UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR IT SECURITY

Garbled Circuits: From White-Box Cryptography to Zero-Knowledge Proofs

Garbled Circuits: Von White-Box Cryptography zu Zero-Knowledge Proofs

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Jacqueline Thaeter

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
Okan Seker, M.Sc.

Lübeck, den 27.09.2018

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, den 27.09.2018

Abstract

This thesis covers the topics of garbled circuits, white-box cryptography and zero-knowledge proofs. We present multiple new approaches on how to use garbled circuits in white-box cryptography. White-box cryptography based on garbled circuits has the advantage of being universal, meaning it can be applied to arbitrary circuits. One of the protocols proposed is used for creating a new kind of zero-knowledge proofs. We designed a zero-knowledge proof protocol that uses garbled circuits, but does not need any oblivious transfer or commitment. Different variations of the protocol are analyzed and compared to state-of-the-art protocols.

Zusammenfassung

Diese Arbeit beschäftigt sich mit den Themengebieten garbled circuits, white-box cryptography und zero-knowledge Beweisen. Wir präsentieren verschiedene neue Ansätze, wie sich garbled circuits in der white-box cryptography verwenden lassen. Das Verwenden von garbled circuits in einem white-box Kontext hat den Vorteil, dass die resultierenden Verfahren universell sind, was bedeutet, dass sie mit beliebigen Schaltkreisen verwendet werden können. Eines der präsentierten Protokolle wird von uns verwendet um eine neue Art von zero-knowledge Beweisen zu erschaffen. Wir haben ein zero-knowledge Beweisprotokoll entworfen, dass garbled circuits verwendet, jedoch weder oblivious transfer noch commitment benötigt. Verschiedene Varianten dieses Protokolls werden analysiert und mit aktuellen Protokollen verglichen.

Contents

1 Introduction

Today, music, films and games are commonly consumed via streaming platforms. Less and less media is distributed physically, and the amount of media distributed over the internet constantly increases. This brings a new challenge: How can content be provided without allowing the user to extract the data? Ideally, a piece of software on the user's hardware is used to decrypt the data that was sent in an encrypted form. The user should not be able to extract the data without making use of the software. This way, access to the data can also be limited to specific data or to a limited period of time. But how can such software be installed on a user's device and simultaneously be secured in a way that the user cannot extract secret keys? This is the main issue of white-box cryptography.

Streaming is not the only application for white-box implementations. Licensed productivity software is another important use case. It is also possible to create white-boxes that enable the user to apply a certain cryptographic operation on self-chosen inputs. The key used for the cryptographic operation is embedded in the white-box in such a way, that it cannot be extracted by the user. This prevents the user from inverting the cryptographic operation or using the key in another manner he is not supposed to.

White-box implementations work by obfuscating the executed operation. This is similar to garbled circuits, where the inputs to an operation are obfuscated in order to do secure two party computation. Inspired by this similarity, we constructed different protocols combining white-box cryptography with garbled circuits. One of them, called *simple white-box encryption scheme* (`SWBES`), enables a sender to transmit an encrypted message to a receiver in a way that the receiver can decrypt the message, but does not get any information about the ciphertext. This is an interesting property that is not needed for data transfer, but opens the doors to another field – zero-knowledge proofs.

Using the `SWBES` as a basis, we developed a zero-knowledge proof protocol using garbled circuits. We call it `ZKGC-nOT`, since unlike the only

other existing protocol using garbled circuits for zero-knowledge proofs by Jawurek et al. [JKO13], it does not need any oblivious transfer. `ZKGC-nOT` is a completely new approach to using garbled circuits for zero-knowledge proofs, as there are many differences to the JKO13 protocol. It is also very versatile and can be varied in different ways. We present a variant that uses oblivious transfer to get other bonus properties and show how it can be converted into a non-interactive zero-knowledge proof.

In Section 2 we introduce commitment, oblivious transfer, garbled circuits and zero-knowledge proofs together with the corresponding notations so we can use these techniques in the following sections. Next, in Section 3, white-box cryptography is introduced. We present a single-use white-box using garbled circuits along with the protocols `SWBES` and `MWBES`. These protocols are analyzed thoroughly and an introduction to a two-layer scheme is given. The `SWBES` protocol is the basis for our new zero-knowledge proof protocol using garbled circuits, presented in Section 4. We analyze the protocol, show some variants and compare it to related work. In Section 5 we draw our conclusion and present ideas for future work.

2 Preliminaries and Notations

In the following, some preliminaries including commitment, oblivious transfer, garbled circuits and zero-knowledge proofs will be introduced together with the corresponding notations used in this thesis. Oblivious transfer is needed for the garbled circuits used in some protocols we discuss.

2.1 Commitment

The *commitment protocol* COM depicted in Figure 2.1 is a two-party protocol between a *committer* and a *receiver*. In the first message Commit, the committer commits himself to a specific value towards the receiver. In the second message called Decommit, the committer reveals the value he committed himself to to the receiver. A valid commitment protocol must guarantee, that the receiver has no access to the committed value until he gets the Decommit message. Further, the committer must not be able to change the value he committed himself to when sending the second message. COM can be implemented by applying COT discussed below or by using a dedicated protocol like the one proposed by Lindell in [Lin11].

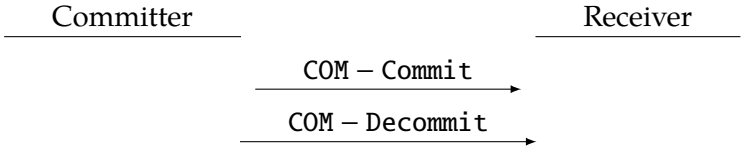

Figure 2.1: The commitment protocol COM

2.2 Oblivious Transfer

Oblivious transfer (OT) is a technique to send a message that is delivered with probability of one half. The sender of the message does not know,

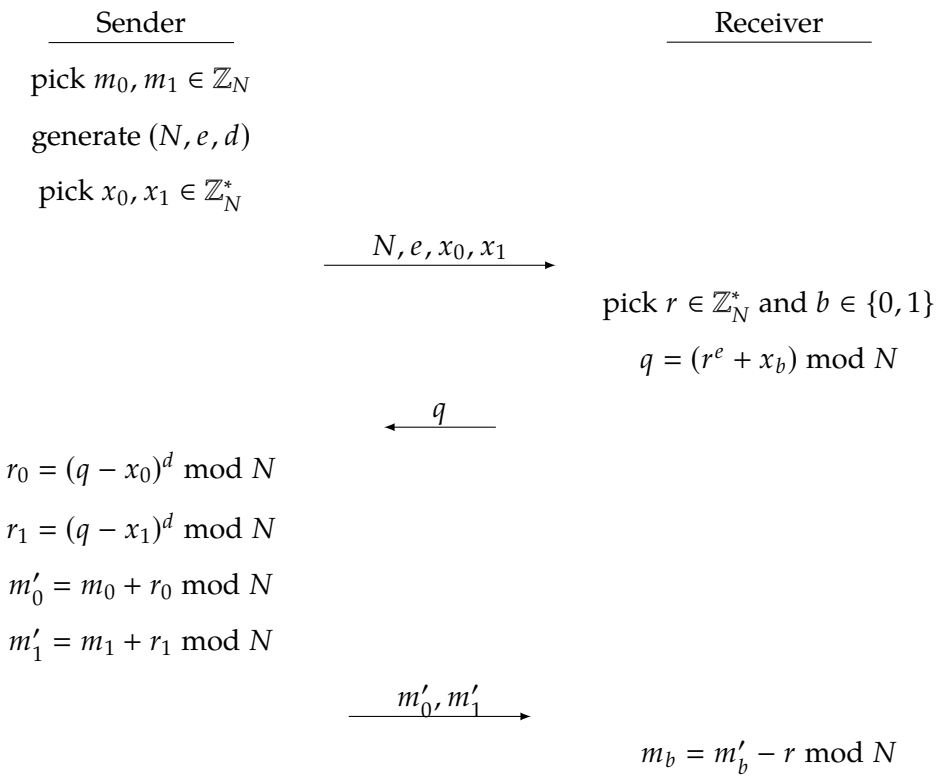

Figure 2.2: The 1-2-0T protocol by Even, Goldreich and Lempel [EGL85] is based on the RSA encryption scheme, with (N, e, d) being an RSA key pair. m_0 and m_1 are the two messages to be sent, of which the receiver gets one by choosing the index b.

whether it is delivered or not. In this thesis, we make use of a variant called *1-out-of-2 oblivious transfer* (1-2-0T). In this variant, two messages $m_0, m_1 \in \{0,1\}^\ell$ of length ℓ are sent, and the receiver has to choose which one of them is disclosed. The sender does not know, which message m_b is delivered and the receiver has no possibility to get any information about the other message m_{1-b}. The protocol consists of at least two messages: `Choose` contains the receiver's choice b and `Transfer` contains both messages m_0 and m_1 of which only m_b can be read by the receiver. The following protocol by Even, Goldreich and Lempel from 1985 [EGL85] is a simple example on how 1-2-0T can be realized (for a visualization see Figure 2.2):

Let $m_0, m_1 \in \mathbb{Z}_N$ be the two messages to be sent, and (N, e, d) an RSA

key-pair generated by the sender. In the first step, the sender sends N, e, and some random values $x_0, x_1 \in \mathbb{Z}_N^*$ to the receiver. The receiver then chooses a random number $r \in \mathbb{Z}_N^*$ and a bit $b \in \{0,1\}$ that determines which message he wants to receive. Now, the receiver responds with $q = (r^e + x_b) \bmod N$, which is the `Choose` message. From the perspective of the sender, there are two possibilities for r: It is either $r_0 = (q - x_0)^d \bmod N$ or $r_1 = (q - x_1)^d \bmod N$. Since r serves as the key for the receiver to decrypt the message, the sender encrypts one message with r_0 and the other with r_1. Hence, the `Transfer` message contains $m_0' = m_0 + r_0 \bmod N$ and $m_1' = m_1 + r_1 \bmod N$. In the last step, the receiver decrypts the requested message by computing $m_b = m_b' - r \bmod N$.

The protocol meets the requirements of 1-2-OT because the receiver chooses a bit b and encrypts it in form of q. Since there are two possibilities for r with the same probability, the sender cannot know which r_i is the right one. Hence he gets no information about which message is disclosed to the receiver. On the other hand, it is difficult for the receiver to find the other r_i because he does not know d. Therefore the receiver has no possibility to get both messages.

Note that this description differs from the original protocol. In the variant described in this section, the receiver gets to choose whether to receive m_0 or m_1. In the original variant by Even et al., the receiver does not get to choose the message because the sender chooses which of the messages to encrypt with r_0 and which with r_1. This results in the message that is obtained by the receiver being chosen at random. We altered the original protocol because for our purposes we need the receiver to be able to choose the message.

This is just a basic example for 1-2-OT that teaches us how it is possible to reach the unintuitive state that only one of two messages arrives and the sender does not know which one. Most modern protocols are more efficient and include commitment on the messages sent, as explained in the next section.

2.2.1 Committing Oblivious Transfer

The *committing oblivious transfer protocol* COT, depicted in Figure 2.3, is an extended variant of classical (1-2-)OT. Like in the standard protocol, the sender has two messages $m_0, m_1 \in \{0,1\}^\ell$. The receiver chooses between

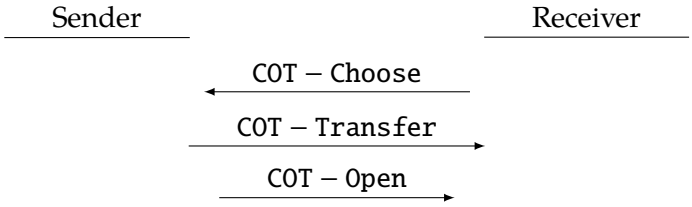

Figure 2.3: The COT protocol consists of at least three messages. In the Choose message, the receiver chooses which of two messages m_0 and m_1 to receive in the Transfer message. The Open message is used to reveal the other message that has not been altered since. Depending on the implementation, there may also be more messages.

obtaining either m_0 or m_1 while he has no possibility to get the other one and the sender does not learn which message the receiver chooses. What makes COT different is that subsequently, the sender can reveal m_0 and m_1 to the receiver without being able to alter the messages in the meantime. The messages performing these tasks are called Choose, Transfer and Open. As the name implies, Choose contains the choice b of the receiver, that cannot be extracted by the sender. Transfer contains the message m_b and Open contains both messages m_0 and m_1. There are several possibilities to implement COT. Two examples can be found in Appendix A in [JKO13].

2.3 Garbled Circuits

The concept of garbled circuits is a common technique for secure two- and multi-party computation. Introduced by Yao in 1986 [Yao86], garbled circuits have been widely adopted for several tasks due to their generic nature. Modern applications of garbled circuits include achieving secure multi-party computation with two-round protocols [GS18, BL18] and zero-knowledge proofs [JKO13].

In secure two-party computation, two parties want to know the result of a computation to which each of the parties knows a part of the input. At the same time, both parties should not get any information about the other party's input besides the information they can deduce from their own input

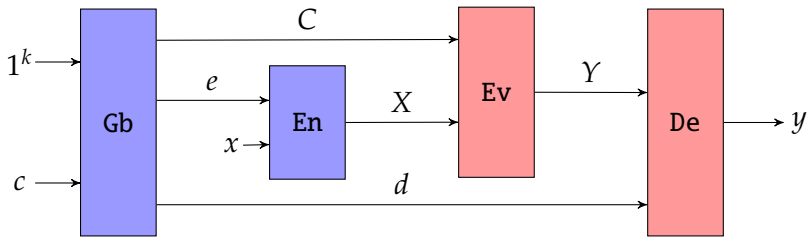

Figure 2.4: The dependencies between the components of a garbling scheme. Inspired by [BHR12]. The color coding describes membership of the algorithms. The blue parts are executed by the garbler and the red parts by the evaluator.

and the output. To reach this goal, a circuit performing the computation is used.

Let's first take a look at the idea behind garbled circuits while introducing the notation. For our definition of garbled circuits, we adapt the definition of garbling schemes by Bellare, Hoang and Rogaway [BHR12]. Garbling schemes are a generalization of garbled circuits that have the same purpose, but do not have to be realized using circuits. In contrast to Bellare et al., we explicitly use circuits, which results in a slightly different notation.

Given a Boolean circuit c computing a function $f : \{0,1\}^n \to \{0,1\}^m$, an input $x \in \{0,1\}^n$ and a unary security parameter k, the garbling scheme $\mathcal{G}$ outputs $y \in \{0,1\}^m$ while complying to several security requirements discussed below.

In a first step, one party, called the *garbler*, modifies (*garbles*) c using the probabilistic *garbling algorithm* $\mathtt{Gb}$. Its output $(C, e, d) = \mathtt{Gb}(1^k, c)$ consists of a *garbled circuit* C, an *encoding function* e, and a *decoding function* d. The functions e and d are easily invertible mappings from bits to labels. The labels for these mappings are chosen at random from $\{0,1\}^k$ to substitute the inputs in a bitwise fashion. Note that d is not the inverse function of e since e contains the information how to represent the input bits through labels while d contains the mapping from output labels to output bits. The process continues with the garbler executing the deterministic *encoding algorithm* $\mathtt{En}$, which transforms e and x into the *garbled input* $X = \mathtt{En}(e, x)$. Now the garbled circuit C, the garbled input X and the decoding function d are sent to the second party, called *evaluator*. Note that since the garbler does not know the whole input, he sends both possibilities via $\mathtt{1\text{-}2\text{-}OT}$ for

every bit that is part of the evaluators input. Since the evaluator does not know how to invert the garbling of the inputs, he is unable to get information about the garblers input. Next, the evaluator executes the deterministic *evaluation algorithm* $\texttt{Ev}$, which takes C and X to compute the *garbled output* $Y = \texttt{Ev}(C, X)$. Finally, the evaluator uses the deterministic *decoding algorithm* $\texttt{De}$ to compute the final output $y = \texttt{De}(d, Y)$ out of d and Y. An alternative possibility would be that the evaluator sends the garbled output Y back to the garbler, who then executes the decoding algorithm. In case both parties need the output, the party executing the decoding can be prevented from cheating while sharing the output, through using commitment. Altogether, $\mathcal{G}$ can be seen as a tuple $\mathcal{G} = (\texttt{Gb}, \texttt{En}, \texttt{Ev}, \texttt{De})$. The whole process is visualized in Figure 2.4. Note that none of the parties gets information about the other party's inputs: The garbler does not know for which inputs the evaluator received the input labels, because he sent them via oblivious transfer. The evaluator cannot know the garblers inputs, because he only sees the garbled version and does not know how to reverse the garbling of the inputs.

Now that we have an overview of the basic idea, let's concentrate on how the garbling works. Let the input x consist of n bits x_p with $p \in \{1, \ldots, n\}$. For every single input position p, there is a label l_p^0 for the bit 0 and another label l_p^1 for the bit 1. In the following, we will refer to l_p^0 and l_p^1 as a pair of labels. These labels are chosen randomly and independently from each other from $\{0,1\}^k$. Now, the input can be represented through $\{(p, l_p^{x_p}) \mid 0 < p \leq n\}$ which contains one label for each position. The circuit is garbled gate by gate. Like the input bits, all wires between the gates are represented through pairs of labels from $\{0,1\}^k$. So each gate's input are two input labels l_a and l_b and the output is represented through an output label l_c. For each gate, there are four possible combinations of input labels $(l_a^0, l_b^0), (l_a^0, l_b^1), (l_a^1, l_b^0), (l_a^1, l_b^1)$, each corresponding to a particular output label. To garble a gate, for each of these combinations the output label is encrypted using the input labels as key. So for example for an AND gate there would be the ciphertexts $\texttt{Enc}_{l_a^0, l_b^0}(l_c^0)$, $\texttt{Enc}_{l_a^0, l_b^1}(l_c^0)$, $\texttt{Enc}_{l_a^1, l_b^0}(l_c^0)$ and $\texttt{Enc}_{l_a^1, l_b^1}(l_c^1)$. This approach ensures that the evaluator is able to decrypt exactly one ciphertext per gate, obtaining the output label belonging to the correct output. If it was possible to decrypt more than one ciphertext per gate, the evaluator could gain information about the input. On top of that, the evaluator could manipulate the outcome of the evaluation through choosing which output label to proceed with. To encrypt the output labels,

a so-called *key-derivation-function* $H : \{0,1\}^* \to \{0,1\}^k$ is used. H is a pseudorandom function that generates binary strings of length k given binary seeds of arbitrary length. So $||$ being the concatenation of two strings and $\oplus$ being an XOR, an output label is encrypted by $\mathtt{Enc}_{l_a,l_b}(l_c) = H(l_a||l_b) \oplus l_c$.

To evaluate the circuit, the evaluator iterates through all gates, decrypting the right ciphertext for each gate using the gate's input labels as key. When the evaluation is complete, the evaluator knows the labels to all of the output bits, but does not know, to which values they correspond.

Up to here, there is no difference between different types of gates. All 16 possible gates with two input bits and one output bit are garbled the same way. Later on, some improvements of garbled circuits will be explained. Some of these improvements use the different nature of the gate types to improve the efficiency of their garbling. To prepare this, let's take a look at how different gates behave in garbled circuits: First of all, there is no difference between garbling a gate and garbling its negation. An AND gate is the same as a NAND gate, an OR gate is the same as a NOR gate, and so on. The only difference is that the values of the output labels are swapped. Therefore, both gates are garbled the same way, but when continuing with the next layer of gates, the output labels are used differently. Besides AND, NAND, OR and NOR, there are four more uneven gates. All these gates are garbled the same way: Three input label combinations encrypt the same output label, while the last input label combination encrypts the other output label. Because of this, there is no difference between garbling different uneven gates, even using the improvements discussed below. Even gates behave different to uneven gates. The important even gates are XOR and XNOR. All other six even gates are trivial and will not appear in any circuit. Like explained above, XOR behaves just like its negation. From now on, to simplify the explanations, we will abstract and explain everything using only AND, representing the uneven gates, and XOR, representing itself and XNOR.

It is important to know that garbled circuits must not be used multiple times with different inputs. If this is done, participants may have the possibility to gain information about the other party's input. Let there be an AND gate with input wires a and b, with a being a part of the garblers input. If the evaluation is done twice, once with l_b^0 and once with l_b^1, the evaluator is able to decrypt two ciphertexts instead of one. If both ciphertexts reveal the same output label, the evaluator knows that $a = 0$. If they reveal different output labels, it is clear that $a = 1$. XOR gates

are different: Evaluated with different labels for b, they will always give different output labels, revealing no information about a. But since the evaluator gets two different output labels, he can proceed with the next gate the same way until he gets to the next AND gate. The more differences there are between the evaluations, the more information the evaluator can get. But, since two evaluations with different inputs may already reveal more than half of the garbler's input (depending on the circuit), one should always restrict garbled circuits to single use.

2.3.1 Properties of Garbling Schemes

There are several properties that garbling schemes may fulfill or not fulfill. In the following, correctness, privacy, obliviousness, authenticity and verifiability will be introduced. Most of them are security notions, but there is also another very fundamental property: the correctness.

For a garbling scheme to be *correct*, we require $y = f(x)$, meaning that $\mathtt{De}(d, \mathtt{Ev}(C, \mathtt{En}(e, x))) = f(x)$ with $(C, e, d) = \mathtt{Gb}(1^k, c)$. So if all algorithms are executed correctly, the output obtained is the correct output of the function f with input x. Definitions can be found in [BHR12, JKO13, FNO15, Zho16].

Further, a garbling scheme is called *private* if the evaluator obtaining (C, X, d) is not able to learn anything about x other than what is revealed by y and f. An exact definition can be found in [BHR12].

Another security notion is *obliviousness*. A garbling scheme is called *oblivious* if from the garbled circuit C and the garbled input X nothing can be deduced about the function f, the input x or the output y. This property is similar to privacy, but neither of them implies the other. An exact definition can be found in [BHR12].

If a garbling scheme fulfills *authenticity*, no set of output labels can be found besides the correct one. This property is related to the correctness property, but includes that the evaluator does not have to stick to the protocol. Different definitions can be found in [BHR12, JKO13, FNO15].

The last property we present here is the *verifiability*. It states that if there are multiple inputs with the same output, one cannot deduce from the output labels which input led to the output. Hence the output labels leak no information about the input [JKO13, FNO15].

2.3.2 Improvements

Although initially seen as a purely theoretical tool because of the high communication and computational complexity, numerous improvements have made garbled circuits the versatile and efficient technique we use today. The most important early improvements are *point-and-permute* [BMR90], *row reduction* [NPS99] and *free-XOR* [KS08] which reduce the number of ciphertexts needed and/or evaluated per gate. Recently garbled circuits have been further improved by techniques called *GRR2* [PSSW09], *fleXOR* [KMR14] and *half-gates* [ZRE15]. For situations in which the circuit evaluator is allowed to know the plaintext input to the garbled circuit (*privacy-free garbling*, [FNO15]), the computational and communication overhead of garbled circuits has been further reduced [ZRE15]. An overview of the improvements done is given in Table 2.1. Besides these technical improvements, the conceptual improvements done by Bellare et al. [BHR12] are worth mentioning. Instead of using garbled circuits merely as a tool, the authors define the parts and properties of garbled circuits in a mathematical, precise way.

The *point-and-permute technique* introduced by Beaver, Micali and Rogaway [BMR90] helps the evaluator to determine which ciphertext to decrypt. In the standard garbled circuit protocol, the evaluator receives four ciphertexts per gate, one of which he can decrypt. Beaver et al. used one of the bits in each label to indicate which is the correct ciphertext so the evaluator does not have to try multiple ones. This so-called *selection bit* or *permutation bit* is not an additional bit, but just the bit at a fixed position in all labels. It is generated like the rest of the label, so either from some input labels to a gate (see further improvements) or at random for the circuit's input. In every pair of labels, one label must have the permutation bit 1 and the other one must have the permutation bit 0. For the circuits input, this reduces the randomness of the label generation, in the following generation of the circuit this is done automatically (see further improvements). So in a gate with two input wires, all four combinations of permutation bits may occur. Note that the combination of input values and the combination of permutation bits are independent from each other. Hence the evaluator cannot deduce information about the input values from the permutation bits. With this technique, the order in which the ciphertexts are sent can be inferred from the combination of permutation bits in the input labels used as key; for example the ciphertext with permutation bits 00 is sent first, the one with 01 is sent second, and so on. This way, the evaluator

Table 2.1: Overview of improvements made to classical garbled circuits. Inspired by tables 1 and 2 in [ZRE15]. H is a hash function used by `Gb` and `Ev`. The number of H-calls indicates the performance of these algorithms. In fleXOR the sizes of XOR gates and number of H-calls per XOR gate vary, depending on the structure of the circuit c. Privacy free refers to privacy-free garbling situations.

	size per gate		H-calls per gate			
			garbler		evaluator	
	XOR	AND	XOR	AND	XOR	AND
classical gc [Yao86]	4	4	4	4	4	4
point-and-permute [BMR90]	4	4	4	4	1	1
row reduction [NPS99]	3	3	4	4	1	1
free-XOR [KS08]	0	3	0	4	0	1
fleXOR [KMR14]	$\{0,1,2\}$	2	$\{0,2,4\}$	4	$\{0,1,2\}$	1
half-gates [ZRE15]	0	2	0	4	0	2
half-gates (privacy free) [ZRE15]	0	1	0	2	0	1

knows which ciphertext to decrypt by looking at the permutation bits of the gates input labels.

The *row reduction* by Naor, Pinkas and Sumner [NPS99] reduces the number of ciphertexts needed per gate. They proposed to compute one of the output-labels of each gate as a function of the inputs. Combined with the point-and-permute technique, one could spare e.g. the first row which could be the ciphertext with both zero permutation bits of each gate. Instead, the output label would be computed from the input labels by a given function. This way, only three ciphertexts need to be created, which reduces the size of the garbled circuit.

Using the *free-XOR technique* by Kolesnikov and Schneider [KS08] XOR gates can be evaluated without encrypting or decrypting any ciphertext. For this, a global offset R is needed. This is the offset between two labels for the same wire, so one could xor R to one label to get the other label. If this offset is the same for the whole circuit, this leads to a big advantage: XOR gates can simply be computed by xoring the input labels. The resulting labels will have the offset R, because both input-label-pairs had this offset. This technique is compatible to both point-and-permute and row reduction.

The *GRR2 technique* by Pinkas et al. [PSSW09] uses polynomial interpolation to spare one more ciphertext than standard row reduction. Hence only two ciphertexts per AND gate are needed. Unfortunately, this is not compatible with free-XOR, so that XOR gates also need two ciphertexts per gate. Kolesnikov et al. found a trade-off between free-XOR and GRR2 which is called *fleXOR* [KMR14]. XOR gates need up to two ciphertexts, but in many cases only one or no ciphertext is needed. In exchange, GRR2 can be used without limitation. The reason why free-XOR cannot be used with GRR2 is that when using GRR2, not all label-pairs have the same offset. Using the fleXOR technique, the number of needed ciphertexts depends on how many label-pairs per gate have the same offset. When the offsets of the input labels of an XOR gate are different, it uses a strategy to convert the offset of the output labels into a third offset chosen by the garbler. If this third offset is one of the offsets from the input, only one ciphertext is needed. At all XOR gates where the offsets of the inputs are the same, free-XOR can be used so no ciphertexts are needed.

Using the *half-gates technique* by Zahur, Rosulek and Evans [ZRE15], even fewer ciphertexts are needed. Like with fleXOR, only two ciphertexts per

AND gate are needed. Additionally, half-gates are fully compatible to free-XOR, therefore no ciphertexts are needed for XOR gates. The name half-gates comes from the idea to split AND gates in two halves: an evaluator half-gate and a generator half-gate. A half-gate is an AND gate, with the value of one of the input wires being known to one of the participants. Due to a clever splitting of the AND gate, the evaluator knows one input for the evaluator half-gate and the garbler knows one input for the generator half-gate. Thanks to this knowledge, each half-gate can be garbled using only one ciphertext. In the evaluation process, the result of the AND gate is computed by xoring the results of the half-gates. As a result, two ciphertexts are needed for each AND gate.

When no privacy is needed because the evaluator is allowed to know all the inputs, *privacy-free garbling* by Frederiksen, Nielsen and Orlandi [FNO15] can be used. Although it seems like there is not much use for a garbled circuit if the evaluator knows all the inputs, there are some use cases. For example the evaluator can be prevented from manipulating a calculation he has to do [JKO13]. We also found an encryption scheme in which the evaluator is the sender and therefore knows all inputs (see `MWBES` in Section 3.3). Frederiksen et al. found a way to let the evaluator use his knowledge of the inputs to need fewer ciphertexts. In the standard case, an AND gate with inputs a and b and output c has three ciphertexts $\mathtt{Enc}_{l_a^0,l_b^0}(l_c^0)$, $\mathtt{Enc}_{l_a^0,l_b^1}(l_c^0)$, $\mathtt{Enc}_{l_a^1,l_b^0}(l_c^0)$ that all encrypt the same output label. It is sufficient to encrypt l_c^0 once using l_a^0 and once using l_b^0, sparing one ciphertext. Since the evaluator knows the values which the labels stand for, he can pick the correct ciphertext to decrypt. The fourth ciphertext $\mathtt{Enc}_{l_a^1,l_b^1}(l_c^1)$ can be spared by choosing $l_c^1 = H(l_a^1||l_b^1)$. This leads to two ciphertexts per AND gate and no ciphertexts for XOR gates. This is the default version, compatible to free-XOR, listed in Table 2.1. In an alternative version, another ciphertext can be spared, if free-XOR is not used. In this case, l_c^0 can be chosen independently from l_c^1. So when l_c^0 is chosen as $H(l_a^0)$, only one ciphertext $\mathtt{Enc}_{l_b^0}(l_c^0)$ remains. Not using free-XOR, the XOR gates are garbled using one ciphertext: When $l_c^0 = l_a^0 \oplus l_b^0$ and $l_c^1 = l_a^0 \oplus l_b^1$, it suffices to send $l_a^0 \oplus l_a^1 \oplus l_b^0 \oplus l_b^1$ as only ciphertext. If the evaluator has l_a^1 and l_b^1 or l_a^1 and l_b^0 as input labels, he xors with the ciphertext after xoring the input labels. Not using free-XOR leads to one ciphertext per AND gate and one ciphertext per XOR gate. Whether or not it is of advantage to use free-XOR depends on which gate appears more frequently in the circuit.

Zahur, Rosulek and Evans [ZRE15] found a way to combine the privacy-free

technique with half-gates. Since in the privacy-free model, the evaluator knows all of the input values, AND gates can be handled directly as evaluator half-gates and do not have to be split. This results in a strategy that uses only one ciphertext per AND gate and none for XOR gates. But, like the privacy-free approach, this strategy may only be applied if the evaluator is allowed to know all inputs. In most scenarios, this is not the case.

Altogether, to garble circuits the most space-saving way known today, the half-gates technique in combination with free-XOR and point-and-permute should be used. In a scenario where privacy is not needed, one should use the privacy-free variant of half-gates which is even more space-saving but unfortunately is not compatible when privacy is needed.

2.4 Zero-Knowledge Proofs

Zero-knowledge proofs are designed to enable a prover to prove some statement to a verifier without revealing any information about the witness. For example there might be a one-way function f for which the prover knows a pair of input $\hat{x}$ and output $\hat{y}$. He could then use a zero-knowledge proof to prove that he knows an $\hat{x}$, such that $f(\hat{x}) = \hat{y}$, without revealing $\hat{x}$ to the verifier.

Initially, both parties agree on a statement s to be proven. The role of the verifier is to verify whether the prover possesses a valid witness w, for example by sending challenges the prover has to respond to correctly. In our example, the statement consists of f and $\hat{y}$, while the witness is the input $\hat{x}$.

Zero-knowledge proofs are often used for authentication. There is a public one-way function f, which each participant has an input to: his private key. The corresponding outputs (public keys) are stored in a public database that cannot be manipulated. To authenticate oneself, one has to prove knowing the private key corresponding to the public key stored in the database. Another possibility would be that the prover owns a trapdoor function to which he knows the secret key. To authenticate himself, he has to provide the input corresponding to an output chosen by the verifier. While the second scenario is simple to realize because the prover can simply compute the inverse trapdoor function and respond with the input, it is more difficult to accomplish the first scenario. The problem is to make sure the verifier does not get any information about the private key, because

otherwise he could authenticate himself as the prover. To overcome this problem, zero-knowledge proofs can be used.

2.4.1 Security Notions

For a protocol to be a zero-knowledge proof, specific properties have to be fulfilled. First, if the prover actually knows $\hat{x}$, he should be able to prove this knowledge to the verifier. The probability that he does not succeed to convince the verifier should be negligible. This property is called *completeness*.

Next, it is important that if the prover does not know $\hat{x}$, he cannot convince the verifier to accept. If *soundness* is fulfilled, the probability that a prover not knowing $\hat{x}$ convinces the verifier is negligible.

The last property needed is the *zero-knowledge property*, stating that the participants do not obtain any information that cannot be deduced from the statement to be proven besides the correctness of the statement. Since the only information to be kept secret is the prover's input $\hat{x}$, it suffices to prove that the verifier cannot get any information about $\hat{x}$. This can be done by showing the existence of a simulator S that knows $\hat{y}$ but not $\hat{x}$. Additionally, S is allowed to query an oracle that samples challenges with the same distribution as a verifier. If such a simulator is able to produce an output that is indistinguishable from an honest prover's output, the verifier cannot learn anything about $\hat{x}$ from the protocol.

3 White-Box Cryptography

White-Box cryptography was first introduced by Chow et al. in 2002 [CEJVO02]. Its idea is that an attacker is not limited to observe programs as a black-box. Instead he has the ability to look into the system, try to decompile it and to analyze its code. So the main goal of white-box cryptography is to prevent such an attacker from extracting important information from the program although he has full access to it.

An important use-case is software that gives users limited access to data. Examples are streaming-platforms for music and video, gaming platforms and licensed productivity software. All these should give the user access to a particular amount of data, sometimes for a limited amount of time, without letting him extract or copy the data. Because these programs are installed on the users hardware to which he has unlimited access, an attacker has so-called white-box access to them.

A white-box could also be an implementation of an encoding or decoding algorithm with an embedded key. So whoever owns the white-box is able to use it to encrypt or decrypt messages but not to extract the key. Delerablée et al. defined the most important security notions for white-box cryptography [DLPR13]:

As mentioned above, there is some information hidden in the white-box, in many cases some kind of key, that can be used but not extracted by the user. Since the main use of the white-box is to provide the usage of this information, the most important security notion is to protect it against an attacker. This security notion is called *unbreakability*. If unbreakability cannot be granted, the white-box is useless.

Another important security notion that is related to unbreakability but a bit weaker is the *incompressibility*. It should be given that an attacker is not able to transform the white-box into a tinier version that takes less space and is still able to provide the usage of the crucial information. If this is not given, it is easier for an attacker to copy and share the white-box with other individuals.

But even if these two notions are given, it is still possible to misuse the white-box in another way: If the information guarded is the key to an encryption algorithm performed by the white-box, an attackers goal could be to transform the encryption algorithm into a decryption algorithm without having to extract the key. Like this he could decrypt messages he only is allowed to encrypt. The *one-wayness* demands that the functionality of the white-box is not invertible, hence an attack as described above would not be possible.

All in all, these security notions are useful because they prevent the attacker from using the white-box in another way than it is supposed to be used. But still, they cannot prevent the attacker from simply making a copy of the white-box and sharing it with other individuals. For this reason, there exists a notion called *traceability*. From every output of the white-box, it should be possible to determine the owner of the white-box. This can be done by embedding some individual watermark into the white-box that is applied to each output. So if an attacker shares data illegally and it spreads over the internet, it should be possible to retrace the way back to the attacker. Furthermore, multiple attackers that consolidate should not be able to remove all traces through combination of their versions of the data. At least one trace should always remain.

Additionally to the security notions presented by Delerablée et al., one could require the white-box to be locked by password. So if the box is stolen, it cannot be used unless the thief knows the password, too.

The main approach on how to realize white-boxes is using obfuscation. In the paper in which Chow et al. first introduced white-box cryptography, they present an AES implementation that is designed to be secure against an attacker with white-box access. In 2004, Billet, Gilbert and Ech-Chatbi proposed the BGE-attack that breaks this implementation [BGEC04]. The implementation by Chow et al. makes use of lookup tables for different parts of the algorithm. Through combination of those tables, it becomes more and more difficult to extract the embedded key, that is integrated into one of the tables. By combining multiple small steps to fewer larger steps, the execution of the algorithm is obfuscated. It is hard for the attacker to separate the steps and reconstruct them one-by-one.

3.1 Applying Garbled Circuits to White-Box Cryptography

Since the user of a white-box is not allowed to know the secret information hidden in it, in most cases there is another party that constructed the white-box. The use of a white-box may remind the keen reader of secure two-party computation: Both parties have some input while the user is not allowed to know the constructor's input. So the constructor hides his input in the white-box, which is used by the user to calculate the output to some input. For a visualization see Figure 3.1. Like most white-boxes, garbled circuits use obfuscation. Replacing each bit by a label, it is nearly impossible for the evaluator to reconstruct the values of the bits to understand the computation he is executing. The only differences are that in white-box cryptography the constructor does not need the output and may in some cases be allowed to know the users inputs.

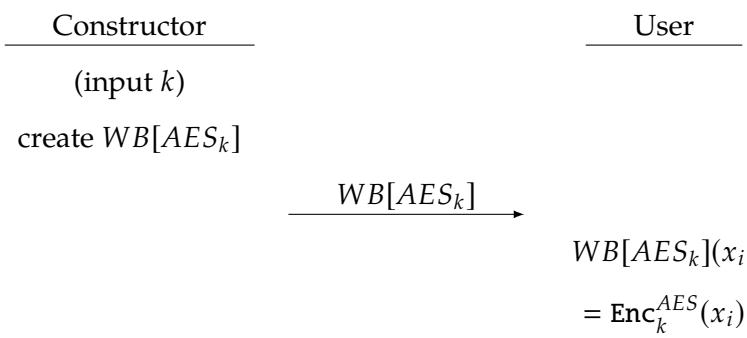

Figure 3.1: Example of a constructor providing a white-box to a user. The constructor creates a white-box for AES encryption and embeds his key k into it. The user can then use the white-box to encrypt messages x_i. The white-box may be used multiple times, but the user gets no information about the key k except what can be deduced from the input-output-pairs.

Due to this resemblance, the idea arose to use garbled circuits for white-box cryptography. In this scenario, the constructor is the garbler and the user is the evaluator. The biggest problem is that garbled circuits may only be used once. Meaning that, if the user of the white-box used it multiple times with different inputs, it would be possible to calculate at least parts of the secret hidden information, as described in Section 2.3. Fortunately, the

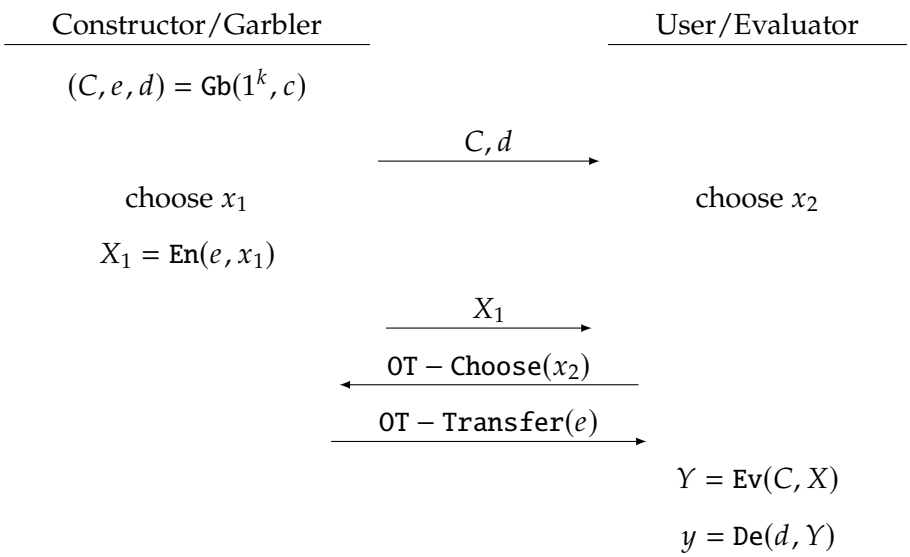

Figure 3.2: Single-use white-box using a garbled circuit. At the end the user gets the output y which is a function of the inputs x_1 and x_2 without learning the creator's input x_1.

number of uses is controlled by the garbler who needs to send the garbled inputs via 1-2-OT. As long as the garbler sends the input labels only once, there is no possibility for the evaluator to get the hidden information. This way, it is possible to create a single-use white-box using a garbled circuit. In Figure 3.2, the protocol for this single-use white-box is presented. As an alternative to this protocol, it is also possible to send C and d together with X_1. Only those parts of $\mathtt{OT\text{-}Transfer}(e)$ that contain the labels X_2 can be read by the receiver. If the creator is indeed allowed to know the user's input, the OT could be dropped such that the user simply sends x_2 and the creator responds with X_2.

Another way to prevent the evaluator from evaluating the circuit multiple times with different inputs would be to let the garbler choose all of the inputs. This is the approach we are going to pursue further. It has less of an interactive white-box, but is a way to send encrypted messages from garbler to evaluator. We basically have the garbler choosing a circuit and all inputs and then garbling it and sending it to the evaluator. The evaluator has no influence on what he receives from the garbler but at the end he gets to decrypt the output that is some message from the garbler that only the

owner of the white-box can decrypt. An advantage of such an encryption scheme is that the garbled circuit may be sent before the garbler decides which message to sent. This is of great value because the garbling is the biggest part of the computational complexity and the garbled circuit is the biggest part of the communication complexity. So when the garbler decides which message to send, the rest of the protocol can be executed very quickly because the majority of the work has been done in advance. Also, there is no need for `OT` anymore, which saves a lot of resources.

3.2 Simple White-Box Encryption Scheme

The simple white-box encryption scheme (`SWBES`) visualized in figure 3.3 is an implementation of the approach explained above. First, the circuit c and the security parameter k are chosen by one of the parties. It is more practical to let the garbler choose, so the parameters do not have to be sent. The circuit c should compute the decryption algorithm of a symmetric encryption scheme $\mathcal{E}$ with a key z. This key can either be hard-coded into the circuit, or be chosen later and be a part of the input. With this information, the sender is already able to execute the garbling algorithm `Gb`. Now he can send the garbled circuit C and the decoding function d over to the receiver, although he did not yet choose the message to send. When the sender has chosen a message, he encrypts it according to the symmetric encryption scheme $\mathcal{E}$ with key z. The result is the input x which contains either just the ciphertext, or both ciphertext and key if the key is not hard-coded into the circuit. After garbling the input x using the encoding function e, he sends the garbled input X to the receiver. The receiver is now able to first evaluate the circuit C on the garbled input X and then decode the garbled output Y using the decoding function d.

In this scenario, the sender can choose m and send X quite a while after sending C and d. Alternatively X, C and d could be sent together if the sender already knows which message to send. This has the disadvantage that an third-party attacker has to eavesdrop only one message to be able to compute m. Since all three parameters are needed to get m, splitting the messages could make it harder for an attacker to overhear them all. Like in a symmetric encryption scheme, it would seem useful to send C and d or at least one of them over a different channel, if available a secure channel. But while in a symmetric encryption scheme the key is used over and over again, the circuit may only be used once. Hence it would not be worth it to

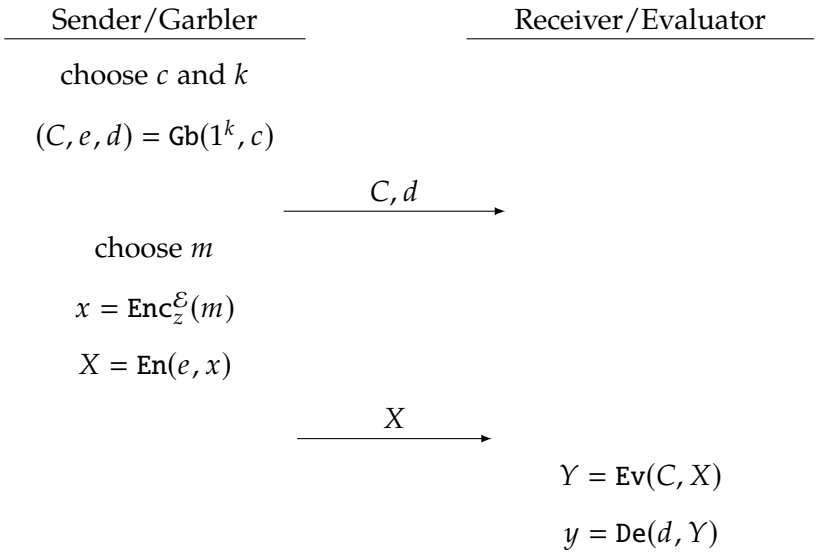

Figure 3.3: Simple white-box encryption scheme (SWBES). In this visualization, the key z is integrated into the circuit c. Alternatively it could be a part of the input x which would then consist of ciphertext and key. If the garbling scheme is correct and the protocol is executed correctly, then $y = m$.

send one message over a secure channel, protecting just one other message sent over a public channel. One could send multiple garbled circuits over the secure channel to be able to send messages at a later point, but this would be the same strategy as sending codebooks for one-time-pad over a secure channel. But since in one-time-pad the keys are not longer than the messages, SWBES has more overhead.

All in all, this simple white-box encryption scheme has some disadvantages like the restriction to one-time use and the rather big communication overhead. But it also has some important advantages like the fact that OT is not necessary and that the receiver does not learn anything about the ciphertext. These advantages make it possible to transform the SWBES into a zero-knowledge proof protocol using garbled circuits (see Section 4.2). This protocol is a whole new approach to zero-knowledge proofs, beside other advantages making it possible to use garbled circuits without any need of oblivious transfer or commitment.

3.2.1 Security

Since there is only unidirectional communication from sender to receiver, there is no possibility for the sender to get any additional information. So let's take a look at what the receiver can learn. First of all, the receiver is not able to influence the server because he does not send anything. Naturally, he learns C, d and X and is able to compute Y and $y = m$. Additionally, he is able to determine k from C or d and needs to know which gates of c are even gates for the evaluation. But is it possible for him to get information about e, $\mathcal{E}$, z or x? Since he has some information about c, it could be possible to determine $\mathcal{E}$ or at least make a good guess. Regarding the other parameters, it is not possible to learn something about them if the garbling scheme is private. Multiple execution of the protocol does not give any advantage for the extraction of x because for each execution the circuit is garbled anew.

Since the knowledge of C, d and X suffices to compute y, a third-party attacker is able to obtain m if all messages are intercepted.

3.2.2 Computational and Communication Complexity

In the following, the computational complexity and communication complexity of SWBES is analyzed. For this analysis, we use the best compatible improvements, which are half-gates, free-xor and point-and-permute. In a second analysis, we compare our results to using only the basic improvements free-XOR and point-and-permute.

We choose f to be AES-128, because it is a sufficiently secure encryption scheme often used to compare white-box schemes. Using another encryption scheme than AES-128, the complexity will differ from the results presented in this section. We use the data from [PSSW09], so the circuit c that implements f has 33880 gates of which 11286 are uneven gates and 22594 are even gates.

The sender has to garble the circuit, encrypt the message and compute the input labels. Let's begin with the complexity of the garbling. First, the offset R between each pair of labels is generated. Then, for each input bit one label is created, resulting in 128 labels because f has 128 input bits. The other 128 labels are computed xoring each of the first 128 labels to R. Next, the gates are garbled one by one. For each uneven gate up to six XORs are performed, and the hash function H is calculated four times. Even gates are simpler since there are only two XOR operations computing both output labels. Let $\texttt{XOR}(k)$ be the time needed to compute an XOR on k bit, and $\texttt{gen}(k)$ be the time needed to generate a k bit label, and $\texttt{H}(k)$ be the time needed to compute the hash function H on k bit. Then the time needed for the garbling is computed as:

$$\begin{aligned} T(\texttt{Gb}) =& 129 \cdot \texttt{gen}(k) + 128 \cdot \texttt{XOR}(k) \\ &+ 11286 \cdot (6 \cdot \texttt{XOR}(k) + 4 \cdot \texttt{H}(k)) \\ &+ 22594 \cdot 2 \cdot \texttt{XOR}(k) \\ =& 129 \cdot \texttt{gen}(k) \\ &+ (11286 \cdot 4) \cdot \texttt{H}(k) \\ &+ (128 + 11286 \cdot 6 + 22594 \cdot 2) \cdot \texttt{XOR}(k) \\ =& 129 \cdot \texttt{gen}(k) + 45144 \cdot \texttt{H}(k) + 113032 \cdot \texttt{XOR}(k) \end{aligned}$$

Additional to the garbling, the sender encrypts m using $\mathcal{E}$ and z. For the encoding of the input, he only has to look up the 128 labels in e.

Let's continue with the computational complexity on the evaluators side. For the evaluation of the circuit, the evaluator also proceeds gate by gate.

Since free-XOR is used, there is only one XOR operation per even gate. For all uneven gates, two half-gates are evaluated. Both half-gates include up to one XOR and one execution of H. One additional XOR per uneven gate is needed to combine the half-gates and compute the output. The time needed for the evaluation is computed as:

$$\begin{aligned} T(\texttt{Ev}) =& 22594 \cdot \texttt{XOR}(k) + 11286 \cdot (2 \cdot \texttt{H}(k) + 3 \cdot \texttt{XOR}(k)) \\ =& (22594 + 11286 \cdot 3) \cdot \texttt{XOR}(k) + (11286 \cdot 2) \cdot \texttt{H}(k) \\ =& 56452 \cdot \texttt{XOR}(k) + 22572 \cdot \texttt{H}(k) \end{aligned}$$

Additionally, the evaluator decodes the output labels which includes 128 lookups in d.

Next we analyze the communication complexity: The garbled circuit C consists of 11286 uneven gates, each represented by two ciphertexts of length k, resulting in $22572 \cdot k$ bit. For the decoding, only one label per bit needs to be sent, for example $\{l_p^1 \mid 0 < p \leq 128\}$. By comparison with the output labels, it can be determined whether the bit's value is 1 (match) or 0 (mismatch). This results in $128 \cdot k$ bits for d. Another $128 \cdot k$ bits are needed to send the input labels X. The overall communication complexity for one execution of the protocol is $22828 \cdot k$ bit which is about $2.79 \cdot k$ KiB.

For comparison, we will compute the computational and communication complexity again, using only the most basic improvements, namely free-XOR and point-and-permute.

On the senders side, the only thing that changes is the complexity of the garbling while the complexity of encryption and encoding stay the same. Again, because we still use free-XOR, the offset R between each pair of labels is generated. The 256 input labels are generated the same way, too. Now for the garbling: To garble an uneven gate, the garbler generates a pair of output labels by first generating one label and then xoring R to it to compute the other one. Next, one ciphertext is created for each input combination, encrypting the corresponding output label. For each ciphertext, one XOR and one execution of H is needed. The even gates are handled like above, so both output labels are computed by xoring the input

Table 3.1: Number of operations needed in SWBES using AES-128 as f, once using the half-gates improvement together with free-XOR and point-and-permute and once using only free-XOR and point-and-permute. # generations denotes the number of generations of labels of length k.

	w/o half-gates		with half-gates	
	garbling	evaluation	garbling	evaluation
# generations	11415	–	129	–
# H-calls	45144	11286	45144	22572
# XORs	101746	33880	113032	56452

labels. In this case, the time needed for the garbling is computed as:

$$\begin{aligned}
T(\texttt{Gb}) =& 129 \cdot \texttt{gen}(k) + 128 \cdot \texttt{XOR}(k) \\
&+ 11286 \cdot (\texttt{gen}(k) + 5 \cdot \texttt{XOR}(k) + 4 \cdot \texttt{H}(k)) \\
&+ 22594 \cdot 2 \cdot \texttt{XOR}(k) \\
=& (129 + 11286) \cdot \texttt{gen}(k) \\
&+ (11286 \cdot 4) \cdot \texttt{H}(k) \\
&+ (128 + 11286 \cdot 5 + 22594 \cdot 2) \cdot \texttt{XOR}(k) \\
=& 11415 \cdot \texttt{gen}(k) + 45144 \cdot \texttt{H}(k) + 101746 \cdot \texttt{XOR}(k)
\end{aligned}$$

The receiver does the evaluation and the decoding of the output labels. The decoding again consists of 128 lookups in d. For the evaluation of each uneven gate, one ciphertext is decrypted by computing H on the input labels and then xoring it to the ciphertext. Even gates are evaluated by simply xoring the input labels. So the time needed for the evaluation is computed as:

$$\begin{aligned}
T(\texttt{Ev}) =& 22594 \cdot \texttt{XOR}(k) + 11286 \cdot (\texttt{XOR}(k) + \texttt{H}(k)) \\
=& (22594 + 11286) \cdot \texttt{XOR}(k) + 11286 \cdot \texttt{H}(k) \\
=& 33880 \cdot \texttt{XOR}(k) + 11286 \cdot \texttt{H}(k)
\end{aligned}$$

The communication complexity increases because four ciphertexts per uneven gate are needed resulting in $45144 \cdot k$ bit. The decoding function d and the input labels x still need $128 \cdot k$ bit each. This results in an overall communication complexity of $45400 \cdot k$ bit which is about $5.54 \cdot k$ KiB.

Comparing the results of our analysis (see Table 3.1 for overview), it can be observed that the computational complexity using half-gates is higher

than using only free-XOR and point-and-permute. For the garbling, the number of XORs increases from 101745 to 113032. For the evaluation, the number of XORs increases from 33880 to 56452 and the number of H-calls doubles from 11286 to 22572. In exchange there are fewer generations of random labels because all output labels of gates depend on their input labels. At the same time, the communication complexity is nearly halved and decreases from $45400 \cdot k$ bit to $22828 \cdot k$ bit. In summary, it can be said that to achieve a lower communication complexity one has to accept a higher computational complexity.

3.3 Modified White-Box Encryption Scheme

The modified white-box encryption scheme (`MWBES`) is a variant of the simple white-box encryption scheme where sender and receiver swap roles so that the receiver garbles the circuit and the sender evaluates it. This change of roles is inspired by the zero-knowledge protocol by Jawurek et al. [JKO13] discussed in Section 4.1.1.

There are two approaches to `MWBES`. The first one, called `MWBES1`, is visualized in Figure 3.4. Like in `SWBES`, c and k are chosen by one of the parties, for example by the receiver. Letting the garbler choose, has the advantage that the parameters do not have to be sent. In the first step, the receiver garbles the circuit so he can send C to the sender. Now the sender chooses the message m and encrypts it using a symmetric encryption scheme $\mathcal{E}$ and a key z. For z, there are two possibilities. It is either hard-coded into the circuit, or a part of the input x. If z is hard-coded into the circuit, the receiver knows it because he is the garbler. If z is a part of the input, it may be chosen by the sender and thus stay secret. In Figure 3.4, the latter possibility is depicted. Since the sender does not know e, he gets the corresponding garbled input X via `1-2-OT`. Knowing C and X, he is able to evaluate the circuit. Next the garbled output Y is sent to the receiver who is able to decode it using d obtaining the message $y = m$.

In `MWBES2`, visualized in Figure 3.5, the circuit computes an encryption instead of a decryption. In this case, the receiver chooses the key z and embeds it into the circuit. The sender chooses the message m, which serves as input x. Then, he gets the labels via `1-2-OT` and evaluates the circuit. There is no encryption step needed, because the encryption is done in the evaluation step. After the evaluation, there are two possibilities how

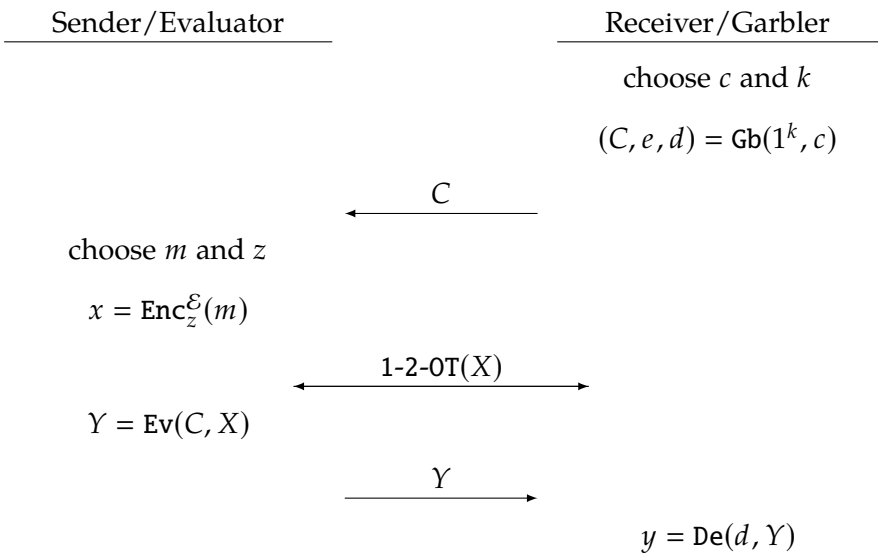

Figure 3.4: Modified white-box encryption scheme (MWBES1). In this modified white-box encryption scheme, the circuit c computes the decryption of an encryption scheme $\mathcal{E}$. So when the sender evaluates the circuit, he decrypts the ciphertext he encrypted beforehand. If the garbling scheme is correct and the protocol is executed correctly, then $y = m$.

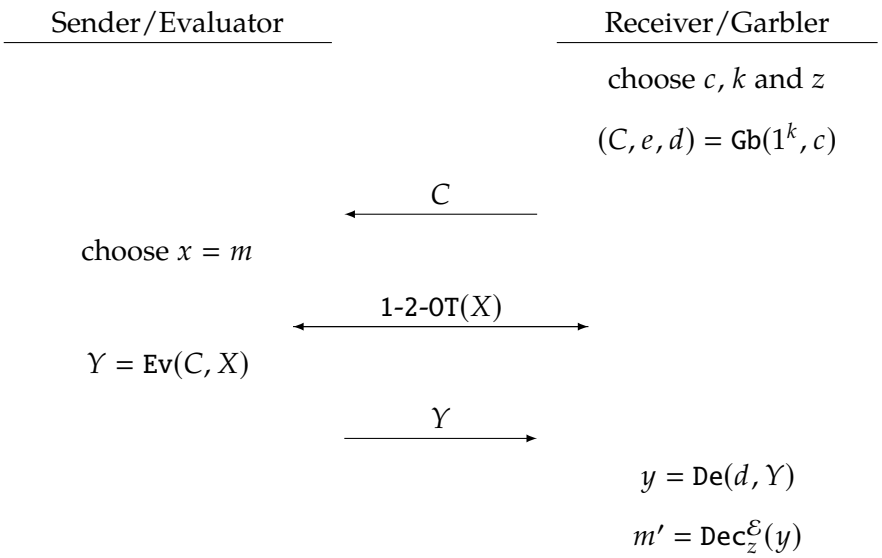

Figure 3.5: Modified white-box encryption scheme (MWBES2). Unlike in MWBES1, in this modified white-box encryption scheme, the circuit c computes the encryption of an encryption scheme $\mathcal{E}$. So in this case $y \neq m$ because y is the ciphertext to m. As an alternative to the protocol presented here, the garbler could send d together with C. In this case, the evaluator would decode Y and send y.

to proceed. Either, if the receiver sent d together with C, to decode Y and send y. Or to simply send Y without decoding it. Note that in this variant $y \neq m$ because $y = \mathtt{Enc}_z^{\mathcal{E}}(m)$. Depending on if the sender sent Y or y, the receiver either decodes and decrypts or just decrypts the message obtaining m. MWBES2 is a possibility to transform a symmetric encryption scheme into a public key encryption scheme (PKES). The garbled circuit is the public key and the key z is the private key. Like in other PKES, no third party is able to read the message or deduce the private key, even if all messages are intercepted. However, like in the standard SWBES the circuit may only be used once.

Both versions of MWBES have another big advantage over SWBES: Since the evaluator knows all of the inputs, privacy-free garbling can be used. Summarized, there are four main differences between MWBES and SWBES:

1. Sender and receiver have swapped roles.
2. Messages are sent in both directions.
3. 1-2-OT is needed, increasing the number of messages
4. Privacy-free garbling can be used.

3.3.1 Security

Assume that in MWBES2, the receiver is the one executing the decoding. Hence in both versions, the only information the sender gets from the receiver is C and X from which he computes Y. Naturally, the sender knows k and partially knows c since he needs to know which of the gates are even gates to perform the evaluation. He does not get any additional information about e, d or z if the garbling scheme used fulfills privacy. If in MWBES2, the receiver sends d and the sender decodes the output labels, this leads to the sender learning d but still learning nothing about e and z.

If the garbling scheme used fulfills privacy, the key z cannot be deduced from the messages sent. Therefore, a third-party attacker has no possibility to decrypt the message m, even if all messages are intercepted and y is sent instead of Y. This is a major advantage of MWBES over SWBES.

3.3.2 Computational and Communication Complexity

In this section, we compare the complexity of MWBES to the complexity of SWBES. On the one hand, computational and communication complexity increase because of the 1-2-OT that is not needed in SWBES. How much they increase depends on which 1-2-OT protocol is used. On the other hand, privacy-free garbling can be used, which decreases computational and communication complexity. Only one ciphertext per uneven gate is needed, causing the garbled circuit C to be only half as big as the garbled circuit in the SWBES. The number of XORs per uneven gate is reduced from up to 6 to 3 for the garbling and from up to 3 to up to 1 for the evaluation. The number of H-calls per uneven gate is reduced from 4 to 2 for the garbling and from 2 to 1 for the evaluation. See also Table 3.2. Applied to the analysis from Section 3.2.2, using AES-128 the total number of H-calls for the garbling and the evaluation is reduced by 22572 and 11286

Table 3.2: Comparison of number of H-calls and XORs per uneven gate between SWBES and MWBES, separated into garbling and evaluation.

	H-calls		XORs	
	SWBES	MWBES	SWBES	MWBES
garbling	4	2	$\{5, 6\}$	3
evaluation	2	1	$\{1, 2, 3\}$	$\{0, 1\}$

respectively. At the same time, the total number of XORs for the garbling is reduced by 22572 to 33858 and for the evaluation by up to 33858.

Summarizing, one can say that the communication complexity of MWBES is clearly lower than the communication complexity of SWBES since halving the size of the garbled circuit saves more space than the 1-2-OT adds. Concerning the computational complexity, it depends on the 1-2-OT protocol used whether MWBES is faster or slower than SWBES.

3.4 Two-Layer Scheme

A scheme with two layers of encryption can have multiple advantages for white-box cryptography. An example can be found on a flyer of the company *inside secure* [Sec]. The basic idea of this scheme is to have two layers of encryption: The upper layer is decrypting the key that is used in the lower layer for the actual encryption of data. This scheme is visualized in Figure 3.6. While the key from the upper layer is fix and embedded in the white-box (*static key*), the key from the lower layer is not fix and can be changed (*dynamic key*). This has multiple advantages over a simple white-box containing only one static key. First, the key used for the encryption or decryption of the data can be changed without altering the white-box. In a normal white-box, the key cannot be changed, so to use a new key a new white-box has to be created around it. This leads to it being less grave when a dynamic key gets extracted by an attacker and extends the lifespan of the white-box. Another advantage is that the dynamic key never leaves the white-box in its unencrypted form.

An important question of this scheme is how to obfuscate the white-box in such a way that the static key cannot be extracted. One possibility is to use a garbled circuit like we do in our single-use white-box in Section 3.1. The secret key and the encrypted dynamic key are the inputs to the garbled

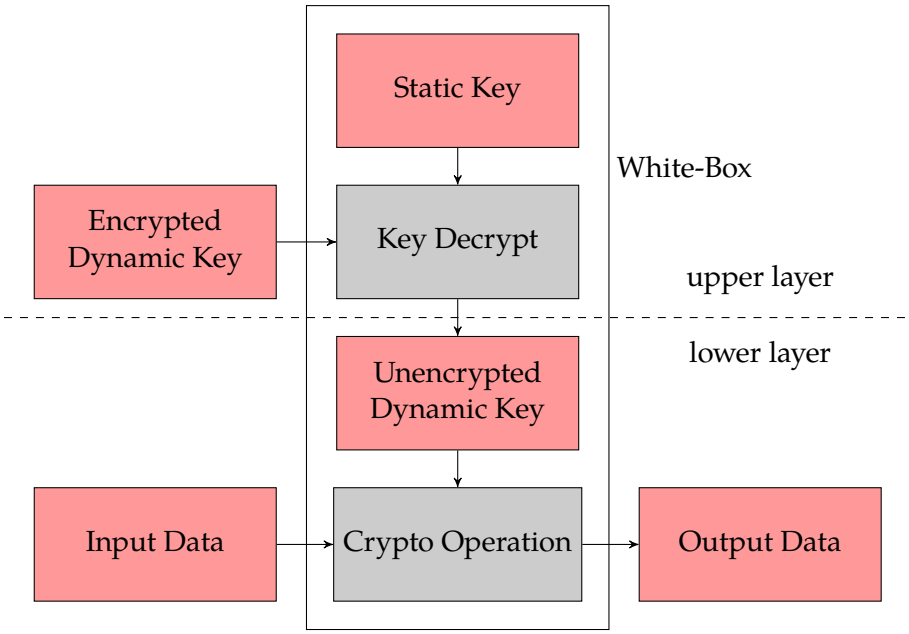

Figure 3.6: In the two-layer scheme, the upper layer decrypts an encrypted dynamic key using a static key. In the lower layer, this unencrypted dynamic key is used to perform a crypto operation. Reproduction of the chart from a flyer by the company *inside secure* [Sec].

circuit, and the dynamic key is the output. Because all inputs to the garbling function `Gb` are fix, the circuit does only have to be garbled once. On top of that, only the label of the secret key has to be stored instead of the secret key itself. When using another garbled circuit in the lower layer, there even is the possibility to use the same labels. Like this, there never is an unencrypted dynamic key.

This all sounds like a good idea, but the problem is that garbled circuits may only be used once. If a garbled circuit is evaluated multiple times with different evaluator's inputs, the evaluator is able to get information about the garbler's input. So in this case, the evaluator can get information about the static key by evaluating the garbled circuit in the upper layer multiple times with different unencrypted dynamic keys. To prevent this, the white-box has to be restricted to one specific input. The only way to achieve this is to know the encrypted dynamic key beforehand and store its labels together with those of the static key. This contradicts with the idea of the dynamic key. If the unencrypted dynamic key (that is not dynamic any more) is stored with the static key, one might just as well store the unencrypted (not) dynamic key directly, reducing our two-layer scheme to a standard white-box scheme. So unfortunately, there is no use for a garbled circuit in the two-layer scheme.

4 Zero-Knowledge Proofs

The SWBES protocol from Section 3.2 does not allow the receiver to get any information about the ciphertext x other than what can be inferred from the plaintext y. This property is not necessary for the SWBES protocol, but comes in handy when transforming it into a protocol for zero-knowledge proofs. In this section, we will introduce our garbled circuit zero-knowledge proof protocol that is based on SWBES and follows a new approach to zero-knowledge proofs.

4.1 Existing Approaches

Before we begin presenting our protocol, we first take a look at existing zero-knowledge proof protocols. As far as we know, the protocol by Jawurek, Kerschbaum and Orlandi [JKO13] is the only existing protocol for zero-knowledge proofs that uses garbled circuits. The ZKBoo protocol by Giacomelli, Madsen and Orlandi [GMO16] is another approach to zero-knowledge proofs that has some advantages over the one by Jawurek, Kerschbaum and Orlandi. In 2017, Chase et al. proposed ZKB++ which is an improved version of ZKBoo [CDG+17].

4.1.1 JKO13

In 2013, Jawurek, Kerschbaum and Orlandi proposed a protocol for zero-knowledge proofs using garbled circuits [JKO13]. Using this protocol, the prover is able to prove owning some input $\hat{x}$ that corresponds to a given output $\hat{y}$ regarding a specific one-way function f without revealing $\hat{x}$, just like in the example from section 2.4. To execute the protocol, a circuit describing f is needed so that x is the input and y is the output. Note that x and y are the actual in- and outputs that may differ from $\hat{x}$ and $\hat{y}$. The idea is that the verifier garbles the circuit and the prover evaluates it using the labels to his input x to obtain the labels to the output y. Giving

these output labels to the verifier, he proves owning x: Since he has no information about the mapping of the output labels, the only way to know the output labels that correspond to y is to know x and correctly evaluate the circuit. At the same time, the verifier cannot gain information about x because he cannot get any information about which input labels were used.

Now let's take a closer look at the course of events visualized in Figure 4.1: In the first step, the verifier garbles the circuit computing $(C, e, d) = \mathtt{Gb}(1^k, c)$. Then $\mathtt{COT}$ is used to transfer the input labels to the prover. The prover chooses the labels X corresponding to his secret input x. Now the prover is able to evaluate the circuit by computing $Y = \mathtt{Ev}(C, X)$. Then the output labels Y are sent to the verifier, so he can check whether they correspond to $\hat{y}$. If everything is done according to the protocol and the output labels correspond to $\hat{y}$, the verifier accepts.

Please note that this explanation does not exactly represent what happens in their protocol. In the original protocol, the circuit does not only compute f but also compares the result to $\hat{y}$. This way, the verifier has to check if the output label corresponds to 1 instead of $\hat{y}$. But since there is no difference in security or performance between these approaches, we decided to explain it the more intuitive way. Additionally, there is a little advantage in using just f for the circuit, because the circuit for f can be reused and does not have to be adjusted to fit $\hat{y}$.

So far, the verifier is able to manipulate the circuit to gain information about the provers input. To prevent this, the verifier has to *open* the garbled circuit, so the prover can check whether the circuit was manipulated. If the circuit is indeed manipulated, he aborts the protocol with the result that the verifier does not get any information about x. Before the circuit is opened, the prover has to commit on the output label. This prevents the prover from using the additional knowledge obtained by the opening of the circuit to cheat pretending to own $\hat{x}$.

By observing an opened circuit, one gets information about the nature of the gates. Meaning that when an evaluator gets the opening of a garbled circuit C, he can deduce the original circuit c. The opening does not reveal the function f implemented by c. Therefore, if an evaluator wants to check whether a given circuit C indeed describes a given function f, he should already know the circuit c implementing this function. This way he can compare the opened C to c and accept if they are equal. For protocols using garbled circuits we conclude that one should only use circuits that

Table 4.1: Comparison of numbers of operations needed for two approaches to check an opened circuit. The half-gates improvement is used in both cases. Numbers in parentheses are the values without optimization.

	garbling approach	evaluation approach
# XORs per even gate	2	2
# H-calls per uneven gate	4	4 (8)
# XORs per uneven gate	8	10 (14)

are either provided by a trusted third party or agreed upon by both parties in advance.

There are multiple possibilities to open a circuit. The first one that comes into mind is to send the mapping from labels to values for all labels used in the circuit. This way, the prover can decrypt all ciphertexts and check whether the gates are garbled correctly by comparing the type of the gates with those in c. A much more efficient way is to only send e. After evaluating the first layer of circuits using all the input labels from e, the evaluator knows all input labels for the next layer. So evaluating layer by layer, no additional information is needed.

An alternative way to check an opened circuit is to re-garble the circuit. The encoding function e together with c is the only information needed to garble the circuit, when the half-gates technique is used. Using other garbling schemes, there might be more randomness involved, so the garbler has to send his seed together with e to open the circuit. In this variant, the evaluator garbles the circuit anew and compares his result to C, accepting in case of a match.

The communication complexity of both approaches can be improved by only sending one label of each label pair and additionally sending the offset R. Another possibility is to only send the seed that was used to create R and the input labels. This improvements come with the trade-off of a slightly higher computational complexity since the evaluator has to generate e out of the labels and R or out of the seed.

To compare the computational complexity of the garbling approach and the evaluation approach, we assume that the half-gates technique is used. In Table 4.1 the comparison of the numbers of needed operations is summarized. In both approaches, there are two XORs per even gate. For the

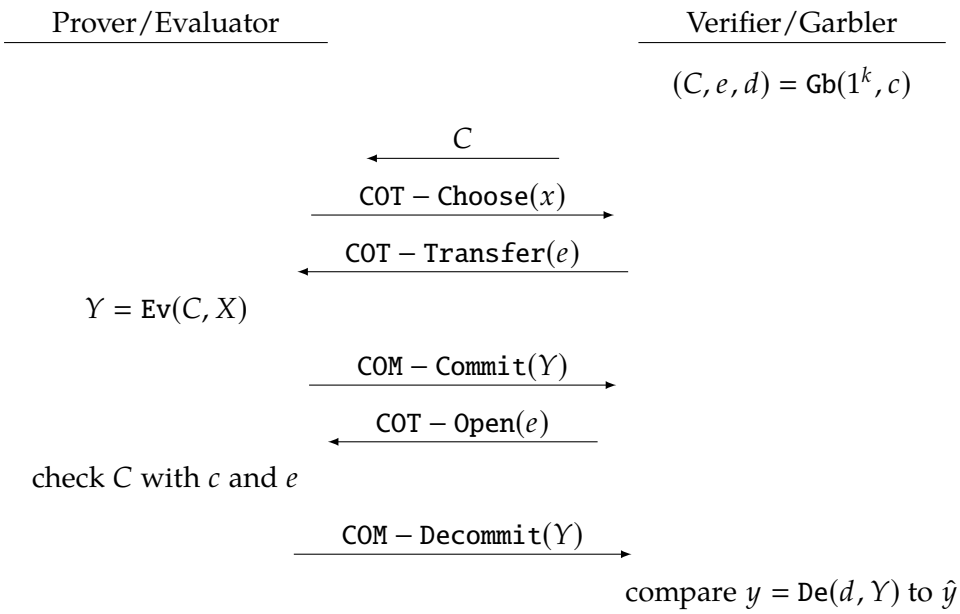

Figure 4.1: The JKO13 protocol by Jawurek, Kerschbaum and Orlandi is a zero-knowledge proof protocol using garbled circuits enabling the prover to prove knowing an input $\hat{x}$ without revealing it to the verifier [JKO13].

uneven gates, there is a little difference: Using the garbling method, there are 8 XORs and 4 H-calls per uneven gate. Using the evaluation method there would be 14 XORs and 8 H-calls, but these numbers can be slightly improved. When decrypting all four ciphertexts one after another, there are some redundancies in the computation. By eliminating these redundancies, the computational complexity can be reduced to 10 XORs and 4 H-calls per uneven gate. At the end, there is only a difference of 2 XORs per uneven gate, letting the evaluation method be slightly more expensive. Another difference is that for the garbling method, all generated ciphertexts have to be compared to those in C, while for the evaluation method, only the resulting decoding function has to be compared to the given d. This advantage of the evaluation method compensates its disadvantage in the number of XORs. All in all both variants have a similar computational complexity and can both be used to check the opening of a circuit.

Security

In the following, we analyze the security of the JKO13 protocol by checking whether the security notions for zero-knowledge proofs from Section 2.4.1 are fulfilled.

To fulfill completeness, the probability that an honest prover knowing $\hat{x}$ fails proving this knowledge although the protocol was executed correctly has to be negligible. This security notion is fulfilled if a correct garbling scheme is used. The correctness of the garbling scheme guarantees that if the prover uses $\hat{x}$ as input, the output y equals $\hat{y}$. Since the verifier accepts in case of a match between y and $\hat{y}$, the probability that an honest prover knowing $\hat{x}$ fails is 0.

The security notion of soundness requires the probability of a prover not knowing $\hat{x}$ to convince the verifier to be negligible. This security notion can be fulfilled by choosing a garbling scheme that fulfills authenticity: The authenticity property guarantees that an evaluator can only get the set of output labels, that corresponds to the given input labels. Hence it is not possible to obtain output labels Y that correspond to $\hat{y}$ without using input labels X that correspond to $\hat{x}$ as input. The only way to obtain such input labels in the `1-2-OT` is to know $\hat{x}$.

The last security notion to fulfill is the zero-knowledge property. It requires the verifier not to be able to learn any information about the prover's input x that he cannot deduce from the statement to be proven. There are two possibilities for the verifier to get such information: First, the `1-2-OT` of the input labels comes into mind since the labels transferred have to depend on x. To prevent this, the `COT` protocol used has to be zero-knowledge itself. The second possibility would be to manipulate the circuit in such a way that the output gives more information on x than y. This is prevented by the opening of the circuit: If the prover comes to the conclusion that the circuit is manipulated, he aborts the protocol before sending Y.

The zero-knowledge property can be proven formally by showing the existence of a simulator S (see Section 2.4): To generate an output that is indistinguishable from an output generated by an honest prover knowing $\hat{x}$, the simulator has to generate the messages `COT-Choose`, `COM-Commit` and `COM-Decommit`. The `COT-Choose` message is the easiest of the three: Since the `COT` protocol guarantees that the verifier does not get any information about the prover's input, the simulator can simply use a random input x'

to generate this message. It will in any case be indistinguishable from an honest prover's output. For the messages `COM-Commit` and `COM-Decommit` to also be indistinguishable from an honest prover's output, they both have to use the same output labels Y' that correspond to $\hat{y}$. Since there is only one set of output labels Y corresponding to $\hat{y}$, the simulator has to use $Y' = Y$. In order to do so, S generates the opened circuit from the `COT-Open` message that was sampled by the oracle. This way the simulator gets d which allows him to compute the output labels Y. The knowledge of Y enables S to generate the correct messages `COM-Commit` and `COM-Decommit`. These messages are not distinguishable from an honest prover's output, because they are the same messages a prover knowing $\hat{x}$ would generate. When acting as explained above, a simulator S is able to generate a prover's output that is indistinguishable to a real prover's output without knowing $\hat{x}$.

Cost Analysis

The JKO13 protocol is compatible to the half-gates improvement as well as to its variant including privacy-free garbling. Therefore, only one ciphertext and 3 H-calls per uneven gate are needed (see table 2.1). All in all, the protocol is similar to the `MWBES2`. The verifier garbles the circuit and sends C over to the evaluator. There is no encryption like in `MWBES2`, but the input is also transferred via `1-2-OT`. The second difference is that the oblivious transfer needs to be committing to enable the garbler to open the circuit later on in an additional step. Another additional step is the commitment to the output labels Y that is done before the opening of the circuit. The last difference is that the evaluator needs to check whether the circuit was manipulated and the receiver checks whether $y = \hat{y}$.

The base of the computational complexity is the garbling and evaluation. Due to the similarity to our protocols from Section 3, we are not going to repeat all of the analysis. How to compute the amount of operations needed without using privacy-free garbling is presented in Section 3.2.2. The improvements through privacy-free garbling are pointed out in Section 3.3.2. Another big part of the computational complexity is the checking of the opened circuit since it results in a full evaluation or a re-garbling of the circuit. This takes up to 4 H-calls and 10 XOR operations per uneven gate and 2 XOR operations per even gate (see Table 4.1). Aside from that, there are the committing oblivious transfer of the input labels and the commitment

to the output labels that carry weight. The computational complexity of these depend on which protocols are chosen. The comparison of y and $\hat{y}$ is the least important part of the computational complexity.

Concerning the communication complexity, the garbled circuit C is the largest information to be sent. Thanks to the usage of privacy-free garbling it is only half the size compared to using standard half-gates. Only one ciphertext of k bit per uneven gate is needed, to represent the garbled circuit. Using AES-128 like in our analysis in section 3.2.2, its size would be $11286 \cdot k$ bit. The rest of the communication complexity depends on which `COT` and `COM` protocols are used.

4.1.2 ZKBoo and ZKB++

The ZKBoo-Protocol was proposed by Giacomelli, Madsen and Orlandi in [GMO16], and is based on the *MPC-in-the-head* approach by Ishai et al. [IKOS07]. Similar to JKO13, the prover proves the knowledge of an input $\hat{x}$ corresponding to an output $\hat{y}$ concerning a circuit c without revealing $\hat{x}$. A difference is that the prover does not garble the circuit, but uses a multi-party-computation protocol (MPCP) to evaluate it. It is assumed, that the circuit c is fix (Giacomelli et al. use SHA-1 or SHA-256) and there already exists a decomposition into a three-party MPCP with certain properties. This is a big advantage since the prover can directly begin by executing the MPCP. For a visualization of the protocol, see Figure 4.2. There is no other participant of the MPCP because all three parties p_i with $i \in \{1, 2, 3\}$ are "in the prover's head", meaning that he simulates them. The input x to the circuit is divided into three randomly chosen parts x_i with $i \in \{1, 2, 3\}$ such that $x = x_1 \oplus x_2 \oplus x_3$. Each party p_i gets a part x_i of the input and its own random tape r_i. All messages received by a party, as well as its input x_i and the random tape r_i are documented in the party's view v_i. After the execution of the MPCP, each party has knowledge of a part y_i of the computation's output. If the protocol was executed correctly, all parties being honest, y can be reconstructed from the y_i.

After the simulation step, the prover commits himself to the views v_1, v_2 and v_3 and sends the output shares y_1, y_2 and y_3 to the verifier. The verifier picks and sends two indices i and j which he wants to receive the corresponding views to. So the prover reveals the views v_i and v_j that were requested by the verifier. To check whether the MPCP was correctly executed, the verifier checks whether the views are compatible to each

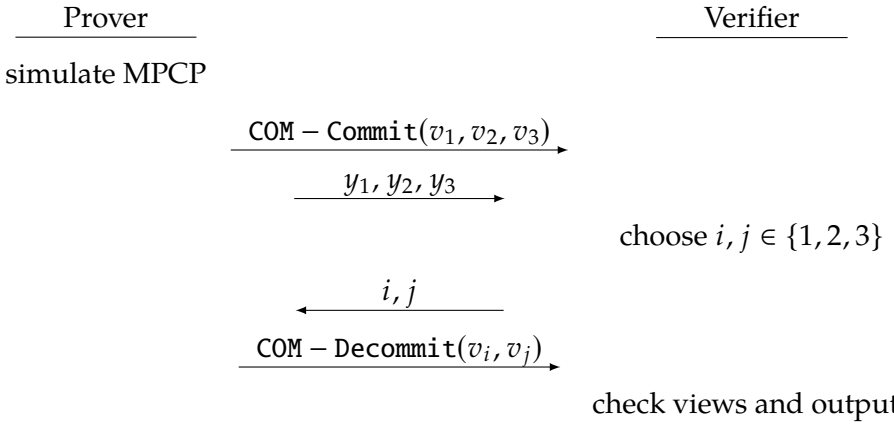

Figure 4.2: The ZKBoo protocol by Giacomelli, Madsen and Orlandi is a zero-knowledge proof protocol enabling the prover to prove knowing an input $\hat{x}$ without revealing it to the verifier [GMO16].

other and the parties acted according to the MPCP. If this is given and the output combination of the y_i equals $\hat{y}$, the verifier accepts.

In 2017, Chase et al. introduced the ZKB++ protocol [CDG+17] which is an improvement of the ZKBoo protocol. The communication complexity of ZKB++ is more than halved compared to ZKBoo, not affecting the computational complexity. This is done by using six different optimizations that are designed to compress all messages sent as much as possible.

Security

To analyze the security of the protocol, we check for completeness, soundness and the zero-knowledge property.

Completeness is given if the MPCP used is correct and used correctly with $\hat{x}$ as input. In this case, all views are compatible and the output reconstructed from the y_i is $\hat{y}$ causing the verifier to accept with probability 1.

Next, we consider the security notion of soundness. There indeed is the possibility for a prover to successfully manipulate the MPCP without being caught. Since the verifier only gets two of the views, he cannot check whether one of these is incompatible with the third view. So if the prover manipulated the MPCP resulting in two parties' views being incompatible

with each other, the verifier can only convict the prover when choosing the correct two views. The probability for this is $1/3$, leaving a probability of $2/3$ that the prover successfully convinces the verifier to accept without knowing $\hat{x}$. For this reason, the protocol is executed multiple times, reducing the probability that a cheating prover convinces the verifier in every round. The probability that a prover not knowing $\hat{x}$ convinces the verifier in t rounds is $(2/3)^t$. By changing t, the protocol can be scaled to different needs.

The zero-knowledge property requests the verifier to have no possibility to get information about the prover's input $\hat{x}$ that cannot be deduced from the output y. To fulfill this, the MPCP used by Giacomelli et al. guarantees that none of the three parties gets any information about the other parties' inputs that cannot be deduced from the output. So for example p_1 gets no information about x_2 and x_3. This way, the verifier always lacks one of the x_i because he only gets two of the views. Since the x_i are chosen at random and all three have to be xored to obtain x, knowing two of them gives no information about the input x. Therefore the ZKBoo protocol fulfills the zero-knowledge property.

Comparison to JKO13

An advantage over JKO13 is that the ZKBoo protocol is a lot faster because no oblivious transfer is needed. Unfortunately, there are no exact values, because there are no comparable analyses of both protocols. But like stated by Giacomelli et al., the oblivious transfers from JKO13 take already more time than 137 executions of ZKBoo, ignoring the other parts of the protocol. A single execution of ZKBoo also has a lower communication complexity than JKO13. While the ZKBoo protocol has a proof size of 3320 Byte, JKO13 has a proof size of 186880 Byte. But since it is possible for the prover to successfully manipulate the MPCP, the protocol is repeated multiple times to lower the probability for a successful manipulation. Repeating the ZKBoo protocol more than 56 times, the proof size gets bigger than the one of JKO13.

Unlike JKO13, ZKBoo can be made non-interactive. There are multiple possibilities to transform a sigma protocol like ZKBoo into a non-interactive zero-knowledge protocol (NIZK protocol) [FS86, Unr15]. For an explanation of sigma protocols see Section 4.3.2. This can be useful when being in a network with much latency.

But all these advantages of ZKBoo over JKO13, do not make it a bad idea to use garbled circuits for zero-knowledge proofs in general: In the next section, we present a zero-knowledge protocol using garbles circuits that does not need any oblivious transfer causing a huge advantage in computational and communication complexity. Additionally, `ZKGC-nOT` is a sigma protocol like ZKBoo and thus may be transformed into a NIZK protocol.

4.2 ZKGC-nOT

In this section, we present a new and more efficient zero-knowledge proof protocol using garbled circuits. Like in the protocols presented above, in `ZKGC-nOT` the prover is able to prove ownership of some input $\hat{x}$ that corresponds to some output $\hat{y}$ concerning a one-way function f. The protocol is based on the `SWBES` presented in Section 3.2. The `SWBES` has the property, that the evaluator is not able to get any information about the garbler's ciphertext. In this zero-knowledge protocol the garbler's input is not a ciphertext, but the secret information $\hat{x}$.

Let's take a look at the course of action illustrated in Figure 4.3. In the first step, the prover garbles the circuit using the garbling function `Gb`. Like explained in Section 4.1.1, the circuit c should be agreed upon in advance or be provided by a trusted third party, to facilitate the validity check of the opened circuit. In the next step, the prover sends the garbled circuit C and the decryption function d to the verifier. Now, the verifier chooses a decision bit b that is either "$labels$" or "$open$" and sends it to the prover. Let's assume the verifier chooses "$labels$". This causes the prover to compute the input labels X corresponding to his input x and sending them to the verifier. With this knowledge, the evaluator is able to evaluate the circuit and decode the output labels. In the last step, the verifier compares the obtained output y to the $\hat{y}$ from the statement to be proven. In case of a match, the verifier accepts.

If the verifier chooses "$open$", the prover sends e instead of X. Sending the encryption function *opens* the circuit, enabling the verifier to check whether C is a garbled version of c. This opening is needed to prevent the prover from manipulating the circuit in such a way, that it always outputs $\hat{y}$ no matter the input. Like in the JKO13 it is possible to only send one half of the labels together with R, or to just send the seed used to generate

the input labels instead of e to reduce the communication complexity. The verifier accepts if no manipulation is detected.

The JKO13 protocol by Jawurek et al. also is a zero-knowledge proof protocol using garbled circuits, but there are several differences: Unlike in JKO13, in `ZKGC-n0T` the prover is the one who garbles the circuit. Because the prover knows the encoding function e, he is able to directly send the input labels X to the verifier without any need of oblivious transfer. This saves a lot of space and time resources. Another difference is that the evaluator does either receive the input labels X to evaluate the circuit, or the encoding function e to check whether the circuit was garbled correctly. In JKO13 both steps are needed. The origin of this difference lies in the fact that in `ZKGC-n0T`, the circuit is not opened to prevent the verifier from extracting $\hat{x}$, but to achieve soundness. There also is no commitment needed in `ZKGC-n0T`.

4.2.1 Security

To analyze the security of the protocol, we are going to check whether the security notions presented in section 2.4.1 are fulfilled.

Completeness

To fulfill completeness, the garbling scheme used has to be correct. If the prover knows $\hat{x}$, he is able to use the encoding function e to determine the corresponding labels $\hat{X}$ after garbling the circuit. Evaluating the circuit with $\hat{X}$ as input, the verifier obtains $\hat{Y}$. In the last step, the verifier uses the decoding function d mapping $\hat{Y}$ to $\hat{y}$. Since the output matches $\hat{y}$, the verifier accepts. Hence the probability of a prover knowing $\hat{x}$ to successfully prove knowing $\hat{x}$ when sticking to the protocol is 1, so completeness is fulfilled.

Soundness

Assume the garbling scheme is correct and the circuit is garbled correctly. In this case, a prover that does not know $\hat{x}$ has no chance to construct an input label $\hat{X}$ that produces the correct output label $\hat{Y}$ besides guessing, since he would have to invert the one-way function to do so. The only

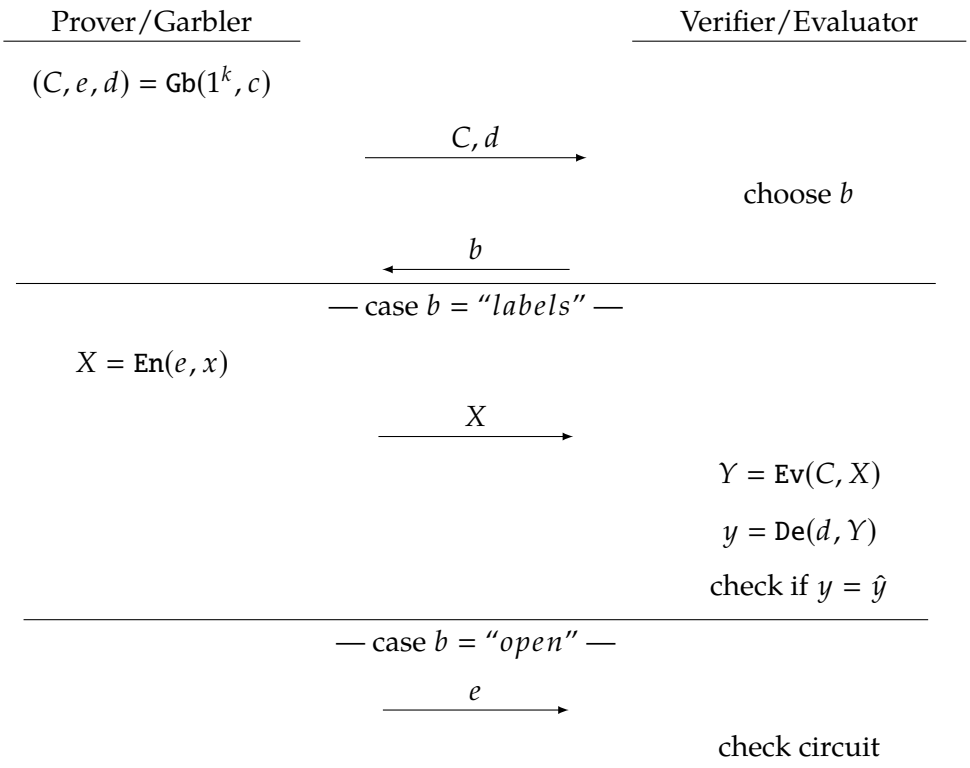

Figure 4.3: ZKGC-n0T. With help of the decision bit b, the verifier chooses whether to get the input labels to verify the proof, or to let the prover open the circuit so the verifier can check whether the circuit was manipulated.

possibility for him to construct an input label that results in an output label $\hat{Y}$ when evaluating the circuit is to manipulate the circuit. So if the verifier could check both whether the circuit is garbled correctly, and whether the input labels produce a valid output, the prover would have no chance to convince the verifier without knowing $\hat{x}$. Since this is not the case, there is a strategy for a malicious prover: He guesses the choice of the verifier and then either manipulates the circuit ($b = "labels"$) or correctly garbles the circuit ($b = "open"$). With this strategy, there is a fifty-percent chance for the attacker to get away with his cheating.

As long as the prover is honest-but-curious or a covert attacker, he would not dare trying to cheat. If the prover is a malicious attacker, cheating-attempts could be uncovered with increasing probability by repeating the protocol multiple times. The probability that a prover not knowing $\hat{x}$ convinces the verifier in t rounds is $(1/2)^t$.

Zero-Knowledge

In order to prove the zero-knowledge property of `ZKGC-n0T`, we show that the verifier cannot learn anything about the input x that cannot be learned from y. For this, we create a simulator S that has no information about $\hat{x}$ but is able to produce an output which is indistinguishable from the output of a prover knowing $\hat{x}$. While this is not achievable for a prover not knowing $\hat{x}$, it is for the simulator because he has the advantage of being allowed to rewind the verifier. This means that S guesses the choice of the verifier and acts depending on that guess, and later on is able to rewind the verifier to try again if the guess was incorrect.

In the case $b = "open"$, the simulator would garble the circuit correctly. This way, the messages containing C, d and e cannot be distinguished from a prover's output, because they have the same distribution (perfectly indistinguishable).

In the case $b = "labels"$, the simulator would also begin with garbling the circuit correctly. But after choosing random input labels X and feeding them into the garbled circuit, the simulator manipulates d in a way that the output labels map to the expected output $\hat{y}$. Therefore, d is the only manipulated parameter and the input labels X and the circuit C are constructed just like a prover would construct them. While manipulating d, the pairs of labels belonging to each bit are not altered. Only their mapping

to zero and one may be swapped. This way, d stays compatible to the garbled circuit C and nobody can distinguish it from a correctly generated d (computationally indistinguishable). Hence, the simulator's output cannot be distinguished from a provers output and the zero-knowledge property is fulfilled.

Another way to give reasons for the zero-knowledge property is to evaluate what the verifier gets to know. He has the choice to get either (C, d, e), or (C, d, X). In the case $b = \text{“}open\text{”}$, he obtains (C, d, e) which contains no information about x since x does not affect the garbling of the circuit. In the case $b = \text{“}labels\text{”}$, he obtains (C, d, X) which contains information about x because X contains the labels for x. The security notion of obliviousness discussed in Section 2.3.1 ensures that knowing a garbled circuit and some input labels, one cannot deduce anything about the inputs or the outputs. Zahur et al. proved in [ZRE15] that the half-gates improvement fulfills obliviousness. Hence, if `ZKGC-n0T` is executed using half-gates, the verifier does not get any information about the provers input.

Note that the only influence the verifier has on the course of the protocol is the choice of b. Hence, we do not need to distinguish between honest, covert and malicious verifier, since there are only two choices for b and there is no strategy to pick it in a special way.

4.2.2 Applicable Garbled Circuit Improvements

`ZKGC-n0T` is compatible with the two-halves improvements suggested by Zahur, Rosulek and Evans in [ZRE15]. However, unlike JKO13, `ZKGC-n0T` is not compatible with the privacy-free improvement by Frederiksen, Nielsen and Orlandi [FNO15]. This is because the evaluator has to know all of the inputs for the improvement to be compatible. In `ZKGC-n0T`, this is not possible because if the evaluator knew the secret inputs x, it would no longer be a zero-knowledge proof.

4.2.3 Computational and Communication Complexity

In the case that the verifier chooses $b = \text{“}labels\text{”}$, the protocol's computational complexity is the same as for `SWBES`. For a detailed analysis, see Section 3.2.2. If the verifier chooses $b = \text{“}open\text{”}$, the evaluation of the circuit and the decoding of the labels are replaced by the validity check

of the circuit. How this check can be done and what its complexity is, is presented in Section 4.1.1. Both possible alternatives have a similar computational complexity, which is higher than the complexity of the simple evaluation since the whole circuit has to be checked, and not just one possible evaluation.

Note that the garbling of the circuit and the encoding of the input can be precomputed by the prover. So if for example, the zero-knowledge proof is used for authentication, the prover can precompute multiple garbled circuits in order to speed up a future authentication process.

The communication complexity of ZKGC-n0T is very similar to the one of SWBES. In the case $b = \text{“}open\text{”}$, the communication complexity of the last message is doubled because e contains all of the input labels instead of only one set like X does. Additionally, in both cases the communication complexity is increased by one bit to send the decision bit b.

4.2.4 Comparison to other Protocols

Compared to JKO13, ZKGC-n0T has some important advantages: It does not need oblivious transfer or commitment, which has positive effects on both computational and communication complexity. Also the security primitives for commitment and oblivious transfer are not needed. The swapping of the roles of garbler and evaluator might be another advantage. Since the prover is the garbler, he begins by garbling the circuit – in JKO13, it is the other way round. If the verifier is a server getting requests by users who want to authenticate themselves, it is much easier to perform a denial of service attack if the server has to begin the protocol by garbling a circuit. Unlike in JKO13, privacy free garbling cannot be used in ZKGC-n0T since the evaluator is not allowed to know the garbler's input. But since we do not need oblivious transfer and commitment, the overall computational complexity of ZKGC-n0T should still be lower. Another difference is, that in ZKGC-n0T there is a possibility for a prover not knowing $\hat{x}$ to convince the verifier. Therefore it has to be executed multiple times to minimize this risk. This problem does only occur if the prover might be malicious. In a scenario with a covert attacker model, the prover would not dare to cheat because he would be convicted with a probability of one half. An additional advantage of ZKGC-n0T is that it is very versatile: Unlike JKO13 it can be transformed into a non-interactive zero-knowledge proof protocol,

and adding oblivious transfer is an optional trade-off that provides other advantages (see Section 4.3).

To compare ZKGC-n0T to ZKBoo and ZKB++ is rather difficult, since their implementations are build around certain circuits for SHA-1 and SHA-256. About the number of repetitions needed can be said that ZKGC-n0T has to be repeated fewer times for the same security level, because the probability for a prover not knowing $\hat{x}$ to convince the verifier is smaller. In numbers, for a probability of 2^{-40} there are 69 executions of ZKBoo needed (because $(2/3)^{69} \approx 2^{-40}$), while only 40 executions of ZKGC-n0T are needed ($(1/2)^{40} = 2^{-40}$). Another advantage of ZKGC-n0T is that it is compatible to arbitrary circuits. On the downside, the proof size of ZKGC-n0T is larger because the whole circuit needs to be sent and the bits are replaced with labels instead of being sent directly.

ZKGC-n0T might not be best one for use in practice known today, but it still is a whole new approach to zero-knowledge using garbled circuits. Maybe with some further research, this will open doors to new efficient ways to do zero-knowledge proofs using garbled circuits. To give some inspiration about how ZKGC-n0T may be varied, we show some variations in the next section.

4.3 Protocol Variations

The original version of ZKGC-n0T does not need a trusted third party, commitment or oblivious transfer. But as it is very flexible, it is possible to include some of these aspects to get advantages the standard version does not have. On the one hand, it is possible to prevent a malicious prover to notice that he was caught cheating by using oblivious transfer. On the other hand, there are multiple possibilities to transform ZKGC-n0T into a non-interactive zero-knowledge protocol. Both approaches are presented in this chapter.

4.3.1 Using Oblivious Transfer

It is possible to substitute the decision between obtaining the encoding function e or the garbled input X by an application of oblivious transfer. Using 1-2-0T, the verifier only obtains either e or X which guarantees the

privacy of x. To enable an 1-2-OT of e and X, X has to be concatenated to random bits to align it to the same length as e. Furthermore, in contrast to our protocol from above, the prover does not know whether the verifier requests to see X or e. An overview of this alternative protocol is given in Figure 4.4.

The benefit of using 1-2-OT in this context is that a dishonest prover does not know whether he is going to be caught cheating before sending X or e. An attacker could try to cheat and abort the protocol when detecting that he did not successfully predict the decision bit in order to not be caught cheating. The verifier would not detect the cheating attempt until e or X is sent. The 1-2-OT variant prevents the attacker from aborting and restarting because he does not get any information about the decision bit until it is too late.

However, this comes at the cost of oblivious transfer, which increases complexity and, depending on its implementation, may introduce new security assumptions.

4.3.2 Non-Interactive Zero-Knowledge Approach

Recently, a number of works [Unr15, BSCTV17, BSCTV14, GGI+15] focused on describing non-interactive zero-knowledge proofs (NIZK proofs). The non-interactive variant for zero-knowledge proofs was introduced by Blum et al. in 1988 [BFM88]. In exchange for the fact that prover and verifier need to have a common random string, it is possible to reduce the communication to just one message from prover to verifier. This technique allows zero-knowledge proofs to be used as signature schemes, which should not rely on interaction to be practical. For authentication however, interaction is fine as long as the total communication costs are low.

Nevertheless, it is easy to transform ZKGC-nOT into a non-interactive zero-knowledge proof, which is done in this section. At first, we show that ZKGC-nOT is a sigma protocol, which is needed for the most common transformations into NIZK-proofs by Fiat and Shamir [FS86] and Unruh [Unr15]. After that, we present our own approach to transform our protocol.

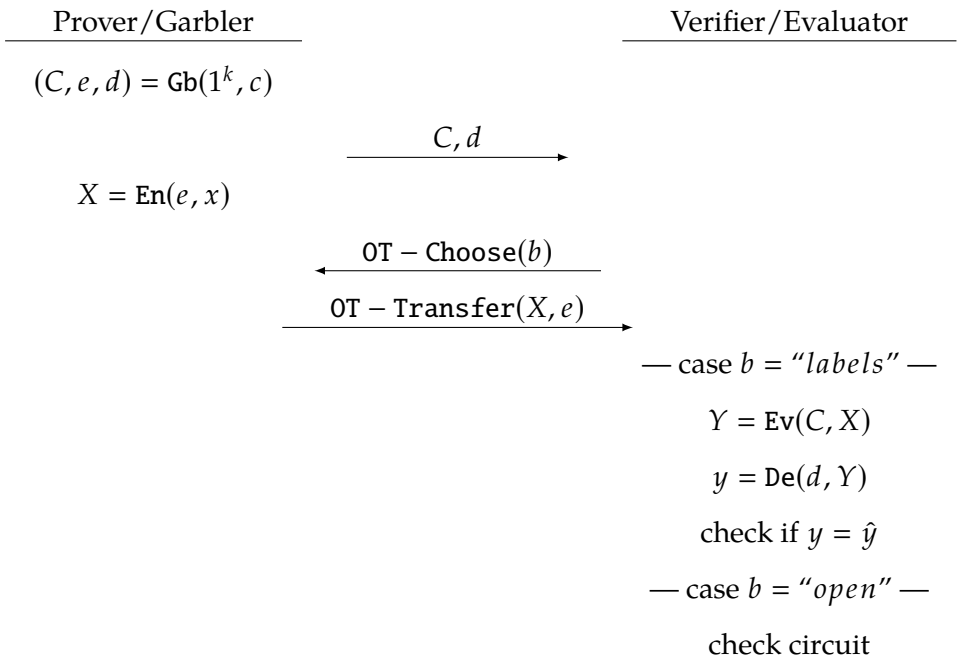

Figure 4.4: OT-variant of ZKGC-nOT. The prover does not get to know b since he sends both X and e via 1-2-OT. This way he does not know whether he is caught cheating. If he knew b and it did not match his expectation, he could abort the protocol and start it anew.

Sigma Protocols

A sigma protocol is a special form of a prover-verifier protocol with exactly three messages, namely *commitment*, *challenge* and *response*. According to a sigma protocol, the prover sends a commitment message `com` derived from a statement s and witness w. On arrival, the verifier randomly draws a challenge `ch` from a fixed set of challenges and sends it to the prover. In the next step, the prover takes `ch`, creates a response message `resp` and sends it to the verifier. Finally, the verifier takes s, `com`, `ch` and `resp`, accepts if the response is appropriate concerning the challenge, and rejects otherwise.

As in any zero-knowledge proof, a sigma protocol has the properties completeness and zero-knowledge as described in section 2.4.1. Further, the property of *special soundness* which is a stricter variant of soundness is needed. For special soundness, there must exist a polynomial-time algorithm which is able to generate a valid witness w out of a statement s and two accepted protocol runs (`com`, `ch`, `resp`), (`com`, `ch'`, `resp'`) for the same commitment message `com` but two different challenges `ch` $\neq$ `ch'`.

The original protocol depicted in Figure 4.3 is a *sigma protocol*. The commitment consisting of the garbled circuit C and the decoding function d is followed by the challenge b, which prompts the prover to respond with either opening the circuit C by sending the input mapping e or by sending the labels X for the input x. The witness w corresponds to the prover's secret input x and the statement s corresponds to the output y of the circuit.

Because there are only two possible challenges, special soundness is easy to prove. Let's assume that a polynomial-time algorithm is provided C, d, y and X as well as e. By knowing e and X, the algorithm can easily calculate the secret x by inverting e and applying it to X.

Non-interactive protocol

There are several ways to transform any generic sigma protocol into a NIZK proof. The most relevant ones are the Fiat-Shamir transformation [FS86] and the more recent construction by Dominique Unruh [Unr15], which achieves security against quantum adversaries. In the following, we explain our own approach transforming `ZKGC-n0T` into a NIZK proof.

The protocol begins with a trusted third party generating a black-box random oracle $O : \{(C, d)\} \to \{\text{“}labels\text{”}, \text{“}open\text{”}\} \times Hashes$. After O is transferred to the prover and the verifier, the prover generates (C, e, d) as usual and obtains b and a hash $h \in Hashes$ by calling $O(C, d)$. The oracle O includes an internal counter i, so the hash h depends on (C, d) and i which is the number of previous calls to O. The prover then sends (C, d, b, h, X) or (C, d, b, h, e) (depending on b) to the verifier. The verifier checks whether $(b, h) = O(C, d)$ and X (or e) meets his expectations.

If the prover is malicious, he could try to call the oracle multiple times, to get a challenge b matching his tuple (C, d) which was generated hoping for a certain choice of b. The random oracle's internal counter prevents this behavior, because if O is queried multiple times, the hash is not the same as the one the verifier obtains. Effectively, we limit the prover to a single call of O.

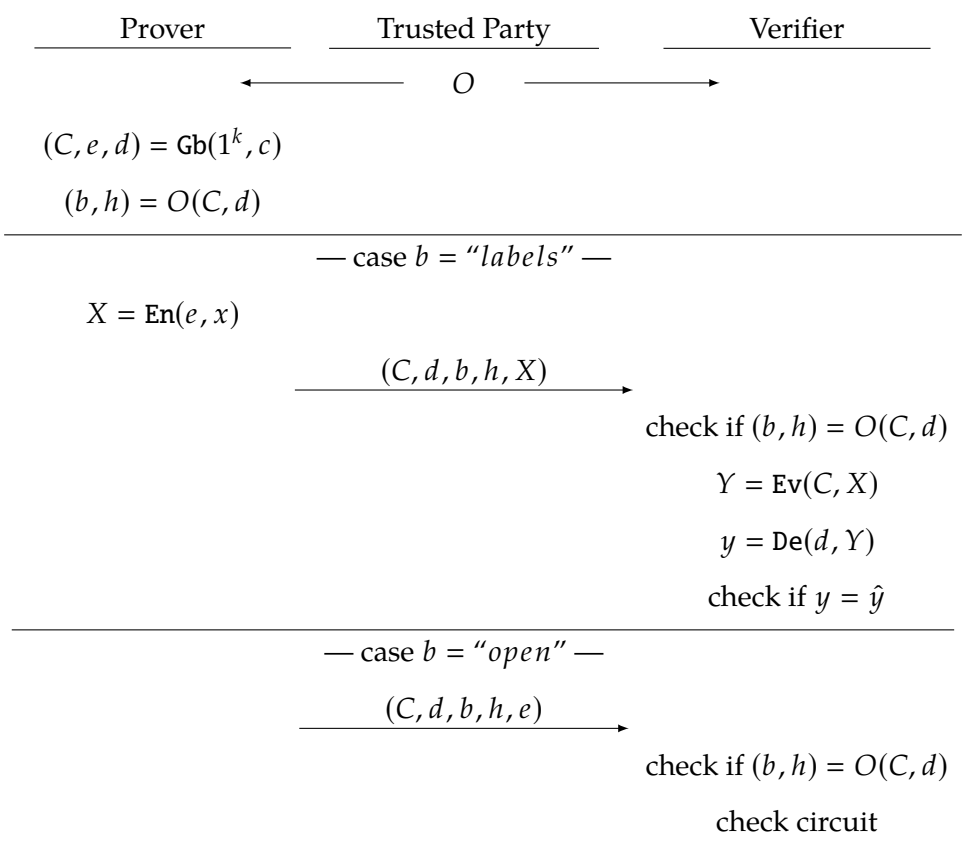

Figure 4.5: The non-interactive variant of our zero-knowledge protocol. To achieve non-interactiveness, a trusted third party providing a black-box random oracle O is needed.

5 Conclusion and Future Work

In this work, we analyzed different approaches on how to use garbled circuits in a white-box scenario. We began by describing how garbled circuits can be used as single-use white-boxes. Next, we introduced the simple white-box encryption scheme `SWBES` and the modified white-box encryption scheme `MWBES`, that can be used to transfer encrypted messages from a sender to a receiver. In comparison, `MWBES` is the better choice to send secret messages, because it is secure against an attacker eavesdropping on the channel. Additionally, `MWBES` is compatible to privacy-free garbling, causing the circuit to be only half as big as in `SWBES`. We also analyzed the idea of a two-layer scheme for white-box cryptography. To have two layers of encryption has several advantages for white-box cryptography, that extend the life span of a white-box. Sadly, it is not compatible to garbled circuits, since the restriction to single-use of garbled circuits contradicts with the requirement to use different dynamic keys. Finding a way to reuse garbled circuits would make the combination of these techniques possible and support the development towards long-life white-boxes.

We designed the `ZKGC-n0T` protocol, which is a zero-knowledge proof protocol using garbled circuits, based on `SWBES`. `ZKGC-n0T` is a whole new way of using garbled circuits in zero-knowledge proofs. In its main variant, it has multiple advantages over JKO13, and as there are other variants, it is very versatile and can be adapted to different use cases. In a scenario with a covert attacker, `ZKGC-n0T` is superior to JKO13 concerning communication and computational complexity.

Bibliography

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 103–112. ACM, 1988.

[BGEC04] Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a white box aes implementation. In *International Workshop on Selected Areas in Cryptography*, pages 227–240. Springer, 2004.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796. ACM, 2012.

[BL18] Fabrice Benhamouda and Huijia Lin. k-round multiparty computation from k-round oblivious transfer via garbled interactive circuits. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 500–532. Springer, 2018.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 503–513. ACM, 1990.

[BSCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *USENIX Security Symposium*, pages 781–796, 2014.

[BSCTV17] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. *Algorithmica*, 79(4):1102–1160, 2017.

[CDG+17] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1825–1842. ACM, 2017.

[CEJVO02] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C Van Oorschot. White-box cryptography and an aes implementation. In *International Workshop on Selected Areas in Cryptography*, pages 250–270. Springer, 2002.

[DLPR13] Cécile Delerablée, Tancrède Lepoint, Pascal Paillier, and Matthieu Rivain. White-box security notions for symmetric encryption schemes. In *International Conference on Selected Areas in Cryptography*, pages 247–264. Springer, 2013.

[EGL85] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(6):637–647, 1985.

[FNO15] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 191–219. Springer, 2015.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology—CRYPTO'86*, pages 186–194. Springer, 1986.

[GGI+15] Craig Gentry, Jens Groth, Yuval Ishai, Chris Peikert, Amit Sahai, and Adam Smith. Using fully homomorphic hybrid encryption to minimize non-interative zero-knowledge proofs. *Journal of Cryptology*, 28(4):820–843, 2015.

[GMO16] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. Zkboo: Faster zero-knowledge for boolean circuits. In *USENIX Security Symposium*, pages 1069–1083, 2016.

[GS18] Sanjam Garg and Akshayaram Srinivasan. Two-round multiparty secure computation from minimal assumptions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 468–499. Springer, 2018.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 21–30. ACM, 2007.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 955–966. ACM, 2013.

[KMR14] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. Flexor: Flexible garbling for xor gates that beats free-xor. In *International Cryptology Conference*, pages 440–457. Springer, 2014.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In *International Colloquium on Automata, Languages, and Programming*, pages 486–498. Springer, 2008.

[Lin11] Yehuda Lindell. Highly-efficient universally-composable commitments based on the ddh assumption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 446–466. Springer, 2011.

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM conference on Electronic commerce*, pages 129–139. ACM, 1999.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P Smart, and Stephen C Williams. Secure two-party computation is practical. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 250–267. Springer, 2009.

[Sec] Inside Secure. White-box software protection. https://www.insidesecure.com/Products/Application-Protection/Software-Protection/WhiteBox. Last accessed on September 26, 2018.

[Unr15] Dominique Unruh. Non-interactive zero-knowledge proofs in the quantum random oracle model. In *Annual International*

Conference on the Theory and Applications of Cryptographic Techniques, pages 755–784. Springer, 2015.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.

[Zho16] Hongsheng Zhong. Secure and trusted partial white-box verification based on garbled circuits. Master's thesis, McMaster University Hamilton, 2016.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 220–250. Springer, 2015.