Proving Correctness and Security of Two-Party Computation Implemented in Java in Presence of a Semi-Honest Sender*

Florian Böhl[1], Simon Greiner[1], and Patrik Scheidecker[2]

[1] firstname.lastname@kit.edu
Karlsruhe Institute of Technology
Institute of Theoretical Informatics
[2] lastname@fzi.de
FZI Research Center for Information Technology

Abstract. We provide a proof of correctness and security of a two-party-computation protocol based on garbled circuits and oblivious transfer in the presence of a semi-honest sender. To achieve this we are the first to combine a machine-assisted proof of correctness with advanced cryptographic primitives to prove security properties of Java code. The machine-assisted part of the proof is conducted with KeY, an interactive theorem prover.

The proof includes a correctness result for the construction and evaluation of garbled circuits. This is particularly interesting since checking such an implementation by hand would be very tedious and error-prone. Although we stick to the secure two-party-computation of an n-bit AND in this paper, our approach is modular, and we explain how our techniques can be applied to other functions.

To prove the security of the protocol for an honest-but-curious sender and an honest receiver, we use the framework presented by Küsters et al. for the cryptographic verification of Java programs. As part of our work, we add oblivious transfer to the set of cryptographic primitives supported by the framework. This is a general contribution beyond our results for concrete Java code.

1 Introduction

Motivation and overview Protocols for secure two-party computation allow two parties to evaluate a function f such that both parties provide a part of the input. Neither of the parties must learn more about the input of the other than can be inferred by its own input, the function f and the computed output. Since first solutions for two-party computation protocols have been presented by Yao [33, 34], the problem has received a lot of attention (e.g., [12, 7, 23, 24, 27]).

Yao's approach Following the initial ideas of Yao, one can construct a two-party computation protocol from garbled circuits [33, 34] and oblivious transfer [30] (see Figure 1). The basic idea is to first encode the function f as a circuit consisting of gates and wires. Such a circuit can then be transformed into a garbled circuit by one of the parties, say the sender S. Instead of a bitstring, the garbled circuit takes an encoding for each bit as input. These encodings are initially only known to the creator of the garbled circuit (S here). When the receiver R wants to evaluate the garbled circuit on a given input $x \in \{0,1\}^n$, it needs to know the corresponding encoding for each input bit.

The encodings for the input bits of R are transmitted via oblivious transfer from S to R. The oblivious transfer protocol guarantees that R learns exactly one encoding for each of its own input bits and that S remains oblivious to which encodings R learned. Subsequently, S transmits the garbled circuit and the encodings of its own input bits to R. These encodings don't tell R anything about S's input bits. Finally, R can evaluate the garbled circuit. Note that R only knows the corresponding encoding for one bitstring $x \in \{0,1\}^n$ and hence can only use the garbled circuit to compute $f(x)$.

From theory to practice While they have always been of theoretical interest, two-party computation protocols seemed far away from being applicable to practical problems for a long time. Beginning with Fairplay [26], methods to construct (garbled) circuits for generic functions have drastically improved (e.g., [18, 14]). This inspired various protocols for practical problems [17, 10, 28, 15]. As performance of garbled circuits is going to increase, we are going to see more practical applications of garbled circuits in the future.

* This work was partially funded by the KASTEL project by the Federal Ministry of Education and Research, BMBF 01BY1172. Florian Böhl was supported by MWK grant "MoSeS".

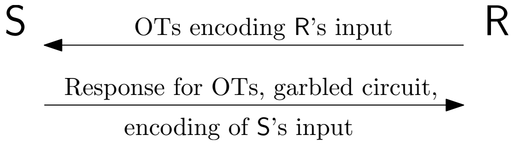

Fig. 1. Yao's protocol for two-party computations.

Mind the gap Although all of those protocols come equipped with security proofs of abstractions of the protocol, there remains a gap between the security of the specification in a theoretical model and a real world implementation, e.g., in Java. There are aspects of actual implementations, which have no counterpart in the abstract world. For example, even if a protocol itself is secure, minor mistakes in its realization can completely break security as the recent Heartbleed bug in the OpenSSL library shows. Under this point of view it is important not only to prove the security and correctness of a protocol in the abstract world but also to verify its actual implementation. This can be achieved by using machine-based verification techniques, and in this work we present a first step to close this gap. We chose Java for being a widely used programming language in the real world, unlike e.g. EasyCrypt [2], which offers verification in its own specific language.

Our contributions We use the KeY tool[3], a deductive verification tool for Java programs, to show the correctness of a two-party computation protocol implemented in Java. KeY was previously used for verification of functional properties[31], non-interference properties[13], and security properties of programs making use of public key encryption[20]. Our first contribution is to extend this body of work by using KeY to prove correctness of an implementation that uses symmetric encryption for garbled circuits. The machine-assisted proof is done for a concrete function f, namely an n-bit AND. While this might seem limited at first glance, the proof is modular, i.e., it uses the correctness of the implementation of garbled gates in a black-box way. Additionally, we explain how correctness proofs for other functions can be conducted in the same fashion. As a proof of concept, we also prove the correctness of a XOR gate. The correctness of our implementation of an n-bit AND can be used in a black-box way to show the correctness of more complicated circuits. This is the first paper that presents a security and correctness proof of an implementation in a real-world programming language. Alternative tools for languages like Java, C, and C#, are, for example, Spec# [1], Krakatoa [11] or VCC [32]. Another tool using symbolic execution for Java and C programs is the VeriFast system [16].

Our second contribution is to show the security of the implementation in presence of a passive adversary for corrupted sender S and honest receiver R; independently of the function f. To achieve this, we add oblivious transfer as a cryptographic building block to the framework for Cryptographic Verification of Java-like programs by Küsters et al. [19, 21]. That is, we provide an ideal interface in Java for oblivious transfer following the ideal OT functionality of [7] for Canetti's UC-framework [5]. We show that this ideal interface can be implemented by any UC-secure oblivious transfer protocol (e.g., [29, 9, 8]).

On our restrictions. We would like to point out that – although we only consider honest-but-curious security in this paper – our work is a necessary step towards proving security of implementations of adaptively secure protocols based on garbled circuits (like [25] for example). Every security argument for these protocols assumes that the actual implementation is correct; this is where our result is needed. Furthermore, we would like to point out that it is sufficient that the output is only learned by R as explained in [24].

Outline The structure of this paper is as follows. In the next section we introduce some preliminaries, in particular we introduce the framework for the Cryptographic Verification of Java Programs by Küsters et al. [19]. Furthermore, we briefly introduce cryptographic building blocks used in this paper, the specification language *Java Modelling Language* and the interactive theorem prover *KeY*.

Subsequently, in Section 3, we describe the protocol we analyze in this work. In Section 4 we present details on an abstraction for the secret key encryption scheme, followed by a description of the modular implementation for the cryptographic building blocks introduced earlier. We then show two lemmas stating the correctness of the implementation with respect to the specification.

Finally, we prove the security of our protocol for a semi-honest sender S (i.e., in presence of a passive adversary) in Section 5 using the results from Section 4. We conclude in Section 6 and present future work.

2 Preliminaries

2.1 The CVJ framework

In this section we briefly review the parts of the CVJ (Cryptographic Verification of Java Programs) framework by Küsters et al. required for our results. We refer the reader to [19] for a full description of the framework. The CVJ framework is formulated for the language Jinja+ which comprises a rich fragment of Java. Our implementation is not only a Java program, but a Jinja+ program. Hence we can make use of the framework.

Systems, runs, interfaces, and environments A Jinja+ *program* or *system* is a set of class declarations, just like common Java code restricted to the syntax of Jinja+. Programs may call a function `randomBit`. They are called *randomized* if they do and *deterministic* otherwise. An *interface* is a system without method bodies and static field initializers (see Figure 12 for example). Dropping all method bodies and field initializers from a system S yields an interface I that we consequently call *the interface of* S. If we additionally drop all private fields and methods from I, we call the result the *public interface of* S. Our concept of interfaces is not to confuse with the interface key word in Java. Basically, interfaces will be used to define what parts of a system other systems may utilize. For two interfaces I and I' we say that I' is a *subinterface* of I if it can be obtained from I by dropping whole classes (with their method and field declarations), dropping methods and fields, dropping `extends` clauses, and/or adding the `final` modifier to class declarations. Two interfaces are called *disjoint* if the set of class names declared in these interfaces are disjoint. We call them *compatible* if there exists an interface I they are both subinterfaces of.

A system S *implements* an interface I if I is a subinterface of the public interface of S. A system *uses* an interface I if, besides its own classes S uses at most classes, methods, and fields declared in I. We say that a system is *complete* if it uses the empty interface, i.e., the system does not have any external dependencies. We say that two systems are *composable* if they use compatible interfaces (see [19] for details).

An *environment* is a program that declares a private static variable `result` of type `boolean` with initial value `false`. We implicitly assume throughout the paper that the variable `result` is unique in every program and is always declared by what is the environment for a given context. For an interface I an environment E is called I*-environment* for a program S if

1. S implements I and E uses I,
2. there is an interface I_E such that E implements I_E and S uses I_E,
3. and either S or E contains `main`.

Then E and S are composable and $E \cdot S$ is a complete program. For finite runs of $E \cdot S$ the value of `result` at the end of the run is the *output of* $E \cdot S$. For infinite runs, we define the output to be `false`. If $E \cdot S$ is deterministic, we write $E \cdot S \rightsquigarrow \texttt{false}$ and $\mathsf{E} \cdot S \rightsquigarrow \texttt{true}$ for the two possible outputs respectively. For randomized programs we write $\mathbf{Pr}\left[E \cdot S \rightsquigarrow \texttt{false}\right]$ and $\mathbf{Pr}\left[E \cdot S \rightsquigarrow \texttt{true}\right]$ to denote the probabilities for the respective outputs. Finally, we say that two systems S_1 and S_2 *use the same interface* if S_1 uses I iff S_2 uses I for all interfaces I.

Indistinguishability of systems We quantify the run time of programs w.r.t. a given security parameter λ to use computational assumptions in a meaningful way. We say that

1. a program P is *almost bounded* if there exists a polynomial p such that the probability that a run of $P(\lambda)$ exceeds $p(\lambda)$ is negligible,
2. an environment E is *bounded* if there exists a polynomial p such that for every program P that is composable with E the number of steps performed in the code of E does not exceed $p(\lambda)$ for every run of $(E \cdot S)(\lambda)$, and
3. a system S is *environmentally* I*-bounded*, if S implements I and for each bounded I-environment of S, the program $E \cdot S$ is almost bounded.

Computational indistinguishability Let I be an interface and S_1 and S_2 be two environmentally I-bounded systems. Then S_1 and S_2 are *computationally indistinguishable* w.r.t. I, denoted $S_1 \overset{I}{\approx} S_2$, if S_1 and S_2 use the same interface and for every bounded I-environment E for S_1 and S_2 we have that $E \cdot S_1$ and $E \cdot S_2$ are computationally equivalent. That is, $\left|\mathbf{Pr}\left[(E \cdot S_1)(\lambda) \rightsquigarrow \texttt{true}\right] - \mathbf{Pr}\left[(E \cdot S_2)(\lambda) \rightsquigarrow \texttt{true}\right]\right|$ is a negligible function in the security parameter λ.

Definition 1 (Strong Simulatability). *Let $I_{out}, I_{in}, I_E, I_S$ be disjoint interfaces. Let F and R be systems. Then R realizes F w.r.t. the interfaces I_{out}, I_{in}, I_E, and I_S written $R \leq^{(I_{out}, I_{in}, I_E, I_S)} F$ if*

1. *R uses $I_{in} \cup I_E$ and implements I_{out},*
2. *F uses $I_{in} \cup I_E \cup I_S$ and implements I_{out},*
3. *either R and F both contain the `main` method, or none of them,*
4. *R is an environmentally I_{out}-bounded system, and*
5. *there is a system S (the simulator) such that*
   - *(a) S does not contain `main`*
   - *(b) S uses I_E and implements I_S*
   - *(c) $(S \cdot F)$ is environmentally I_{out}-bounded*
   - *(d) and $R \overset{I_{out}}{\approx} S \cdot F$*

Intuitively, I_{out} is the interface for the service provided by R and F. They may use trusted external services specified by I_{in} or untrusted external services I_E which will be implemented by the environment. I_S is the interface of the simulator used by the functionality F.

2.2 Cryptographic building blocks

SKE A *secret key encryption scheme* (SKE scheme) with keyspace $\mathcal{K}$ and message space $\mathcal{M}$ features three probabilistic-polynomial-time algorithms:

- Gen takes the security parameter λ and generates a key $k \in \mathcal{K}$,
- E takes a key $k \in \mathcal{K}$ and a message $M \in \mathcal{M}$ and outputs a ciphertext, and
- D takes a key $k \in \mathcal{K}$ and a ciphertext C and outputs the plaintext if decryption works and $\perp$ otherwise.

We say that an SKE scheme is *correct* if for all $k \in \mathcal{K}$ and all $M \in \mathcal{M}$ we have $\mathsf{D}(k, \mathsf{E}(k, M)) = M$. Furthermore, we stipulate for SKE schemes throughout the paper that $\mathbf{Pr}\left[\mathsf{D}(k', \mathsf{E}(k, M)) \neq \perp : k, k' \leftarrow \mathsf{Gen}(\lambda)\right]$ is negligible in λ. Note that this already implies that two honestly generated keys are equal only with negligible probability independent of the security of the SKE scheme.

```
public final class Key {
  public final int ident;
  public Key(int id);
}
public final class SKE {
  public static Key GenKey();
  public static Cipher Encrypt(Key k, Object m);
  public static Object Decrypt(Key key, Cipher c);
}
```

```
public final class Cipher {
 public final int ident;
 public Cipher(int id);
}
public final class GCnBitAND {
  public GCnBitAND(Key[] in0, Key[] in1,
                  Key out0, Key out1);
  public Key evaluate(Key[] in);
}
```

Fig. 2. Interfaces $\mathcal{I}_{\mathsf{SKE}}$ for secret key encryption and $\mathcal{I}_{\mathsf{GC}}$ for n-bit AND garbled circuit.

The algorithms Gen, E and D can be provided in many ways; in our implementation they are provided by the interface $\mathcal{I}_{\mathsf{SKE}}$ as shown in Figure 2. The methods `GenKey`, `Encrypt` and `Decrypt` provide the respective functionality. The classes `Key` and `Cipher` provide a constructor, which we do not further specify and the identifier `ident` which represents the numerical representation of a byte array. An implementation can use the `ident` field to store an arbitrary representation of the object. We use this concept throughout the paper for all abstract objects we have. Although identifiers are defined here as of data type *int*, these fields can hold arbitrary natural numbers during verification. Hence, they are merely a placeholder and an actual representation would not be bounded by a 32 or 64 bit integer size. The method `GenKey` creates a key which is unique but not efficiently distinguishable from a random number.

The `Encrypt` method provides encryption functionality. An object is encrypted with a key into a cipher. The cipher returned has an identifier, which is also indistinguishable from a random number for S and R.

The `Decrypt` method takes as input a cipher and a key. If the key has the same identifier as the key originally used for encryption, the encrypted object is returned. If the provided key has a different identifier, the method returns `null`.

Next, we briefly introduce circuits and garbled circuits as used throughout this paper. A thorough and comprehensive state-of-the-art description can be found in [4].

Circuits A *circuit* consists of input pins, output pins, gates and wires. Each gate has two input pins and one output pin. Each wire connects exactly two pins and each pin is connected to exactly one wire. Furthermore, each wire connects an input pin of the circuit to an input pin of a gate. An output pin of a gate is connected to an input pin of a gate or an output pin of the circuit. If the gates, the input pins, and the output pins of the circuits are viewed as nodes of a graph and the wires are viewed as the edges, a circuit must be a directed acyclic graph. Each wire can take a value from $\{0, 1\}$ and each gate resembles an arbitrary binary function $g : \{0, 1\}^2 \to \{0, 1\}$. [3] To *evaluate* a circuit having n input wires for an input $x \in \{0, 1\}^n$, we assign x_i to the wire connected to the ith input pin and then evaluate gate after gate in a straightforward way. Obviously, for every function $f : \{0, 1\}^n \to \{0, 1\}^m$ we can find a circuit encoding f, i.e., evaluating the circuit for $x \in \{0, 1\}^n$ yields $f(x)$.

Garbled Circuits Given a circuit, the idea behind garbling is basically to obfuscate the function encoded by it to some extend. Using an SKE scheme, we can *garble* a circuit as follows: First, we generate two keys k_0, k_1 for each wire. These keys represent the two possible values the wire can take. The input of each gate are now two keys $l \in \{l_0, l_1\}$ and $r \in \{r_0, r_1\}$. The output must be a key $out \in \{out_0, out_1\}$. For l_b and $r_{b'}$ we compute $\mathsf{E}(l_b, \mathsf{E}(r_{b'}, out_{g(b,b')}))$ which yields a list of four ciphertexts (as above, $g\{0, 1\}^2 \to \{0, 1\}$ is the binary function describing the functionality of the gate). A random permutation of that list, also referenced as *evaluation table* later, is the description of the gate. To evaluate a garbled gate, one can, given the input keys l and r, try all ciphers and see which one decrypts correctly to retrieve the output key. In this manner, the garbled circuit can be evaluated gate by gate.

We provide an implementation of a concrete garbled circuit for n-bit AND. While the inner workings of the code are explained in depth in Section 4 we already present the interface in Figure 2 here.

Oblivious Transfer The protocol we describe uses a cryptographic primitive called *oblivious transfer* (OT), introduced by [30]. More concretely, we use a 2-1 oblivious transfer for two parties. One party (S) has two secrets of which another party (R), may learn exactly one. S must not learn the choice of R while R must learn only one of the secrets.

In our Java code, we use an interface resembling an abstraction of a two message OT protocol (we refer the interested reader to Figure 12 in the appendix for the complete interface): The receiver R starts by generating some secret information `OTKey`. This is used to prepare the request `OTReq` which is then sent to S. From the request S can generate a response by providing to inputs `in0` and `in1`. If R receives the response, it can extract `in0` or `in1` depending on the value of `choice` used for generating the request.

2.3 The Verification Setup

We use *JML**, an extension of the Java Modelling Language (JML) for specification of Java programs and the *KeY-tool* as a prover. A full account of JML can be found in [22]. Specifications are given as annotations in the source code of a program. The main concept follows a design-by-contract approach, whose central specification artefacts are *method contracts* and *class invariants.*

A method contract is indicated by the keyword `normal_behaviour` and consists of a precondition, indicated by the keyword `requires` and a postcondition indicated by `ensures`. In this paper we do not consider contracts which argue about exceptions thrown by a method but prove that all methods terminate without throwing an exception. A method satisfies its contract if for all states satisfying the method's precondition it terminates in a state that satisfies the postcondition. A class invariant describes a global state which has to be preserved by all methods, unless the method is annotated by the keyword `helper`.

Pre-, postconditions and invariants are boolean valued JML expressions. A JML expression can be almost any side-effect free Java expression. Besides the built-in operators in Java, some additional operators can be used. One we frequently use is the all-quantifier `\forall T x; guard; body`, where `T` is the type over which the expression ranges, x is a variable of this type, and `guard` and `body` are boolean expressions. The operator `\old` can be used in the postcondition of a contract, taking an expression as input and evaluates to the value the expression evaluates to in the prestate of a method call. A special variable is `\result` which refers to the return value of a method.

[3] Although there are more general definitions of circuits, this one is simple and doesn't restrict our results.

We frequently make use of *ghost variables*, which are variables that can be used for specifications. They do not influence the actual behaviour of a program, but allow us to perform bookkeeping of information during execution of a program and we use them during proofs. Ghost variables cannot be referred to by Java code.

We use the KeY-tool [3] for proofs for Java programs. The interactive theorem prover built-in in the KeY-tool can is based on a generalization of Hoare logic. During a KeY proof, a program is *symbolically executed*, i.e., transformed into a set of logical constraints representing the behavior of the program. Using first-order reasoning, the KeY tool evaluates the postcondition, given the constraints.

3 The protocol

In this section we describe the concrete protocol we analyze and present the interfaces of sender and receiver. We model a two-party computation of an n-bit AND between a sender S and a receiver R following Yao's initial construction (see Figure 1). During a normal run of a protocol, both parties send exactly one message (we omit distributing the output computed by the receiver).

Intuitively, the protocol works as follows: R starts by preparing oblivious transfers (OTs) according to its own input. E.g., if R's ith input bit is 1, it will prepare the OT such that it will learn the second input of S to this OT later. R then sends the OTs to S. S generates a garbled circuit. For each input and each output wire of the circuit S generates a pair of keys that corresponds to the possible bit-value of the wire (0 or 1). For the input wires that belong to R's input, S fills the OTs received from R with the corresponding key pairs. R will receive only the corresponding key (and, by the security of the OT, R's choice remains hidden from S). S then sends the garbled circuit, the keys corresponding to its own input, the filled OTs and the key pairs for the output wires to R. R extracts the keys corresponding to its inputs from the OTs, evaluates the garbled circuit and can – using the key pairs for the output wires – interpret the resulting keys as a bitstring. This bitstring is the result of the two-party computation.

We now describe our implementation of this protocol in Java. The interfaces of sender and receiver are given in Figure 3 and Figure 4 respectively. They both contain a wrapper class for the message. The sender program uses the receiver interface and the other way round. Both use the interfaces for oblivious transfer from Figure 12, secret key encryption Figure 2, and garbled circuits Figure 2.

Figure 5 shows a `main` method as it could be provided by the environment. The protocol starts with a message by the receiver which contains n_{R} OT requests (class `OTReq`) where n_{R} is the bit length of R's input. When the sender receives this message it generates a garbled circuit (using the interfaces $\mathcal{I}_{\mathsf{GC}}$ and $\mathcal{I}_{\mathsf{SKE}}$) and prepares the response `SenderMessage` as follows:

- `gc` contains the garbled circuit.
- `sender_keys` contains n_{S} keys corresponding to the first n_{S} input bits of the garbled circuit where n_{S} is the bit length of S's input.
- `ots` contains n_{R} OT responses. Response i contains the two keys corresponding to the $(n_{\mathsf{S}}+i)$th input bit of the garbled circuit.
- `out0` and `out1` contain the key corresponding to a 0 and to a 1 output respectively.

Finally, the method `getOutput`, given the `SenderMessage`, evaluates the garbled circuit: First, R retrieves the n_{R} keys matching its input from the oblivious transfers. R then calls the `evaluate` function of the garbled circuit which yields the output key. Depending on whether the output key is `out0` or `out1`, it outputs `true` or `false`.

4 Correctness of our protocol

In this section we describe the proof of correctness for our implementation. We provide a formal specification for the interfaces $\mathcal{I}_{\mathsf{Key}}$, $\mathcal{I}_{\mathsf{Cipher}}$ and $\mathcal{I}_{\mathsf{SKE}}$. Further, we provide a realization of the interface $\mathcal{I}_{\mathsf{GC}}$ in Java using the interfaces of the cryptographic primitives and a formal specification of invariants for the realization. Finally, we proof the correctness of our implementation with respect to method contracts, which yields correctness of the realization of the interface $\mathcal{I}_{\mathsf{GC}}$.

We will present in this paper only the core of the specification and implementation for a compact presentation. The complete implementation and machine-assisted proofs are available online [4].

[4] http://formal.iti.kit.edu/~greiner/cans2014/CodeAndProofCANS2014.zip

```
public final class SenderMessage {
  GarbledCircuit gc;
  Key[] sender_keys;
  OTResp[] ots;
  Key out0, out1;
}
public final class Sender {
  public Sender(boolean[] input);
  public SenderMessage
    getMessage(Receiver r, ReceiverMessage m);
}
```

Fig. 3. Interface for sender in our protocol.

```
public final class ReceiverMessage {
  OTReqt[] ots;
}

public final class Receiver {
  public Receiver(boolean[] input);
  public ReceiverMessage getMessage();
  public boolean
    getOutput(Sender s, SenderMessage m);
}
```

Fig. 4. Interface for receiver in our protocol.

```
public static void main(String[] args) {
  boolean[] sender_in = {true, true};
  boolean[] receiver_in = {false, true};

  Sender s = new Sender(sender_in);
  Receiver r = new Receiver(receiver_in);
  ReceiverMessage m1 = r.getMessage();
  SenderMessage m2 = s.getMessage(m1);
  boolean out = r.getOutput(m2);
}
```

Fig. 5. Example for a `main` method utilizing sender and receiver.

A word on modularity and re-usability One of the most tedious tasks during the verification of a program is finding a correct and sufficient specification. This is especially true in the case of garbled gates and circuits, because a lot of information is given implicitly by the code and the interworking of methods following after another. In order to prove correctness, we have to make this information explicit in the form of class invariants.

Our implementation is modular in the way that only the contracts of other objects are used for verification, not their actual implementation. In order to implement binary gates with different algorithms the same functionality the specification provided can be re-used. Also, when binary gates realizing a different function are implemented, our specification can be reused, by only changing the specification of the truth table and fixing two lines in the postcondition of one method. As a proof of concept, we implemented and verified an additional garbled gate with XOR-functionality, which can be found in the online sources. The proof process for the new gate is essentially the same for both gates.

A circuit is built by wiring gates (which are used in a black-box fashion but may again be circuits themselves) in a certain way. The way gates are wired is called the *topology* of the circuit. In our work we use something one could call a *linear topology* for the circuit that then forms the n-bit AND. Our specification can easily be re-used for other circuits with a linear topology. For example, realizing an n-bit OR would only require straightforward changes in two lines of the postcondition of the evaluate method. Realizations of n-bit AND and n-bit OR are particularly interesting to us because they are the basis for disjunctive and conjunctive normal forms. Further research is necessary in order to identify and specify a practical set of topologies for circuits which allow an effective realization of arbitrary formulas.

4.1 Encryption Abstraction

Instead of using a real implementation for the interfaces $\mathcal{I}_{\mathsf{SKE}}$ from Figure 2 we provide an abstract specification of this cryptographic primitive following the design-by-contract paradigm. Figure 6 shows the specification of the classes `Cipher` and `Key`.

A `Cipher` has two ghost fields we use for specifying the encryption information. The ghost field `key` holds the key, which was used for encryption of a message. We call two objects of type `Key` corresponding, if they have the same `ident` value. The ghost field `msg` holds the message that is encrypted in the cipher by a key corresponding to the value of `key`.

Instead of directly using constructors or methods provided by `Key` or `Cipher`, we encapsulate this functionality in a secret key encryption scheme, providing the interface $\mathcal{I}_{\mathsf{SKE}}$. The specification used for verification is shown in Figure 7.

The SKE class has two static ghost fields. The field randoms holds a collection of random numbers, which represent random byte arrays. The content of randoms can be seen as a stream of random numbers from which elements can be drawn. This way, we make execution of the methods provided by SKE deterministic and we can treat randomisation independent from execution of our actual code.

The field counter is a pointer to the element in randoms which is drawn the next time a random number is needed. The management of this pointer is ensured by the methods provided by SKE. To create a new key, we use the method GenKey. It returns a new Key object, where the identifier has the same value as the next element in randoms. Line 11 expresses the assumption that each time a random number is drawn, it is different to all random numbers drawn earlier by SKE. This assumption is justified since the probability of a collision has to be be negligible for any polynomial number of generated keys.

The method SKE.Encrypt encapsulates the encryption functionality in our program. The Cipher object returned by the method has a fresh random number as identifier. The ghost field key remembers the key which was used for encryption, while the ghost field msg remembers the clear text information.

The method SKE.Decrypt provides decryption functionality, which ensures that a null object is returned, if the key passed as parameter does not correspond to the key originally used for encryption of the message. In the case of a corresponding key being provided, we consider two situations. First, we state in lines 36ff the expected behaviour for the case when the encrypted message is of type Cipher, which encrypts a Key object. In this case, the method returns a cipher corresponding to the originally encrypted one, which also encrypts a Key object corresponding to the key originally encrypted twice.

Second, we specify on lines 42ff that if the encrypted message is of type Key, the Decrypt method returns a key corresponding to the originally encrypted key object. So the object structure is preserved by decryption. For other cases, we leave the behaviour of Decrypt underspecified.

4.2 Implementation Details

Our realization of a garbled circuit consists of two classes. The class GarbledANDGate implements a garbled gate with binary AND functionality. The GCnBitAND makes use of the class GarbledANDGate to realize the interface $\mathcal{I}_{\mathsf{GC}}$.

GarbledANDGate The class GarbledANDGate defines the field Cipher[] eT, representing the evaluation table as explained in Section 2.2. We introduce several ghost variables for bookkeeping of the state of a GarbledANDGate shown in Figure 8 to explicitly store information given by the structure of the evaluation table.

The variables kl0 and kl1 store an object of type Key which represents the keys expected as 0- or 1-valued input on the left pin. The variables kr0 and kr1 do the same for the right pin. The variables out0 and out1 hold the objects representing the keys used as 0- and 1-valued output.

The evaluation table holds on each position an encryption either of out0 or out1, first encrypted by one of the input keys for the right pin and then encrypted by one of the input keys for the left pin. As explained in Section 2.2, the evaluation table contains a random permutation the ciphers and we use the ghost variables ci0, ci1, ci2, ci3 to store the indices of the ciphers after permutation.For example, the encryption of out0 with the keys kr0 and kl0 is stored at position eT[ci0]. A more detailed definition of the invariant can be found in the Appendix in Figure 13, for a full account, the reader may be referred to implementation.

Further, we provide a method contract for the constructor and the method evaluate provided by the gate. These contracts ensure that the gate does realize a garbled AND functionality, assuming some preconditions. The correctness of the implementation according to the contract is used as a lemma during the proof of correctness of the garbled circuit.

```
public final class Key {
  public final int ident;
   ...
}
```

```
public final class Cipher {
 //@ public ghost Key key;
 //@ public ghost Object msg;
 public final int ident;
  ... }
```

Fig. 6. Specification of classes Cipher and Key providing the interfaces $\mathcal{I}_{\mathsf{Cipher}}$ and $\mathcal{I}_{\mathsf{Key}}$

```
public final class SKE {
 //@ public static ghost int counter;
 //@ public static ghost int[] randoms;

 /*@ public normal_behaviour
   @ requires
   @     0 <= counter < randoms.length;
   @ ensures
   @     \old(counter) + 1 == counter &&
   @     \result.ident == randoms[counter-1] &&
   @     (\forall int i; 0 <= i < randoms.length &&
   @            i != \old(counter);
   @         \result.ident != randoms[i]);
   @ ... */
 public static Key GenKey() { ... }

 /*@ public normal_behaviour
   @ requires
   @    0 <= counter < randoms.length;
   @ ensures
   @    \result.key == k &&
   @    \result.message == m &&
   @    \result.ident == randoms[counter-1] &&
   @    \old(counter) + 1 == counter &&
   @    (\forall int i; 0 <= i < randoms.length &&
   @         i != \old(counter);
   @       \result.ident != randoms[i]) &&
   @  ... ;
   @ */
 public static Cipher Encrypt(
         Key k, Object m) { ... }

 /*@ public normal_behaviour
   @ ensures
   @   \if (c.key.ident == key.ident)
   @   \then (\if(\typeof(c.msg) == Cipher &&
   @            \typeof(c.msg.msg) == Key)
   @      \then (\typeof(\result) == Cipher &&
   @         \typeof(\result.msg) == Key &&
   @         \result.ident == c.msg.ident &&
   @         \result.msg.ident == c.msg.msg.ident)
   @      \elseif(\typeof(c.msg) == Key)
   @      \then (\typeof(\result) == Key &&
   @         \result.ident == c.msg.ident)))
   @   \else (\result == null); */
 public static Object Decrypt(
            Key key, Cipher c) { ... }
}
```

Fig. 7. Specification of class `SKE` providing $\mathcal{I}_{\mathsf{SKE}}$

```
public final class GarbledANDGate {
  Cipher[] eT;
  /*@ ghost Key kl0, kl1, kr0, kr1,
    @ out0, out1;
    @ ghost int ci0, ci1, ci2, ci3;*/
  public GarbledANDGate(Key kl0, kl1,
    kr0, k1, out0, out1){...}
  public Key evaluate(Key inl, inr){...} }
```

Fig. 8. Definition of `GarbledANDGate` including Ghost variables.

```
public final class GCnBitAND {
  GarbledANDGate[] gates;
  /*@ ghost Key[] in0, in1;
    @ ghost Key out0, out1;*/
}
```

Fig. 9. Definition of `GCnBitAND` including ghost variables.

Garbled n-bit AND circuit The class `GCnBitAND`, realizing $\mathcal{I}_{\mathsf{GC}}$, defines the field `gates` as an array of type `GarbledANDGate[]`. The gates stored in this array are responsible for the functionality implemented by the circuit. The correct wiring of the circuit is indirectly ensured by the constructor and the `evaluate` method.

We define four ghost variables as shown in Figure 9. The keys expected by the circuit as 0-valued input are stored in the array `in0`, those representing a 1-valued input are stored in the array `in1`. The ghost variables `out0` and `out1` store the keys representing the respective 0- and 1-valued outputs.

The invariant of `GCnBitAND` ensures a well-definedness property and can be found in Figure 14 in the appendix. First, it is ensured that the expected input of a gate on the left pin corresponds to the output of the previous gate, i.e. the circuit implements a linear structure. Second, it is ensured that the keys expected as an input on the right pin by the gates correspond to the keys expected as input by the circuit. A spacial case here is the first gate, for which both inputs come from the user. Finally, the invariant ensures that the output of the last gate corresponds to the output the circuit is supposed to provide.

4.3 Correctness of the Implementation

We prove the correctness of our implementation of the methods provided by `GCnBitAND` against their contracts.

```
 1 /*@ requires
   @   inkeys0.length == inkeys1.length &&
 3 @   inkeys0.length >= 2 &&
   @   (\forall int i; 1 <= i < inkeys0.length;
 5 @      \distinct(inkeys0[i−1], inkeys1[i−1],
   @          inkeys0[i], inkeys1[i].ident)) &&
 7 @   \distinct(outkey0, outkey1); */
   @ ensures
 9 @   (\forall int i; 0 <= i < in0.length;
   @      inkeys0[i].ident == in0[i].ident &&
11 @      inkeys1[i].ident == in1[i].ident) &&
   @   outkey0.ident == out0.ident &&
13 @   outkey1.ident == out1.ident;
   public GarbledNBitANDCircuit(Key[] inkeys0,
15   inkeys1; Key outkey0, outkey1) {...}
```

Fig. 10. Contract of constructor of `GCnBitAND`.

```
 1 /*@ requires
   @ in.length == in0.length &&
 3 @ (\forall int i; 0 <= i < in.length;
   @    in[i].ident == in0[i].ident ||
 5 @    in[i].ident == in1[i].ident);
   @ ensures
 7 @   \if (\forall int i; 0 <= i < in.length;
   @       in[i].ident == in1[i].ident)
 9 @   \then (\result.ident == out1.ident)
   @   \else (\result.ident == out0.ident); */
11 public Key evaluate(Key[] in) {...}
```

Fig. 11. Contract of `evaluate` for `GCnBitAND`.

Figure 10 shows the contract for the constructor of `GCnBitAND`. It requires in line 2 that as many keys provided as 0-valued input are also provided as 1-valued input. The amount of keys provided determines the number of pins provided by the circuit. At least two keys have to be provided for each value, which expresses the circuit to be built provides at least two pins.

The quantifier at line 4 states that the keys used as possible input for each pin do not correspond, neither do the keys used as possible input on two subsequent gates. The predicate `distinct` used here indicates that its arguments have pairwise unequal values of the respective `ident` fields. Both of these conditions are necessary for the correctness of the gates, which are built during execution of the constructor. Line 7 states that the possible output keys do not correspond, i.e. it can be distinguished between a 0- and a 1-valued output.

The postcondition in line 9 states that the identifier representing expected input keys in the ghost fields `in0` and `in1` are the same as the identifiers of the keys used as input to the constructor. Additionally, the invariant is required to hold after termination of the constructor, which is an implicit postcondition for the constructor.

Theorem 1 states the correctness of the constructor of $\mathcal{I}_{\mathsf{GC}}$, since its implementation is correct with respect to its specification.

Theorem 1. *Let GCnBitAND be the realization of $\mathcal{I}_{\mathsf{GC}}$; let $in0$ and $in1$ be arrays of keys, such that the length of $in0$ equals the length of $in1$, all elements in $in0$ and $in1$ are not `null` and $in0$ contains at least two elements; let $out0$ and $out1$ be keys that are not `null` and let all keys in $in0$, $in1$ and the keys $out0$ and $out1$ have pairwise different identifier.*

Then a call $\mathcal{I}_{\mathsf{GC}}$.GCnBitAND(in0, in1, out0, out1) returns an object gc realizing $\mathcal{I}_{\mathsf{GC}}$ such that the invariant of gc holds and; for all $i \in \{0 \,..\, length\ of\ in0 - 1\}$ the keys expected by gc as input on pin n representing a 0-valued input corresponds to the key $in0[n]$ and the key expected by gc as input on pin n representing a 1-valued input corresponds to the key $in1[n]$ and the key provided by gc as output representing a 0-valued output corresponds to $out0$ and the key provided by gc as output representing a 1-valued output corresponds to $out1$.

Proof. The conditions stated as preconditions in Theorem 1 imply the preconditions as stated in Figure 10. The postconditions stated in Theorem 1 are equivalent to the postcondition in Figure 10. The realization of $\mathcal{I}_{\mathsf{GC}}$ is given by the class `GCnBitAND`.

We proved with the KeY-tool that `GCnBitAND.GCnBitAND` satisfies its contract. Therefore Theorem 1 holds.

The contract of `evaluate` of `GCnBitAND` is shown in Figure 11. Line 2 requires R to provide exactly one key for each pin of the circuit, while each key either has to correspond to the key expected by the circuit on the respective pin as 0- or 1-valued input (line 3). Implicitly it is also required that the invariant of the circuit holds right before a call to `evaluate`, although not directly stated in the precondition.

The postcondition states in line 7 that a correct implementation of `evaluate` returns a key corresponding to the 1-valued output key, if all input keys correspond to 1-valued input. If at least one input key does not correspond to a 1-valued input, the circuit returns a 0-valued output (line 10). It is easy to see that the postcondition expresses a n-bit AND functionality.

Theorem 2 states the correctness of method `evaluate` of $\mathcal{I}_{\mathsf{GC}}$, since the implementation of the constructor is correct due to its specification.

Theorem 2. *Let* `GCnBitAND` *be the realization of* $\mathcal{I}_{\mathsf{GC}}$*; let gc be the object giving access to* $\mathcal{I}_{\mathsf{GC}}$*; let the invariant of gc hold; let in be an array of keys with the same length as the amount of expected input keys by gc and let* $in[i]$ *correspond to the* 0*-valued or* 1*-valued input key expected by gc.*

Then a call $\mathcal{I}_{\mathsf{GC}}$`.evaluate(in)` *returns a key o such that o represents a* 1*-valued output if all keys in in represent* 1*-valued inputs and o represents a* 0*-valued output if not all keys in in represent* 1*-valued inputs.*

Proof. The conditions stated in Theorem 2 are equivalent to the precondition as stated in Figure 11. The postcondition stated in Theorem 2 are equivalent to the postcondition as stated in Figure 11. The realization of $\mathcal{I}_{\mathsf{GC}}$ is given by the class `GCnBitAND`.

We proved with the KeY-tool that the method `evaluate` of class `GCnBitAND` satisfies its contract. Therefore Theorem 2 holds.

5 Security of our protocol

In this section we prove that our protocol (see Section 3) is secure against an honest-but curious adversary if the sender S is corrupted. Security in this setting means, that the inputs of the receiver R remain secret. This follows from the correctness of the protocol and the UC-security of oblivious transfer.

Figure 15 describes the interface of the two-party computation in presence of a corrupted sender to the environment. In addition to what a passive adversary can usually observe during a run of the protocol, it now gets access to methods `getSenderInput` and `getSenderKeys` which leak S's secrets. In particular, `getSenderKeys` returns the list of all encryption keys generated by S to construct the garbled circuit. Note that the adversary cannot change the behavior of S.

We now introduce two implementations of the interface $\mathcal{I}_{(\hat{\mathsf{S}},\mathsf{R})}$.

- The real implementation $\mathsf{2PC}_{\mathsf{real}}^{(\hat{\mathsf{S}},\mathsf{R})}$ runs the two-party protocol in the constructor on the given inputs and saves the exchanged messages, generated encryption keys, etc. for later retrieval by the adversary through the corresponding getters.
- The ideal implementation $\mathsf{2PC}_{\mathsf{ideal}}^{(\hat{\mathsf{S}},\mathsf{R})}$ doesn't run the protocol but uses a simulator implementing the interface from Figure 16 to provide the output of all getters except `getOutput`. It resembles a wrapper for the ideal functionality (for corrupted S and honest R) running in parallel with a simulator. Note that the simulator is only given the input of the sender and the length of the receiver's input.

$\mathsf{OT}_{\mathsf{ideal}}$ is an ideal implementation of the interface for oblivious transfer from Figure 12 resembling the ideal functionality $\mathcal{F}_{\mathsf{OT}}$. Internally, $\mathsf{OT}_{\mathsf{ideal}}$ maintains a list of tuples `(OTKey k, boolean choice, OTResp r, Object in0, Object in1)` each representing one OT instance. Constructing an OT request creates a new entry in that list with `r`, `in0` and `in1` set to `null`. Upon `genResponse` for an `OTKey` the corresponding `r`, `in0` and `in1` are set according to the given values (which must not be `null` and the length of their serialization must not exceed a fixed maximum). On `getOutput` for `OTResp r` and `OTKey k`, depending on the value of `choice`, `in0` or `in1` is returned. The `ident` attributes of keys, responses and requests are set to uniformly random values on object creation.[5] As usual throughout the paper, these idents are placeholders that can be used for data by a real implementation.

Theorem 3. *Let* $\mathsf{2PC}_{\mathsf{real}}^{(\hat{\mathsf{S}},\mathsf{R})}$, $\mathsf{2PC}_{\mathsf{ideal}}^{(\hat{\mathsf{S}},\mathsf{R})}$ *and* $\mathsf{OT}_{\mathsf{ideal}}$ *be the programs introduced above and* $\mathsf{SKE}_{\mathsf{real}}$ *be a correct implementation of* $\mathcal{I}_{\mathsf{SKE}}$*, then we have*

$$\mathsf{SKE}_{\mathsf{real}} \cdot \mathsf{OT}_{\mathsf{ideal}} \cdot \mathsf{2PC}_{\mathsf{real}}^{(\hat{\mathsf{S}},\mathsf{R})} \leq^{(\mathcal{I}_{\mathsf{out}},\emptyset,\emptyset,\mathcal{I}_{\mathsf{Sim}})} \mathsf{SKE}_{\mathsf{real}} \cdot \mathsf{OT}_{\mathsf{ideal}} \cdot \mathsf{2PC}_{\mathsf{ideal}}^{(\hat{\mathsf{S}},\mathsf{R})}$$

[5] Theoretically, other distributions are also possible. For Theorem 3 we just need that the idents of OT requests are independent of the choice bit.

where $\mathcal{I}_{\mathsf{out}} := \mathcal{I}_{(\hat{\mathsf{S}},\mathsf{R})} \cup \mathcal{I}_{\mathsf{OT}} \cup \mathcal{I}_{\mathsf{SKE}}$ *and* $\mathcal{I}_{\mathsf{Sim}}$ *the interface from Figure 16.*

Proof. The simulator can successfully fake `getReceiverMessage` because the OT keys are random handles independent of `choice` in $\mathsf{OT}_{\mathsf{ideal}}$. It creates `inR_length` OTs and returns the corresponding OT requests on a call of `getReceiverMessage`. To simulate the sender, it generates a garbled circuit as an honest S would do and prepares the OT responses accordingly to assemble the sender response. The correctness of the garbled circuit (see Theorem 1, Theorem 2) guarantees that the output in the real world actually matches that in the ideal world.

What remains to do is to show that we can replace the ideal implementation for oblivious transfer by a real one.

Security of Oblivious Transfer We first describe the simplified functionality for oblivious transfer [7]. $\mathcal{F}_{\mathsf{OT}}$ interacts with a sender S and a receiver R.

- Upon receiving a message (in_0, in_1) from S, store (in_0, in_1).
- Upon receiving a message b from R, check if a (in_0, in_1) message was previously sent by S. If yes, send in_b to R. If not, send nothing to R (but continue running).

Let $\mathsf{OT}_{\mathsf{real}}$ be a system that implements the OT interface from Figure 12. We show that the system $\mathsf{OT}_{\mathsf{ideal}}$ can safely be replaced by $\mathsf{OT}_{\mathsf{real}}$ if $\mathsf{OT}_{\mathsf{real}}$ suitably implements a two-party-two-message realization of $\mathcal{F}_{\mathsf{OT}}$. Such realizations exists under standard cryptographic assumptions, e.g., decisional Diffie-Hellman, quadratic residuosity, or learning with errors [29].[6]

Theorem 4. *If* $\mathsf{OT}_{\mathsf{real}}$ *implements a realization* $\mathcal{R}$ *of* $\mathcal{F}_{\mathsf{OT}}$*, then* $\mathsf{OT}_{\mathsf{real}} \leq^{(\mathcal{I}_{\mathsf{OT}},\emptyset,\emptyset,\emptyset)} \mathsf{OT}_{\mathsf{ideal}}$

Proof. The basic idea is that, since $\mathcal{R}$ realizes $\mathcal{F}_{\mathsf{OT}}$, there is a simulator $\mathcal{S}$ such that $\mathcal{R}$ and $\mathcal{S} \cdot \mathcal{F}_{\mathsf{OT}}$ are indistinguishable for every environment in the computational UC model (for a suitable composition $\cdot$ in that model). The output of $\mathcal{S}$ is independent from the original inputs of the parties S and R (it doesn't get those values from $\mathcal{F}_{\mathsf{OT}}$). As output distribution for $\mathsf{OT}_{\mathsf{ideal}}$ we can hence pick that of $\mathcal{S}$. Since we can simulate Turing machines with Jinja+ programs, simulatability in the CVJ framework in the sense of Definition 1 follows.

6 Future work

This work provides the proof of security of a two-party computation implemented in Java against a semi-honest sender. In particular, we prove correctness of the implementation of a garbled circuit using cryptographic primitives via a formally specified interface.

One obvious direction for future work is to prove security for the two remaining scenarios, i.e., security against a corrupted receiver and security if both parties are honest (all in presence of a passive adversary).

One interesting challenge towards this goal is to prove at code level that the evaluation of a garbled circuit does not leak more than the encoded function f and the output $f(x)$.[7] For this, implementation details of the garbled circuit (e.g., that the evaluation table is randomized) will become important.

Since the security against a corrupted receiver will also depend on the security of the used encryption scheme, a suitable functionality for secret key encryption will be necessary. This functionality should be realizable in the sense of strong simulatability and sufficient for the construction of garbled circuits.

Finally, it would be interesting to build a compiler from functions to (garbled) circuits that automatically outputs Java code that is verifiably correct. E.g., if we have the description of a function in conjunctive normal form (one multi-bit AND, a number of multi-bit ORs and NOTs) we can use the modularity of our correctness proof as explained in Section 4. However, more work would be needed to get from the proof for a conjunctive normal form to a high-level description of the function like "addition of two integers given as bitstrings".

[6] These realizations need a common reference string functionality [6] which can be part of $\mathsf{OT}_{\mathsf{real}}$.

[7] Actually, a garbled circuit should leak f only to some extent. However, since f is public in our setting, even a complete leakage of f would not be problematic which relaxes the difficulty of the proof.

Bibliography

[1] Mike Barnett, Rustan Leino, and Wolfram Schulte. The spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-24287-1.

[2] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and SantiagoZanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, Lecture Notes in Computer Science, pages 71–90. Springer Berlin Heidelberg, 2011.

[3] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 3-540-68977-X, 978-3-540-68977-5.

[4] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12*, pages 784–796. ACM Press, October 2012.

[5] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[6] Ran Canetti and Tal Rabin. Universal composition with joint state. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 265–281. Springer, August 2003.

[7] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th ACM STOC*, pages 494–503. ACM Press, May 2002.

[8] Seung Geol Choi, Jonathan Katz, Hoeteck Wee, and Hong-Sheng Zhou. Efficient, adaptively secure, and composable oblivious transfer with a single, global CRS. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 73–88. Springer, February / March 2013. doi: 10.1007/978-3-642-36362-7_6.

[9] Ivan Damgård, Jesper Buus Nielsen, and Claudio Orlandi. Essentially optimal universally composable oblivious transfer. In Pil Joong Lee and Jung Hee Cheon, editors, *ICISC 08*, volume 5461 of *LNCS*, pages 318–335. Springer, December 2008.

[10] Zekeriya Erkin, Martin Franz, Jorge Guajardo, Stefan Katzenbeisser, Inald Lagendijk, and Tomas Toft. Privacy-preserving face recognition. In *Privacy Enhancing Technologies*, pages 235–253, 2009.

[11] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-73367-6.

[12] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[13] S. Greiner, P. Birnstill, E. Krempel, B. Beckert, and Jürgen Beyerer. Privacy preserving surveillance and the tracking-paradox. In M. (Ed.) Lauster, editor, *8th Future Security. Security Research Conference. Proceedings.*, pages 296–302, Berlin, September 2013. Fraunhofer Verlag.

[14] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, 2011.

[15] Yan Huang, Lior Malka, David Evans, and Jonathan Katz. Efficient privacy-preserving biometric identification. In *NDSS*, 2011.

[16] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 41–55, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-20397-8.

[17] Somesh Jha, Louis Kruger, and Vitaly Shmatikov. Towards practical privacy for genomic computation. In *2008 IEEE Symposium on Security and Privacy*, pages 216–230. IEEE Computer Society Press, May 2008.

[18] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, July 2008.

[19] Ralf Küsters, Tomasz Truderung, and Jürgen Graf. A framework for the cryptographic verification of java-like programs. In *CSF 2012*, pages 198–212. IEEE, 2012.

[20] Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Jürgen Graf, and Christoph Scheben. A hybrid approach for proving noninterference and applications to the cryptographic verification of Java programs. In Christian Hammer and Sjouke Mauw, editors, *Grande Region Security and Reliability Day 2013*, Luxembourg, 2013.

[21] Ralf Küsters, Enrico Scapin, Tomasz Truderung, and Juergen Graf. Extending and applying a framework for the cryptographic verification of java programs. *IACR Cryptology ePrint Archive*, 2014:38, 2014.

[22] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: A behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006. ISSN 0163-5948.

[23] Yehuda Lindell and Benny Pinkas. A proof of yao's protocol for secure two-party computation. *Electronic Colloquium on Computational Complexity (ECCC)*, (063), 2004.

[24] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 52–78. Springer, May 2007.

[25] Yehuda Lindell, Benny Pinkas, and Nigel P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *SCN 08*, volume 5229 of *LNCS*, pages 2–20. Springer, September 2008.

[26] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*, pages 287–302, 2004.

[27] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 368–386. Springer, March 2009.

[28] Margarita Osadchy, Benny Pinkas, Ayman Jarrous, and Boaz Moskovich. SCiFI - a system for secure face identification. In *2010 IEEE Symposium on Security and Privacy*, pages 239–254. IEEE Computer Society Press, May 2010.

[29] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 554–571. Springer, August 2008.

[30] Michael O. Rabin. How to exchange secrets with oblivious transfer. In *Technical Report TR-81*. Harvard University, 1981.

[31] Peter H. Schmitt and Isabel Tonin. Verifying the mondex case study. In *SEFM*, pages 47–58. IEEE Computer Society, 2007. ISBN 978-0-7695-2884-7.

[32] Wolfram Schulte, Xia Songtao, Jan Smans, and Frank Piessens. A glimpse of a verifying c compiler. In *C/C++ Verification Workshop 2007*, 2007.

[33] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.

[34] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

Appendix

```
public final class OTKey {
  public final int ident;
  public OTKey(int id);
}
public final class OTReq {
  public int ident;
  public OTReq(OTKey key, boolean choice);
  public OTResp
    genResponse(Object in0, Object in1);
}
public final class OTResp {
  public int ident;
  public OTResp(OTReq request,
                Object in0, Object in1);
  public Object getOutput(OTKey key);
}
```

Fig. 12. Interface $\mathcal{I}_{\mathsf{OT}}$ for oblivious transfer.

```
/*@ invariant
  @ eT[ci0].key.ident == kl0.ident &&
  @ eT[ci0].msg.key.ident ==
  @       kr0.ident &&
  @ eT[ci0].msg.msg.ident ==
  @       out0.ident &&
  @ eT[ci1].key.ident == kl0.ident &&
  @ eT[ci1].msg.key.ident ==
  @       kr1.ident &&
  @ eT[ci1].msg.msg.ident ==
  @       out0.ident &&
  @ eT[ci2].key.ident == kl1.ident &&
  @ eT[ci2].msg.key.ident ==
  @       kr0.ident &&
  @ eT[ci2].msg.msg.ident ==
  @       out0.ident &&
  @ eT[ci3].key.ident == kl1.ident &&
  @ eT[ci3].msg.key.ident ==
  @       kr1.ident &&
  @ eT[ci3].msg.msg.ident ==
  @       out1.ident; */
```

Fig. 13. Invariant for `GarbledANDGate` including ghost variables.

```
/*@ invariant
  @ (\forall int i; 0 <= i < gates.length−1;
  @   gates[i].out0 == gates[i+1].kl0 &&
  @   gates[i].out1 == gates[i+1].kl1) &&
  @ (\forall int i; 0 <= i < gates.length;
  @   gates[i].kr0 == in0[i+1] &&
  @   gates[i].kr1 == in1[i+1]) &&
  @ gates[0].kl0 == in0[0] &&
  @ gates[0].kl1 == in1[0] &&
  @ gates[gates.length−1].out0 == out0 &&
  @ gates[gates.length−1].out1 == out1;
```

Fig. 14. Invariant of `GCnBitAND`.

```
public final class ProtoCS {
  public ProtoCS(boolean[] inS, boolean[] inR);
  public ReceiverMessage getReceiverMessage();
  public boolean[] getSenderInput();
  public Key[] getSenderKeys();
  public SenderMessage getSenderMessage();
  public boolean getOutput();
}
```

Fig. 15. Interface $\mathcal{I}_{(\hat{\mathsf{S}},\mathsf{R})}$ for a passive adversary in case of a corrupted sender.

```
public final class Simulator  {
  public Simulator(boolean[] inS, int inR_length);
  public ReceiverMessage getReceiverMessage();
  public boolean[] getSenderInput();
  public Key[] getSenderKeys();
  public SenderMessage getSenderMessage();
  public boolean getOutput();
}
```

Fig. 16. Interface $\mathcal{I}_{\mathsf{Sim}}$ for simulator.

```
public final class ProtoCS {
  private final Simulator sim;
  public ProtoCS(boolean[] inS, boolean[] inR) {
    sim = new Simulator(inS, inR.length)
  }
  public ReceiverMessage getReceiverMessage() {
    return sim.getReceiverMessage();
  }
   ...
  public boolean getOutput() {
    return AND(inS, inR); // AND of all input bits
  }
}
```

Fig. 17. Implementation $\mathsf{2PC}_{\mathsf{ideal}}^{(\hat{\mathsf{S}},\mathsf{R})}$ of interface $\mathcal{I}_{(\hat{\mathsf{S}},\mathsf{R})}$.