvers are irrelevant.

We chose CBMC as our architecture because CBMC is very well maintained, has a clean software architecture and is well documented. Our tool is based on version 2.4 of CBMC[3] and performs the following steps, described in more detail in the rest of the section:

1. *Syntactic Preprocessing:* The input C program is syntactically pre-processed in order to perform some optimizations that can best be implemented on the level of C programs, but reduce the resulting circuit size significantly.

2. *Circuit Synthesis:* The resulting processed C program is handed over to a modified version of CBMC, which internally creates a circuit representation of the program. We leave placeholders for certain basic operations whose implementation is particularly important for the efficiency of STC, e.g. full adders and multiplexers.

3. *Circuit Optimization:* In a final step, the output circuit is assembled by replacing the placeholders with circuit implementations that are favorable for STC, i.e., have large numbers of XOR gates. The final circuit is stored as a netlist (a list of basic operations such as AND, OR, ADD, etc.).

[3]`http://www.cprover.org/cbmc/`

3.2.1 Syntactic Preprocessing

Given the input program, our tool first performs *syntactic loop unrolling:* It attempts to identify simple `for`-loops, and replaces them with repeated copies of the loop bodies, where the loop variables are replaced by constants. Furthermore, we do *constant propagation* to simplify the computations of the program as much as possible. The main purpose of this step is to optimize handling of arrays. Whenever an element of an array is read or stored in a way that the address of the element is not known at compile time, standard CBMC models this access with a multiplexer circuit; multiplexers require a significant number of gates and their evaluation is quite costly. If however the exact address is known during compilation time, the address can be hard-wired into the circuit in order to avoid the overhead. The result of this preprocessing step is stored as an ANSI compliant C program, which is next fed to the tool CBMC.

3.2.2 Circuit Synthesis

As described above, CBMC uses circuits for the internal bit-precise representation of the program. We use this capability to obtain the circuits that we need as input for STC.

At some places, we had to modify the circuit generation of CBMC for a subtle reason: Since CBMC aims to produce good instances for a SAT solver, it has the freedom to use circuits which are equisatisfiable with the circuits we expect, but not logically equivalent. This happens in one construction: CBMC sometimes introduces circuits with free input variables, and adds constraints which requires them to coincide with other variables. In these places, we had to change the circuit generation to reflect actual computation.

Moreover, we perform two additional steps which improve practical efficiency. First, for important basic operations (such as additions and multiplexers) we prevent CBMC from hardwiring the corresponding circuits, but represent them by distinguished "placeholder gates" which are instantiated in step 3 below. Second, we do a reachability analysis on the resulting circuit to prune branches of the circuit which contain dead code. The resulting circuit is finally handed to the next stage of the toolchain.

3.2.3 Circuit Optimization

The circuits produced by standard CBMC have good sizes but are not optimal from the point of view of secure computation. Especially, XOR gates are preferable to other gates, as they can be evaluated essentially without any cost [26]. In the last step we thus instantiate the remaining placeholder gates with efficient implementations. For example, unmodified CBMC translates one addition in the C program into a full-adder composed of 4 non-XOR gates per bit; in contrast, we instantiate the addition with a full-adder requiring 4 XOR gates and one other gate, whose evaluation is faster, despite the larger number of gates. At the moment, our tool replaces addition operations, multiplexers and integer comparisons with hand-optimized versions. However, the tool offers full flexibility; once our modified version of CBMC is able to identify other basic blocks that turn out to be performance bottlenecks, they can easily be replaced by optimized circuits.

We output the generated circuit as a netlist with additional mappings of input and output variables to input and output pins in the circuit. At the moment our netlist format is text-based, but we plan to replace it by a more space-efficient binary format. An STC framework can subsequently read the netlist and translate it into its internal circuit representation.

3.2.4 Input Syntax

CBMC-GC accepts ANSI C programs with multiple functions as input. The programmer needs to specify in the form of a command-line option which function is considered to be the main function, implementing the desired functionality. Secret inputs of both parties need to be specified as variables, where the variable name is preceded by `INPUT_A_` and `INPUT_B_`, respectively. The output, which will be available to both parties after the computation terminated, must be stored in variables whose names start with the prefix `OUTPUT_`. For example, a CBMC-GC program that solves Yao's Millionaires problem can be specified in a very simple way:

```
void millionaires() {
int INPUT_A_mila;
int INPUT_B_milb;
int OUTPUT_res;

  if (INPUT_A_mila > INPUT_A_milb)
    OUTPUT_res = 1;
  else
    OUTPUT_res = 0;
}
```

3.2.5 Security & Correctness

When considering the security and correctness of the approach one can distinguish three different aspects. First, the garbled circuit approach is known to be secure for the execution of any circuit [27]. CBMC-GC produces circuits that are executed on that platform. Second, the correctness of the compilation step reduces to the question of verifying the correctness of a compiler, a topic of current interest in programming languages. While no formal correctness proof is available, CBMC is widely used in the research community and quite mature. Third, having the description of the circuit available as C code, we can test functional properties of the program by testing the C code. In particular, we can apply the unmodified version of CBMC to verify correctness properties (assertions contained in the program).

3.2.6 Limitations

Bounded Programs. Since a circuit can only encode a fixed number of computation steps, CBMC-GC requires for each loop or recursive function call a constant bound (similar to the situation of embedded systems software, where computations have to be bounded as well). CBMC implements a static analysis that can automatically determine loop bounds in many cases; obviously this does not always work, as the problem itself is undecidable. However, for each of the examples given in Section 4, CBMC determined the loop bounds automatically. In case CBMC fails to determine the loop bounds it is possible to state the bounds explicitly. Furthermore, CBMC is able to check whether an explicitly specified bound is indeed an upper bound for the specific loop or recursive function call (see the discussion of unwinding assertions in Section 3.1). Since each program has to be bounded, CBMC can handle dynamic memory allocation in a straightforward way: it replaces each call to `malloc` or `calloc` by the address of a fresh variable of appropriate type and size [12].

Undefined Behavior. CBMC-GC introduces nondeterminism in the case of program behavior that is undefined with respect to the ANSI C standard, e.g., access to nonallocated memory. This nondeterminism is introduced by adding additional input pins to the circuit whose values determine the outcome of the nondeterministic choice. CBMC-GC detects such inputs and alarms the user. Furthermore, CBMC itself can be used to show the absence of such behaviors. For example, CBMC includes checks for index-out-of-bounds accesses in arrays. In case such an array access happens, CBMC provides an execution trace which leads to this access and helps to efficiently debug the C program. Note that for the bounded and moderately sized programs considered in STC, CBMC can detect all such violations.

Floating-Point Computations. We extended CBMC version 2.4 for the implementation of CBMC-GC. The publicly available source code of CBMC does not include support for floating-point computations (only fixed-point computations). Therefore, CBMC-GC does not support this feature. However, the closed-source version of CBMC supports floating-point arithmetic. Translating C programs with floating-point computations into circuits is not a principal hurdle.

Pointer Arithmetic. At the moment, the support of pointer arithmetic in CBMC-GC is limited: Only pointer arithmetic involving addresses of variables and constants, e.g., `*(array + 5)`, is supported. Internally, we translate accesses to constant-size arrays, like `array[i]`, into a nested *if*-structure. We can therefore support accesses to arrays of constant size (see examples given in Section 4). CBMC itself supports full pointer arithmetic but produces circuits for these operations which are equisatisfiable (when translated to an SAT formula) to the actual computation, but do not encode the computation directly (the circuits involve nondeterminism which is resolved by the SAT solver). Equisatisfiability of these circuits is sufficient for software verification but not suitable for STC. We plan to adapt the corresponding circuit generation part of CBMC and, then, will support full pointer arithmetic involving statements like `*(array + i)` in a future release of CBMC-GC.

Data Types. CBMC-GC supports 16-, 32-, and 64-bits as the size of the data type `int`. In future work, we plan to make the size of integer variables completely customizable (the restricted set of available sizes of the integer data type is inherited from unmodified CBMC). Note that CBMC supports the C99 `_Bool` data type representing a single bit; thus, using arrays of type `_Bool` one can already simulate differently sized integer data types.

4. EFFICIENCY ANALYSIS

In order to test the performance of CBMC-GC, in particular the size of the obtained circuits, we performed several experiments on code fragments of increasing complexity; the results are reported in the rest of the paper.[4]

4.1 Experimental Setup

In the evaluation we focus on the complexity of the circuits generated by CBMC-GC. The time required to compile C code to a circuit ranges between a few seconds and a couple of minutes, and is thus well within the range of modern compilers; furthermore, these computations are typically not time critical, since they need to be done only once (and offline).

Besides the size of the resulting circuits, we also report realistic estimates on the resources required to evaluate the circuits in a framework for secure computation. We therefore run the circuits through the framework of [20], which is one of the fastest known implementation of garbled circuits. In a nutshell, the framework allows to test Secure Two-Party Computation in a realistic fashion. Both parties are implemented as individual Java programs running on two computers, communicating via a network connection.

To this end, both parties are given the functionality to be computed (i.e., the compiled CBMC-GC circuit). Subsequently, one party acts as server, whose purpose is to garble the circuit and initiate the transmission of the keys representing the inputs; the other party acts as client, which receives and evaluates the garbled circuit. This process involves two steps: Server and client first exchange the necessary keys corresponding to their inputs: for the inputs of the server, this can simply be done by transmitting the keys representing the inputs to the client; however, transmission of the keys representing the client inputs requires running Oblivious Transfer. Once the client has access to the required keys, the server starts garbling the circuit and transmitting the garbled gates on the fly to the client, who performs the evaluation. This process assures the efficiency of the overall circuit garbling and evaluation, as the entire garbled circuit does not need to be stored in memory.

We repeat all experiments in two different setups: in one case the two machines reside in local proximity, thus communication passes a Local Area Network (LAN). In the second case, the machines are located at two different institutions, requiring the communication to pass a Wide Area Network (WAN). All timing measurements presented below thus include network latency. For the LAN experiments, we use two desktop computers with two 3.2 GHz cores and 4 and 16 GB memory, respectively. On these machines we run OpenSUSE 11.1 Linux and Java 1.6. For our WAN experiments we used the 16 GB memory machine described above and a four core 2.33 GHz machine with 16 GB memory running Debian 5.0.9 Linux and Java 1.6.

Table 1 depicts our results. For each test case described in the sequel, it shows the number of gates of the resulting circuit, the number of gates other than XOR, and the execution time in milliseconds within the framework of [20] in the LAN and WAN environments. In the measurements, we differentiate between the above mentioned two steps, namely the time it takes to prepare the inputs of the circuit (i.e., transmission of the necessary keys and running the OT protocol) and the processing time for garbling and evaluating the circuit. We disregard other operations (such as setting up the test environment), as they take constant time. We can generally observe that the execution times in the WAN are slightly higher due to the network latency.

4.2 Arithmetic Computations

Functions based on arithmetic operations belong to the most basic set of operations that needs to be supported efficiently by a framework for secure computations. The first set of tests thus concerns programs that predominantly consist of additions and multiplications of integers; we will always use 32 bit integer values.

[4]CBMC-GC is available at `http://www.forsyte.at/software/cbmc-gc/`.

Experiment	Number of gates		LAN experiment (ms)		WAN experiment (ms)	
	Total	Non-XOR	Preparation	Evaluation	Preparation	Evaluation
Addition	161	32	654	8	870	9
Multiplication	6,223	1,741	673	127	796	175
100 Arithmetic operations	76,621	20,290	729	469	982	849
1000 Arithmetic operations	765,561	202,772	714	2,715	1,376	4,447
2000 Arithmetic operations	1,531,601	405,640	604	6,275	855	6,983
3000 Arithmetic operations	2,298,441	608,668	970	9,774	942	11,037
3×3 matrix multiplication	170,875	47,583	705	838	771	1,088
5×5 matrix multiplication	793,751	220,825	717	2,702	1,001	4,243
8×8 matrix multiplication	3,257,345	905,728	680	18,173	835	21,008
Comparison	234	65	686	15	782	16
Median, bubble sort, 11 elements	18,030	5,440	639	279	683	346
Median, merge sort, 11 elements	111,339	35,776	598	519	673	644
Median, bubble sort, 21 elements	67,710	20,480	637	611	719	812
Median, merge sort, 21 elements	541,669	175,936	635	1,689	823	2,810
Median, bubble sort, 31 elements	149,040	45,120	733	1,644	993	2,287
Median, merge sort, 31 elements	1,339,084	436,916	660	3,790	1,112	4,794
Median, bubble sort, 41 elements	262,020	79,360	899	1,477	803	2,240
Median, bubble sort, 51 elements	406,650	123,200	623	2,205	1,018	3,221
Median, bubble sort, 61 elements	582,930	176,640	646	3,003	840	4,108
Median, bubble sort, 71 elements	790,860	239,680	675	4,557	712	6,127
Median, bubble sort, 81 elements	1,030,440	312,320	858	5,467	828	6,966
Median, bubble sort, 91 elements	1,301,670	394,560	660	8,025	719	9,262
Hamming distance, 160 bit	9,436	3,003	707	86	712	104
Hamming distance, 320 bit	19,031	6,038	735	115	1,095	143
Hamming distance, 800 bit	47,816	15,143	788	145	774	215
Hamming distance, 1600 bit	95,791	30,318	746	291	912	353
Interpreter, 1 gate	12,844	4,196	649	219	744	324
Interpreter, 2 gates	26,079	8,520	666	367	809	648
Interpreter, 5 gates	68,136	22,260	639	620	674	802
Interpreter, 10 gates	146,071	47,720	724	995	675	1,345
Interpreter, 15 gates	233,806	76,380	614	1,279	807	1,969
Interpreter, 50 gates	1,122,351	366,600	1,150	6,267	884	7,799
MBI(10, 19, 16)*	59,027	19,342	666	539	733	914
MBI(10, 26, 16)	83,891	27,490	663	723	725	932
MBI(10, 35, 32)	115,947	38,006	677	1,084	734	1,472
MBI(50, 19, 16)	303,027	99,302	690	1,517	830	1,935
MBI(50, 26, 16)	428,691	140,490	653	2,170	789	3,054
MBI(50, 35, 32)	598,987	196,366	700	3,039	913	3,750
MBI(100, 19, 16)	608,027	199,252	688	2,934	802	3,975
MBI(100, 26, 16)	859,691	281,740	646	4,504	875	5,496
MBI(100, 35, 32)	1,202,787	394,316	1,017	6,918	845	8,205

*) MBI(g, m, i): Memory bounded interpreter with g gates and m bits memory where i bits are initialized by input.

Table 1: Experimental results: circuit sizes (number of gates and number of gates other than XORs) and timing results for the preparation and evaluation phases (in ms).

First we consider the complexity of one addition and one multiplication operation, see Table 1. While specifying one addition or multiplication of two unsigned integers can be done by a single statement in C, the corresponding addition circuit already consists of $5 \cdot 32 = 160$ gates, of which 32 are non-XOR gates; a multiplication even requires $6,223$ gates where $1,741$ are non-XOR gates. While the number of non-XOR gates is optimal for the addition (e.g. compared to a hand-optimized circuit), it is possible to use better optimized versions for the multiplication. For example, [19] reports a multiplication circuit consisting of $1,729$ non-XOR gates implementing a method by Karatsuba [22]. Since many arithmetic operations can be expressed in one single line of a C program, the circuits corresponding to even small programs can be huge.

Circuits for programs that perform several arithmetic operations sequentially essentially scale linear in the number of operations. To test this, we created programs containing up to 3000 random arithmetic operations, comprising of 90% additions and 10% multiplications with varying numbers of input and output variables. Table 1 depicts the results. The largest circuit consists of more than 2.2 million gates; still, the time it takes to evaluate the circuit (approximately 13 seconds) is rather moderate. This shows that common medium-size programs consisting of arithmetic operations—which occur, for example, in standard spreadsheet computations—can be implemented in practical applications with acceptable performance.

As a second test case, we consider the problem of implementing the $S \times S$ matrix multiplication obliviously; the CBMC-GC implementation is shown in Figure 1. Compiling the program using CBMC-GC for 3×3 matrices results in a circuit with 170,875 gates, while 8×8 matrices already require more than 3.2 million gates; still, an 8×8 multiplication requires only 23 seconds to finish. Again, runtimes of the program for different values of S are given in Table 1.

```
#define S 2 // size of matrices
int INPUT_A_a[S][S];
int INPUT_B_b[S][S];
int OUTPUT_c[S][S];

void multiply()
{
  int i, j, k;

  for (i = 0; i < S; i++)
    for (j = 0; j < S; j++)
      for (k = 0; k < S; k++)
        OUTPUT_c[i][j] += INPUT_A_a[i][k] * INPUT_B_b[k][j];
}
```

Figure 1: CBMC-GC program for matrix multiplication.

4.3 Bit Operations and Comparisons

Most programs that are of interest in the context of secure computations rely heavily on bit operations and comparisons. Probably the most classic example of a secure computation is the solution to Yao's Millionaires Problem, which consists of a single integer comparison operation. The straightforward CBMC-GC implementation depicted in Section 3.2 yields to a circuit of 234 gates, of which 65 are non-XOR gates. This is roughly two times the number of gates that a hand-optimized circuit would require. The time required to garble and evaluate the comparison circuit amounts to only 15ms (albeit with a rather high overhead to prepare the input).

In order to test the scalability of problems that require a large number of successive dependent comparisons, we consider the problem of obliviously sorting an array and picking the median element. While this application has been considered in the past mainly by implementing sorting networks, it can be solved by CBMC-GC in a surprisingly simple manner by implementing any standard sorting algorithm and accessing the median element. Figure 2 depicts two different CBMC-GC implementations, using either Bubble Sort or Merge Sort. The latter is even a recursive implementation, which shows the power of CBMC. In fact, due to the constant size of the input array, CBMC is able to statically determine the maximum number of recursive calls, which allows a successful unwinding of the recursive program. Interestingly, while the computational complexity of Merge Sort is theoretically optimal, it yields to significantly

```
#define K 11 // length of array
#define MEDIAN 5 // position of median

int INPUT_A_a[K];
int OUTPUT_median;

void median_bubblesort() {
  int i, j, tmp, tmp1, tmp2;

  for (i = K - 1; i > 0; i--) {
    for (j = 0; j < i; j++) {
      tmp1 = INPUT_A_a[j];
      tmp2 = INPUT_A_a[j + 1];
      if (tmp1 > tmp2) {
        INPUT_A_a[j] = tmp2;
        INPUT_A_a[j + 1] = tmp1;
      }
    }
  }

  OUTPUT_median = INPUT_A_a[MEDIAN];
}

int b[K]; // temporary array for mergesort

void mergesort(int l, int r) {
  int i, j, k, m;
  if (r > l) {
    m = (r + l)/2;
    mergesort(l, m);
    mergesort(m + 1, r);
    for (i = m + 1; i > l; i--)
      b[i - 1] = INPUT_A_a[i - 1];
    for (j = m; j < r; j++)
      b[r + m - j] = INPUT_A_a[j + 1];
    for (k = l; k <= r; k++) {
      if (b[i] < b[j])
        INPUT_A_a[k] = b[i]; i++;
      else
        INPUT_A_a[k] = b[j]; j--;
    }
  }
}

int median_mergesort() {
  mergesort(0, K - 1);

  return INPUT_A_a[MEDIAN];
}
```

Figure 2: CBMC-GC program for computing the median element of an array in two ways: using either Bubble sort or merge sort.

slower implementations in CBMC-GC. The reason for this behavior is again the large number of array accesses, where the element accessed is determined by a variable computed at runtime. These accesses are treated by CBMC-GC as MUX circuits; this leads to a significantly larger circuit and hence a larger processing time.

Finally, we consider the problem of computing the Hamming distance between two binary vectors; Figure 3 depicts a CBMC-GC implementation computing the Hamming distance of two vectors of length $32 \cdot K$. For two bit vectors of length 160 bit, the fastest known hand-coded implementation in [20] requires about 2,798 gates. Translating the code of Figure 3 automatically results in a circuit with 9,436 gates which corresponds in an overhead of factor 3.5.

5. PRIVATE FUNCTIONS

Traditional Secure Two-Party Computation assumes that both parties are aware of the program they compute and only want to hide the inputs from each other. However,

```
#define K 5
int INPUT_A_a[K];
int INPUT_B_b[K];
int OUTPUT_hd;

int popcount32(unsigned int y) {
    int x = y - ((y >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = (x + (x >> 4)) & 0x0f0f0f0f;
    x += x >>  8;
    x += x >> 16;

    return x & 0x7f;
}

void hammingdistance(void) {
  int tmp = 0;

  for (int i = 0; i < K; i++)
     tmp += popcount32(a[i] ^ b[i]);

  OUTPUT_hd = tmp;
}
```

Figure 3: CBMC-GC program for computing the Hamming distance between two bit arrays.

in some scenarios it is desirable to consider an asymmetric situation, where only one party holds the function to be evaluated, while the second party holds the respective data on which the function should be applied. For example, this is relevant once algorithms must be evaluated which are considered the intellectual property of one of the parties. In this case, the parties can jointly evaluate the algorithm without revealing the algorithm as well as the private input.

Two different approaches to this problem were reported in the literature. The first solution is based on the simple observation that code and data can be interchanged by the use of universal programs (or interpreters). More precisely, instead of evaluating the desired circuit directly, both parties run a universal circuit that gets as input both a description of the desired program (again in terms of a circuit), held held by one party, and the input, held by the other. Given that the size of the program to be executed is known to both parties, the garbled circuit technique can be applied to the universal circuit in a straightforward manner. The approach using universal circuits has been described in detail in [25]. Given a secret program consisting of n Boolean gates, the asymptotic complexity of the universal circuit construction is $O(n \log^2(n))$. A second solution to this problem has been proposed in [23]. Encrypting the circuit directly, while simultaneously hiding the circuit topology, they achieve an asymptotic complexity of $O(n)$. However, in order to garble the entire circuit a large number of public key operations is necessary. Since the size of the circuits can become rather large, the required computational complexity is likely to exceed typical computing resources (unfortunately, no implementation is presented in [23] to experimentally verify this assumption).

Solving the problem of STC with private functions is surprisingly simple with CBMC-GC. In fact, one can write a simple interpreter in C, which interprets a binary circuit of a given size. In the sequel we describe two such interpreters with different properties.

The first interpreter is depicted in Figure 4. The interpreter takes as input a list of gates, where each gate is encoded as follows: Two bits are used to represent the binary operation (AND, OR, XOR and NOT) to be performed

```
#define CIRCUIT_SIZE 50
#define INPUT_SIZE 32

void interpreter_1() {
  _Bool INPUT_A_a[INPUT_SIZE];
  _Bool INPUT_B_b[INPUT_SIZE];

  _Bool INPUT_A_circuit_ops[2 * CIRCUIT_SIZE];
  short INPUT_A_circuit_left[CIRCUIT_SIZE];
  short INPUT_A_circuit_right[CIRCUIT_SIZE];

  _Bool results[CIRCUIT_SIZE + 2 * INPUT_SIZE];

  int i;

  for (i = 0; i < INPUT_SIZE, i++)
    results[i] = INPUT_A_a[i];
  for (i = INPUT_SIZE; i < 2 * INPUT_SIZE; i++)
    results[i] = INPUT_B_b[i];
  for (i = 2 * INPUT_SIZE; i < CIRCUIT_SIZE; i++)
    results[i] = 0;

  for (i = 0; i < CIRCUIT_SIZE; i++) {
   short left_index = INPUT_A_circuit_left[i];
   short right_index = INPUT_A_circuit_right[i];
   _Bool left_value = results[left_index];
   _Bool right_value = results[right_index];
   _Bool tmp = 0;

   if (INPUT_A_circuit_ops[2 * i] == 0) {
     if (INPUT_A_circuit_ops[2 * i + 1] == 0)
       tmp = left_value & right_value;  // AND
     else
       tmp = left_value | right_value;  // OR
   } else {
     if (INPUT_A_circuit_ops[2 * i + 1] == 1)
       tmp = !left_value; // NOT
     else
       tmp = left_value ^ right_value; // XOR
   }

   results[2 * INPUT_SIZE - 1 + i] = tmp;
  }

  _Bool OUTPUT_result =
    results[2 * INPUT_SIZE - 1 + CIRCUIT_SIZE];
}
```

Figure 4: CBMC-GC code for interpreting arbitrary circuits.

while two additional integer values represent the indices of the operands. The result of each gate evaluation is stored in a separate array `results`. Now, evaluating a gate is done in three steps: First the opcodes and operands are fetched (observe that all operands originate from the parties inputs or previous gate evaluations, and thus can be loaded from the array `results`). Next, the operation is performed. In a final step the result is written to `results`. Once all gates are evaluated, the interpreter outputs the result of the computation. For simplicity of presentation we assume that the circuit always outputs a single bit and that the result is computed by the last gate in the list. However, more complex scenarios with multiple output bits can be implemented with ease as well. The design of the interpreter allows arbitrary programs to be interpreted, at the cost of an asymptotic complexity of $O(n^2)$.

Using C as programming language one can easily develop customized versions of interpreters. For example, in case it is known that interpreting the circuit needs only memory significantly smaller than n, a different interpreter, depicted in Figure 5, can be utilized. This interpreter makes use of a small constant scratch memory, holding the intermediate results (just like a standard RAM). This memory initially

```
#define MEMORY_SIZE 3
#define INPUT_SIZE 32
#define CIRCUIT_SIZE 100

_Bool mem[INPUT_SIZE + MEMORY_SIZE];

typedef struct {
  _Bool opcode1;
  _Bool opcode2;
  short leftOperand;
  short rightOperand;
  short target;
} operation;

operation INPUT_A_ops[CIRCUIT_SIZE];
_Bool INPUT_B_mem[INPUT_SIZE];

void interpreter_2() {
  unsigned i = 0;
  _Bool opcode1 = 0, opcode2 = 0, result = 0;
  short lOp = 0, rOp = 0, target = 0;
  _Bool lOpValue = 0, rOpValue = 0;

  for (i = 0; i < INPUT_SIZE; i++)
    mem[i] = INPUT_B_mem[i];
  for (i = INPUT_SIZE; i < INPUT_SIZE + MEMORY_SIZE; i++)
    mem[i] = 0;

  for (i = 0; i < CIRCUIT_SIZE; i++) {
    opcode1 = INPUT_A_ops[i].opcode1;
    opcode2 = INPUT_A_ops[i].opcode2;
    lOpValue = mem[INPUT_A_ops[i].leftOperand];
    rOpValue = mem[INPUT_A_ops[i].rightOperand];
    target = INPUT_A_ops[i].target;

    if (opcode1) {
      if (opcode2)
        result = lOpValue & rOpValue;
      else
        result = lOpValue | rOpValue;
    } else {
      if (opcode2)
        result = lOpValue ^ rOpValue;
      else
        result = !lOpValue;
    }

    mem[target] = result;
  }

  _Bool OUTPUTmem[MEMORY_SIZE];

  for (i = 0; i < MEMORY_SIZE; i++)
    OUTPUTmem[i] = mem[INPUT_SIZE + i];
}
```

Figure 5: CBMC-GC code for interpreting circuits with bounded memory.

holds the parties' input values and is later updated to store intermediate results occuring during the computation. To implement the interpreter it now suffices to iterate over all gates, evaluating one gate at a time: reading the required inputs from scratch memory, performing the binary operation and writing the result back to a specific location in the constant size memory.

For both approaches, the interpreter must be tailored towards the size of the program (circuit) to be executed: once the number of input bits and the number of gates of the input program are known, the constants in Figures 4 and 5 can be set accordingly. Subsequently, CBMC-GC can compile the interpreter into a circuit, which can be used in STC to evaluate *any* circuit of the given size (with the constraint that the size of the memory can be set to a small constant, independent of the circuit size, for the second interpreter). In fact, CBMC-GC distills a universal circuit out of the first interpreter. Note that the input to both interpreters is itself a circuit, which can again be constructed out of a C program using CBMC-GC.

Table 1 again depicts the experimental results; when interpreting 50 gates, the first interpreter is compiled to a circuit with approximately 1.2 million gates, among them $366,600$ gates other than XOR. Note that the size of the compiled interpreter scales with $O(n^2)$, due to the way of handling arrays and the representation of the circuit. The second interpreter uses a small fixed memory: e.g. to interpret circuits with 50 gates which require a memory of 35 bits, CBMC-GC produces a circuit with about $600,000$ gates, of which $196,366$ are non-XOR gates.

6. CONCLUSIONS & FUTURE WORK

This paper demonstrated that secure two-party computation can be realized for ANSI C programs. We believe that this is an important step to put secure computation to wide practical use. Our experiments show that the tool chain has good practical performance. We expect that the separation of concerns between garbled circuit evaluation and C compilation will enable the programming language and the compiler communities to further improve the performance of practical secure computation. Our work therefore facilitates and encourages a closer interaction between programming languages and STC.

To realize the C compiler, our paper reused the CBMC tool chain originally developed for software model checking. This has the added advantage of a tight coupling between the tools for STC and verification of code correctness. In future work, we plan to further investigate this relationship as to achieve both functional correctness and security with a single tool, and to directly verify the circuits. To support different programming languages one might also consider LLVM[5] bytecode as source language for circuit synthesis.

In future releases of CBMC-GC we will provide full support for pointer arithmetic and floating point arithmetic as discussed in Section 3.2.6. Moreover, we will also provide support for the integration of CBMC-GC with other STC implementations such as [28] and will define interfaces that enable secure computations in the context of larger programs.

7. ACKNOWLEDGMENTS

This work was supported in part by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF), by the Vienna Science and Technology Fund (WWTF) grant PROSEED, and by CASED.

8. REFERENCES

[1] VIFF, the Virtual Ideal Functionality Framework. `http://viff.dk/`.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 203–213. ACM Press, 2001.

[5] `http://llvm.org/`

[3] M. Barni, P. Failla, V. Kolesnikov, R. Lazzeretti, A.-R. Sadeghi, and T. Schneider. Secure Evaluation of Private Linear Branching Programs with Medical Applications. In *Proceedings of the 14th European Conference on Research in Computer Security*, ESORICS'09, pages 424–439. Springer, 2009.

[4] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: A System for Secure Multi-Party Computation. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 17–21. ACM, 2008.

[5] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, October 2007.

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207. Springer, 1999.

[7] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications.* IOS Press, 2009.

[8] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 192–206. Springer, 2008.

[9] P. Bogetoft, I. Damgård, T. Jakobsen, K. Nielsen, J. Pagter, and T. Toft. A Practical Implementation of Secure Auctions Based on Multiparty Integer Computation. In *Proceedings of the 10th International Conference on Financial Cryptography and Data Security*, FC '06, pages 142–147. Springer, 2006.

[10] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '04, pages 168–176. Springer, 2004.

[11] E. Clarke, D. Kroening, and K. Yorav. Behavioral Consistency of C and Verilog Programs using Bounded Model Checking. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pages 368–371. ACM, 2003.

[12] E. Clarke, D. Kroening, and K. Yorav. Behavioral Consistency of C and Verilog Programs using Bounded Model Checking. Technical Report CMU-CS-03-126, Carnegie Mellon University, School of Computer Science, 2003.

[13] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV '00, pages 154–169. Springer, 2000.

[14] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *Proceedings of the 14th European Conference on Programming Languages and Systems*, ESOP '05, pages 21–30. Springer, 2005.

[15] R. Cramer, I. Damgård, and J. B. Nielsen. Multiparty Computation from Threshold Homomorphic Encryption. In *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques: Advances in Cryptology*, EUROCRYPT '01, pages 280–299, 2001.

[16] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. Privacy-Preserving Face Recognition. In *Proceedings of the 9th International Symposium on Privacy Enhancing Technologies*, PETS '09, pages 235–253. Springer, 2009.

[17] M. K. Ganai, A. Gupta, and P. Ashar. *DiVer*: SAT-Based Model Checking Platform for Verifying Large Scale Systems. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 575–580. Springer, 2005.

[18] B. Goethals, S. Laur, H. Lipmaa, and T. Mielikäinen. On Private Scalar Product Computation for Privacy-Preserving Data Mining. In *Proceedings of the 7th International Conference on Information Security and Cryptology*, ICISC'04, pages 104–120. Springer, 2004.

[19] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-partY computations. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 451–462. ACM, 2010.

[20] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *Proceedings of the 20th USENIX Security Symposium*, USENIX '11, 2011.

[21] G. Jagannathan and R. N. Wright. Privacy-Preserving Distributed k-Means Clustering over Arbitrarily Partitioned Data. In *Proceedings of the eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, pages 593–599. ACM, 2005.

[22] A. Karatsuba and Y. Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. In *Doklady Akad. Nauk SSSR, Vol. 145, Translation in Physics-Doklady, 7 (1963), pp. 595–596*, 1962.

[23] J. Katz and L. Malka. Constant-Round Private Function Evaluation with Linear Complexity. In *Proceedings of the 17th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '11, pages 556–571. Springer, 2011.

[24] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved Garbled Circuit Building Blocks and Applications to Auctions and Computing Minima. In *Proceedings of the 8th International Conference on Cryptology and Network Security*, CANS '09, pages 1–20. Springer, 2009.

[25] V. Kolesnikov and T. Schneider. A Practical Universal Circuit Construction and Secure Evaluation of Private Functions. In *Proceedings of the 12th International Conference on Financial Cryptography and Data Security*, FC '08, pages 83–97. Springer, 2008.

[26] V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *Proceedings of the 35th International Colloquium on*

Automata, Languages and Programming, Part II, ICALP '08, pages 486–498. Springer, 2008.

[27] Y. Lindell and B. Pinkas. A Proof of Security of Yao's Protocol for Two-Party Computation. *Journal of Cryptology*, 22:161–188, April 2009.

[28] L. Malka. VMCrypt: Modular Software Architecture for Scalable Secure Computation. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 715–724. ACM, 2011.

[29] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — A Secure Two-Party Computation System. In *Proceedings of the 13th Conference on USENIX Security Symposium*, SSYM'04, pages 20–20. USENIX Association, 2004.

[30] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure Two-Party Computation Is Practical. In *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '09, pages 250–267. Springer, 2009.

[31] A. C.-C. Yao. Protocols for Secure Computations (Extended Abstract). In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, FOCS '82, pages 160–164. IEEE Computer Society, 1982.

[32] A. C.-C. Yao. How to Generate and Exchange Secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, FOCS '86, pages 162–167. IEEE Computer Society, 1986.