DUPLO: Unifying Cut-and-Choose for Garbled Circuits*

Vladimir Kolesnikov[1], Jesper Buus Nielsen[2], Mike Rosulek[3], Ni Trieu[3], and Roberto Trifiletti[2](✉)

[1] Bell Labs
kolesnikov@research.bell-labs.com
[2] Aarhus University
{jbn,roberto}@cs.au.dk
[3] Oregon State University
{rosulekm,trieun}@eecs.oregonstate.edu

Abstract. Cut-and-choose (C&C) is the standard approach to making Yao's garbled circuit two-party computation (2PC) protocol secure against malicious adversaries. Traditional cut-and-choose operates at the level of entire circuits, whereas the LEGO paradigm (Nielsen & Orlandi, TCC 2009) achieves asymptotic improvements by performing cut-and-choose at the level of individual gates. In this work we propose a unified approach called DUPLO that spans the entire continuum between these two extremes. The cut-and-choose step in our protocol operates on the level of arbitrary circuit "components," which can range in size from a single gate to the entire circuit itself.

With this entire continuum of parameter values at our disposal, we find that the best way to scale 2PC to computations of realistic size is to use C&C components of intermediate size, and not at the extremes. On computations requiring hundreds of millions of gates or more, our more general approach to C&C gives almost an order of magnitude improvement over existing approaches.

In addition to our technical contributions of modifying and optimizing previous protocol techniques to work with general C&C components, we also provide an extension of the recent Frigate circuit compiler (Mood et al, EuroS&P 2016) to effectively express any C-style program in terms of components which can be processed efficiently using our protocol.

1 Introduction

Garbled Circuits (GC) are currently the most common technique for practical two-party secure computation (2PC). GC has advantages of high performance, low round complexity, low latency, and, importantly, relative engineering simplicity. Both the core garbling technique itself and its application in higher level protocols have been the subject of significant improvement. In the semi-honest model, there have been relatively few asymptotic/qualitative improvements since the original protocols of Yao [Yao86] and Goldreich et al. [GMW87]. The more challenging task of providing security in the presence of *malicious* parties has seen more striking improvements, such as reducing the number of garbled circuits needed for cut-and-choose [LP07, LP11, sS11, Lin13], exploring trade-offs between online and offline computation phases [HKK$^+$14, LR14], and exploring slight weakenings of security [MF06, KMRR15, AO12, KM15]. These improvements have brought the malicious security setting to a polished state of affairs, and even small-factor performance improvements are rare.

* The first author was supported by Office of Naval Research (ONR) contract number N00014-14-C-0113. Third and fourth author partially supported by NSF awards #1149647 and #1617197. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731583 (SODA).

Cut-and-choose. The focus of this work is to unify two leading approaches for malicious security in GC-based protocols, by viewing them as extreme points on a single continuum. We will find that optimal performance — often significantly better than the state-of-the-art — is generally found somewhere in the middle of the continuum. We start with reviewing the idea of cut-and-choose (C&C) and the two existing approaches which we generalize.

According to the "Cut-and-Choose Protocol" entry of the Encyclopedia of Cryptography and Security [TJ11], a (non-zero-knowledge) C&C protocol was first mentioned in the protocol of Rabin [Rab77] where this concept was used to convince a party that the other party sent it a specially formed integer n. The expression "cut and choose" was introduced later by Chaum in [BCC88] in analogy to a popular cake-sharing problem: given a cake to be divided among two distrustful players, one of them cuts the cake in two shares, and lets the other one choose.

Whole-circuit C&C. Recall, Yao's basic GC protocol is not secure against a cheating GC generator, who can submit a maliciously garbled circuit. Today, C&C is the standard tool in achieving malicious security in secure computation. At the high level, it proceeds in two phases.

C&C phase. The GC generator generates a number of garbled circuits and sends them to GC evaluator, who chooses a subset of them (say, half) at random to be opened (with the help of the generator) and verifies their correctness.

Evaluation phase. If all opened circuits were constructed correctly, the players proceed to securely evaluate the unopened circuits, and take the majority (or other protocol-prescribed) output.

A statistical analysis shows that the probability of the GC generator violating security (by making the evaluator accept an incorrect output) is exponentially small in the number of circuits n.

Significant progress has been made [Lin13, HKE13, Bra13, LR14, HKK$^+$14] in reducing the concrete value of n needed to achieve a given failure probability. Specifically, if the evaluation phase of the protocol requires a *majority* of unopened circuits to be correct (as in [sS11]), then $\sim 3s$ circuits are required in total for statistical security 2^{-s}. If the evaluation phase merely requires at least *one* unopened circuit to be correct (e.g., [Lin13, Bra13]), then only s circuits are required for the same security. This multiplicative overhead in garbling material due to replication, the **replication factor**, in the above protocols is $3s$ and s, respectively. In the amortized setting where parties perform N independent evaluations of the same circuit, all evaluations can share a common C&C phase where only a small fraction of circuits needs to be opened. Here, the (amortized) replication factor per evaluation is $O(1) + O(s/\log N)$ for statistical security 2^{-s} [LR14, HKK$^+$14]. As an example, for $N = 1024$ and $s = 40$ the amortized replication factor is around 5.

LEGO. The LEGO paradigm (Large Efficient Garbled-circuit Optimization), introduced by Nielsen & Orlandi [NO09], works somewhat differently. First, the generator produces many independent *garbled gates* (e.g., NAND gates). Similarly to the whole-circuit C&C, the evaluator chooses a random subset of these gates to be opened and checked. Now, the evaluator randomly assigns the unopened gates into *buckets*. The garbled gates in each bucket are carefully combined in a certain way, so that, as long as a majority of gates in each bucket are correct, the bucket as a whole behaves like a *correct* logical garbled NAND gate. These buckets are then assembled into the final garbled circuit, which is finally evaluated.

The extra step in the LEGO paradigm of randomly assigning unopened gates into buckets improves the protocol's asymptotic replication factor. More precisely, if the evaluated function has N gates, then the LEGO protocol has replication factor $2 + O(s/\log N)$ for security 2^{-s} (compared to s or $3s$ for conventional whole-circuit C&C). The main disadvantage of the LEGO approach is that there is a nontrivial cost to connect independently generated gates together ("soldering," in LEGO terminology). Since soldering needs to be performed for each wire of the Boolean circuit, LEGO's asymptotic advantages overtake whole-circuit C&C in performance only for circuits of large size. In Section 3 we give more details about the LEGO paradigm.

1.1 DUPLO: building garbled circuits from big pieces

We introduce DUPLO (**D**UPLO **U**nifying **P**rocedure for **LEGO**), a new approach for malicious-secure two-party computation.

As discussed above, the two standard approaches for malicious-secure 2PC perform C&C at the level of entire circuits (whether in the single-execution setting or in the multi-execution setting [LR15,RR16]), or at the level of individual gates (LEGO). DUPLO is a single unifying approach that spans the *entire continuum between these extremes.* The DUPLO approach performs C&C at the level of arbitrary garbled subcircuits (which we refer to as *components*). After the C&C phase has completed, the parties can use the resulting garbled components in any number of 2PC executions, of any (possibly different) circuits that can be built from these components.

What is the value in generalizing C&C in this way? In short, the DUPLO approach unlocks a new degree of freedom in optimizing practical secure computation. To understand its role, we first review in more detail the costs associated with the C&C techniques (including LEGO).

The most obvious (and often the most significant) cost is the GC replication factor, discussed above. When evaluating a function consisting of N components (either entire circuits, gates, or generalized components explored in this work), the replication factor is $O(1) + O(s/\log N)$, for desired security 2^{-s}. Clearly, using smaller components improves the replication factor, since N is increased.

The replication factor converges to a lower limit of 2 [ZH17]. As the number of components grows, the benefit of amortization quickly reaches its effective maximum. With practical parameters, there is little improvement to be gained beyond a few million components.

It is when the number of components is "maxed out" that the flexibility of DUPLO starts to have its most pronounced effect. There will be a wide range of different component sizes that all give roughly the same replication factor. Among these choices for component size, it is now best to choose the *largest*, thereby reducing the *cost of soldering*, or connecting the components. This cost is proportional to the number of input/output wires of a component (whole-circuit C&C can be also seen this way, since we have special processing for the inputs and outputs). When a circuit is decomposed into *larger* components, a smaller fraction of wires will cross a boundary between components and therefore require soldering.

In other words, we expect a "sweet spot" for ideal component size, and for computations of realistic size this sweet spot is expected to be between the extremes of gate-level and whole-circuit components. We confirm this analysis by the empirical performance of our prototype implementation. We indeed find such a "sweet spot" between the extremes of component size, as we start considering computations with millions of gates. For these realistic problem sizes, the DUPLO approach improves performance by almost an *order of magnitude* over gate-based and circuit-based C&C. Details are given in Section 7.

Is it realistic to express computations in terms of moderately sized components? We note that the C&C components need to garble identical circuits, i.e. be interchangeable in GC evaluation. Indeed, all NAND gates in LEGO and all circuits in whole-circuit C&C are interchangeable in the sense that they are garblings of the same functionality. One may rightly ask whether it is reasonable to expect realistic computations to be naturally decomposable into interchangeable and non-trivial (i.e. not a single-gate or entire-circuit) subcircuits.

We argue that this is indeed a frequent occurrence in standard (insecure) computation. Standard programming language constructs (standard-size arithmetic operations, subroutine calls, loops, etc.) naturally generate identical subcircuits. Given the recent and growing tendency to automate circuit generation and to build 2PC compilers for higher-level languages [MNPS04, HFKV12, ZE15, LWN$^+$15, MGC$^+$16], it is natural to presume that many practical circuits evaluated by 2PC will incorporate many identical components. Specifically, consider the following scenarios:

- Circuits compiled from higher level languages containing multiple calls to the same subroutine (e.g. algebraic calculations), loops, etc. For example, a boolean circuit for matrix multiplication can be expressed in terms of subcircuits for multiplication and addition.
- Two parties know they will perform many secure computations of a CBC-MAC-based construction (e.g., CMAC) using AES as the block cipher, where one party provides the key and the other provides the message to be authenticated. They can use the AES circuit (or a CBC-composition of several AES circuits) as the main DUPLO component, and use as many components as needed for each evaluation of CMAC. Another example involving AES is to consider the AES round function as the DUPLO component. As this is the same function used internally in AES-128, AES-192 and AES-256 (only the key schedule and number of rounds differ) this preprocessing becomes more independent of the final functionality.
- Two parties agree on a predetermined low-level instruction set, where for each instruction (represented as a circuit), the parties can produce a large number of preprocessed garbled components without knowing *a priori* the final programs/functionalities to be computed securely. This CPU/ALU emulation setting has recently been considered in the context of secure computation of MIPS assembly programs [SHS$^+$15, WGMK16]. The DUPLO approach elegantly and efficiently provides a way to elevate these results to the malicious setting.

In Section 7 we investigate several of these scenarios in detail, and compare our performance to that of previous work.

1.2 Related work

Maliciously secure 2PC using Yao's garbled circuit technique has seen dramatic improvements in recent years, both algorithmic/theoretical and implementations. Since the first implementation in [LPS08], tremendous effort has been put into improving concrete efficiency [LP07, NO09, PSSW09, LP11, sS11, HEKM11, KsS12, FJN$^+$13, Bra13, FN13, HKE13, Lin13, MR13, sS13, HMsG13, AMPR14, HKK$^+$14, LR14, LR15, RR16, WMK16, NST17, ZH17, KRW17] yielding current state-of-the-art prototypes able to securely evaluate an AES-128 computation in 6 ms (multi-execution) or 65 ms (single-execution). Multi-execution refers to evaluating the same function several times (either in serial or parallel) on distinctly chosen inputs while the more general single-execution setting treats the computation atomically. In addition, some of these protocols allow for dividing the computation into different phases to utilize preprocessing. In the most general case the computation can be split into three consecutively dependent phases. Following the convention of [NST17] we have:

Function-independent preprocessing depends only on the statistical and computational security parameters s and k. It typically prepares a given number of gates/components that can be used for later computation.

Function-dependent preprocessing uses the previously computed raw function-independent material and stitches it together to compute the desired function f.

Online/Eval phase lastly depends on the parties inputs to the actual computation and is typically much lighter than the previous two phases.

Of notable interest are the protocols of [RR16] and [WMK16] which represent the current state-of-the-art protocols/prototypes for the multi- and single-execution settings, respectively. Both protocols also support function-*dependent* preprocessing. With regards to constant-round function-*independent* preprocessing the works of [NST17, ZH17, KRW17] are the most efficient, however at this time only the work of [NST17] provides a public prototype implementation.

In addition to the above garbled circuit approaches another very active and fruitful research area in secure computation is the secret-sharing based protocols [NNOB12, DPSZ12, DKL+13, KSS13, DZ13, DLT14, LOS14, BLN+15, KOS16, DZ16, DNNR16]. These protocols share a common blueprint in that initially the parties secret share their inputs and interactively compute the function in question. This has the advantage of being less bandwidth demanding than the garbled circuit approach, but at the cost of requiring $\mathcal{O}(\mathsf{depth}(f))$ rounds of interaction to securely evaluate f. This approach also has the benefit of usually supporting function-independent preprocessing and allowing for n participating parties rather natively. In contrast, it seems considerably harder adapting the garbled circuit approach to n-parties [BMR90, LPSY15, LSS16].

The idea of connecting distinct garbled circuits has also previously been studied in [MGBF14] by mapping previous output garbled values to garbled input values in a following computation. Their model and approach is different from ours and is mainly motivated by enabling garbled state to be reusable for multiple computations. Finally we point out the recent work of [GLMY16] for the *semi-honest* case of secure 2PC using garbled circuits. [GLMY16] likewise considers splitting the function of interest into sub-circuits and processes these independently. As there is no cut-and-choose overhead in the semi-honest setting, their approach is motivated primarily by allowing function-independent preprocessing using the garbled components as building blocks. Although the high-level idea is similar to ours, we apply it in a completely different setting and use different techniques. Further, while malicious security is often significantly more expensive, the efficiency gap in the linking and online phase between [GLMY16] and our protocol is surprisingly small. In the application of computing an AES-128 (by preprocessing the required round functions) we see that [GLMY16] sends 82 kB in the online phase (link + evaluate) vs. 88 kB using our protocol. For the offline step the gap is larger due to the overhead of C&C in the malicious case. However utilizing amortization this can be reduced significantly and in some cases be as low as 3-5x that of the semi-honest protocols'. We also highlight that our extension to the Frigate compiler for transforming a high-level C-style program into Boolean circuit sub-components should be directly applicable for this related work. To the best of our knowledge [GLMY16] does not provide such a tool.

1.3 Our Contributions and Outline of the Work

The main contribution of the paper is putting forward and technically and experimentally supporting the idea of generalizing C&C protocols to arbitrary subcircuits. Due to the generality of the approach and the performance benefits we demonstrate, we believe the DUPLO approach will be the standard technique in 2PC compilers. As a lower-level technical contribution, we propose several improvements to garbling and soldering for this setting.

We implemented our solution and integrated it with the state-of-the-art compiler framework Frigate [MGC+16]. Experimentally, we report of a 4-7x improvement in total running time compared to [WMK16] for certain circuits. For the multi-execution setting we also improve the performance of [RR16] by up to $5\times$ in total running time. We accomplish the above while at the same time retaining the desirable preprocessing and reactive capabilities of LEGO.

We start our presentation with a more technical overview of the state of the art in LEGO, including soldering techniques in Section 3. We then present the technical overview of our approach and improvements in Section 4. We present the overview of our DUPLO framework, including several implementation optimizations and the Frigate extensions in Section 6. We report on performance in Section 7.

2 Preliminaries

Our DUPLO protocol is a protocol for 2PC that is secure in the presence of malicious adversaries. We define security for 2PC using the framework of Universal Composition (UC), due to Canetti [Can01]. This framework is demanding, as it guarantees security when such protocols are executed concurrently, in arbitrary environments like the Internet.

A detailed treatment of UC security is beyond the scope of this work. At the high level, security is defined in the real-ideal paradigm. We imagine an ideal interaction, in which parties give their inputs to a trusted third party who computes the desired function f and announces the result. In this interaction, the only thing a malicious party can do is select its input to f. In the real interaction, honest parties interact following the prescribed protocol, while malicious parties may arbitrarily deviate from the protocol. We say that the protocol *securely realizes* f if the real world is "as secure as" the ideal world. More formally, for every adversary attacking the real protocol, there is an adversary (called "simulator") "attacking" the ideal interaction achieving the same effect.

We assume some familiarity with modern garbled circuit constructions, in particular, the *Free-XOR* optimization of Kolesnikov & Schneider [KS08]. This is reviewed in Section 3. Free-XOR garbled circuits are secure under a circular correlation-robust hash assumption [CKKZ12].

3 Overview of the LEGO Paradigm

We now give more details about the mechanics of the LEGO paradigm. Here we describe the MiniLEGO approach of [FJN$^+$13]. We chose MiniLEGO as it is the simplest LEGO protocol to present. At the same time, it contains and conveys all relevant aspects of the paradigm.

3.1 Soldering via XOR-Homomorphic Commitments

The sender generates many individual garbled NAND gates. Each garbled gate g is associated with *wire labels* L_g^0, L_g^1 for the left input wire, labels R_g^0, R_g^1 for the right input wire, and labels O_g^0, O_g^1 for the output wire. Here the superscript of each label indicates the truth value that it represents. In MiniLEGO, all gates are garbled using the Free-XOR optimization of Kolesnikov & Schneider [KS08]. Therefore, there is a global (secret) value Δ so that $L_g^1 = L_g^0 \oplus \Delta$ and $R_g^1 = R_g^0 \oplus \Delta$ and $O_g^1 = O_g^0 \oplus \Delta$. More generally, a wire label K_g^b can be written as $K_g^b = K_g^0 \oplus b \cdot \Delta$. Importantly, the same Δ is used for *all* garbled gates.

The garbled gate consists of the garbled table itself (i.e., for a single NAND gate, the garbled table consists of two ciphertexts when using the scheme of [ZRE15]) along with **XOR-homomorphic commitments** to the "zero" wire labels L_g^0, R_g^0, and O_g^0. A global homomorphic commitment to Δ is also generated and shared among all gates.

To assemble assorted garbled gates into a circuit, the LEGO paradigm uses a technique called **soldering**. Imagine two wires (attached to two unrelated garbled gates) whose zero-keys are A^0 and B^0, respectively. The sender can "solder" these wires together by decommiting to $S = A^0 \oplus B^0$. We require that such a decommitment can be performed given separate commitments to A^0 and B^0, and that the decommitment reveals no more than S. Importantly, S is enough information to allow the receiver to transfer a garbled truth value from the first wire to the second (and vice-versa). For example, if the receiver holds wire label A^b (for unknown b), he can compute

$$A^b \oplus S = (A^0 \oplus b \cdot \Delta) \oplus S = B^0 \oplus b \cdot \Delta = B^b,$$

which is the garbled encoding of the same truth value, but on the other wire.

Gates are assigned to buckets by the receiver, where each bucket, while possibly containing malicious gates, will be assembled to correctly implement the NAND gate. For the gates inside a bucket, the sender therefore solders all their left wires together, all their right wires together, and all their output wires together with the effect that the bucket can operate on a single set of input labels and produce a single set of output labels. For β gates in a bucket, this gives β ways to evaluate the first gate (use solder values to transfer its garbled inputs to the ith bucket gate, evaluate it, then transfer the result back to the first gate). In the most basic form of LEGO, the cut-and-choose ensures that the majority of gates within the bucket are good. Hence the evaluator can evaluate the bucket in β ways and take the majority output wire label. Each bucket therefore logically behaves like a *correct* garbled gate.

The buckets are then assembled into a complete garbled circuit by soldering output wires of one bucket to the input wires of another.

3.2 Recent LEGO Improvements

In recent years several improvements to the LEGO approach has been proposed in the literature. The TinyLEGO protocol [FJNT15] provide several concrete optimizations to the above MiniLEGO protocol, most notably a more efficient bucketing technique. The subsequent implementation [NST17] further optimized the protocol and showed that, combined with the XOR-homomorphic commitment scheme of [FJNT16, CDD$^+$16], the LEGO paradigm is competitive with previous state-of-the-art protocols for malicious 2PC, in particular in scenarios where preprocessing is applicable.

In addition to the above works, the protocol of [ZH17] also explores optimizations of LEGO using a different soldering primitive, dubbed XOR-Homomorphic Interactive Hash (XOR-HIH). This technique has a number of advantages over commitments as they allow for a better probability than MiniLEGO and TinyLEGO of catching cheating in the C&C phase. XOR-HIH also yields buckets only requiring a single "correct" gate, whereas MiniLEGO requires a majority and TinyLEGO requires a mixed majority of gates and wire authenticator gadgets. However, due to the communication complexity of the proposed XOR-HIH instantiation being larger than that of the [FJNT16, CDD$^+$16] commitment schemes, the overall communication complexity of [ZH17] is currently larger than that of TinyLEGO.

4 Overview of Our Construction

DUPLO protocol big picture. At the high level, our idea is to extend the LEGO paradigm to support components of arbitrary size and distinct functionalities, rather than just a single kind of component that is either a single gate or the entire circuit. The approach is similar in many ways to the LEGO protocol and is broken up into three phases.

In the *function-independent phase*, the garbler generates many independent garblings of each kind of component, along with related commitments required for soldering. For each kind of component, the parties perform a cut-and-choose over all garbled components. The receiver asks the garbler to open some fraction of these components, which are checked for correctness. The remaining components are assembled randomly into *buckets.* The soldering required to connect components into a bucket is done at this step.

In the *function-dependent phase*, the parties agree on circuits that can be assembled from the available components. The parties perform soldering that connects different buckets together, forming the desired circuits.

In the *online phase*, the parties have chosen their inputs for an evaluation of one of the assembled circuits. They perform oblivious transfers for the evaluator to receive its garbled input, and the garbler also releases its own garbled inputs. The evaluator then evaluates the DUPLO garbled circuit and receives the result.

Challenges and New Techniques. The seemingly simple high-level idea described above encounters several significant technical challenges in its realization. We address the issues in detail in Section 5. Here we mention that the main challenge is that the LEGO paradigm uses the same Free-XOR offset Δ for all garbled components, and its soldering technique crucially relies on this fact. This is not problematic when components are single gates, but turns out to lead to scalability issues for larger components. As a result, we must change the fundamental garbling procedure, and therefore change the soldering approach.

The TinyLEGO approach uses an *input recovery* technique inspired by [Lin13]. The idea is that if the garbler cheats in some components, then the resulting garbled circuit will either give the correct garbled output, or else it will leak the garbler's entire input! In the latter case, the evaluator can simply evaluate the function in the clear. As above, the TinyLEGO approach to this input recovery technique relies subtly on the fact that the components are small, and as a result it does not scale for large components. We introduce an elegant new technique that works for components of any size, and improves the concrete cost of the input recovery mechanism.

Implementation, Evaluation, Integration. We implemented a high-performance prototype of our protocol to explore the effect of varying component sizes in the C&C paradigm. We study a variety of scenarios and parameter choices and find that our generalizations of C&C can lead to significant performance improvement. Details are given in Section 7.

We have adapted the Frigate circuit compiler of Mood et al. [MGC+16], which compiles a variant of C into circuits suitable for garbled circuit 2PC applications. We modified Frigate so that subroutines are treated as DUPLO components. As an example, a CBC-MAC algorithm that makes calls to an AES subroutine will be compiled into an "outer circuit" built from atomic AES components, as well as an "inner circuit" that implements the AES component from boolean gates. In our implementation, the inner circuits are then garbled as DUPLO components, and the outer circuits are used to assemble the components into high-level functionalities.

5 DUPLO Protocol Details

We now give more details about the challenges in generalizing the LEGO paradigm, and our techniques to overcome them.

5.1 Different Δ's

The most efficient garbling schemes use the Free-XOR optimization of [KS08]. MiniLEGO/TinyLEGO are compatible with Free-XOR, and in fact they enforce that all garbled gates use the same global Free-XOR difference $\varDelta$. However, having a common $\varDelta$ does lead to some drawbacks. In particular, consider the part of the cut-and-choose step in which the receiver chooses some garbled gates to be opened/checked. If we fully open a garbled gate, both wire labels are revealed for each wire. In MiniLEGO, this would reveal $\varDelta$ and compromise the security of the *unopened* gates, which share the same $\varDelta$. To avoid this, the MiniLEGO approach is to make the sender reveal only *one out of the four* possible input combinations to each opened gate (by homomorphically decommitting to the input wire labels). Note that the receiver may now have only a 1/4 probability of detecting an incorrectly garbled gate (the technique of [ZH17] improves this probability to 1/2). The cut-and-choose analysis must account for this probability.

This approach of only partially opening garbled gates does not scale well for large components. If a component has n input wires, then the receiver will detect bad components with probability $1/2^n$ in the worst case. In the DUPLO protocol, we garble each component c with a separate Free-XOR offset $\varDelta_c$ (so each gate inside the garbled component uses $\varDelta_c$, but other garbled components use different offset). Hence, DUPLO components can be *fully opened* in the cut-and-choose phase, while XOR gates are still free inside each component.

As a result:

- Bad components are detected with probability 1, so the statistical analysis for DUPLO cut-and-choose is better than Mini/TinyLEGO by a constant factor.
- We can use a variant of the optimization suggested in [GMS08] to save bandwidth for cut-and-choose. Initially the sender only sends a short hash of each garbled component. Then to open a component, the sender decommits to the input and output keys as well as the $\varDelta_c$ used for garbling the component. Hence, communication for the opened components is minimal.

Adapting soldering. It remains to describe how to adapt the soldering procedure to solder wires with different Free-XOR offsets (the MiniLEGO approach relies on the offsets being the same). Here we adapt a technique of [AHMR15] for soldering wires. Using the point-and-permute technique for garbled circuits [BMR90], the two wire labels for each wire have random and opposite least-significant bits. We refer to this bit as the *color bit* for a wire label. The evaluator sees the color bit of a wire, but not the truth value of a wire.

In MiniLEGO, the garbler commits to the “zero-key” for each wire, which is the wire label encoding *truth value false.* In DUPLO, we have the garbler generate homomorphic commitments to the following:

- For each wire, commit to the wire label *with color bit zero.* In this section we therefore use notation K^b to denote a wire label with *color bit* (not necessarily truth value) b.
- For each wire, commit to an indicator bit σ for each wire that denotes the color bit of the *false* wire label. Hence, wire label K^b has truth value $b \oplus \sigma$.
- For each component c, commit to its Free-XOR offset $\varDelta_c$.

Consider a wire i with labels $(K_i^0, K_i^1 = K_i^0 \oplus \varDelta_i)$ and indicator bit σ_i, and another wire j in a different component with labels $(K_j^0, K_j^1 = K_j^0 \oplus \varDelta_j)$ and indicator bit σ_j. To solder these wires together, the garbler will give homomorphic decommitments to the following solder values:

$$s^\sigma = \sigma_i \oplus \sigma_j; \quad S^K = K_i^0 \oplus K_j^0 \oplus s^\sigma \cdot \varDelta_j; \quad S^\varDelta = \varDelta_i \oplus \varDelta_j$$

Note that the decommitment to $S^\varDelta$ can be reused for all wires soldered between these two components. Now when the evaluator learns wire label K_i^b (with color bit b visible), he can compute:

$$\begin{aligned}K_i^b \oplus S^K \oplus b \cdot S^\Delta &= K_i^b \oplus (K_i^0 \oplus K_j^0 \oplus s^\sigma \cdot \Delta_j) \oplus b \cdot (\Delta_i \oplus \Delta_j)\\ &= b \cdot \Delta_i \oplus (K_j^0 \oplus s^\sigma \cdot \Delta_j) \oplus b \cdot \Delta_i \oplus b \cdot \Delta_j\\ &= K_j^0 \oplus (s^\sigma \oplus b) \cdot \Delta_j = K_j^{s^\sigma \oplus b}\end{aligned}$$

Also note that a common truth value has opposite color bits on wires i & j if and only if $s^\sigma = \sigma_i \oplus \sigma_j = 1$. Hence, the receiver obtains the wire label $K_j^{s^\sigma \oplus b}$ which encodes the same truth value as K_i^b.

DUPLO bucketing. In Section 3.1 we described how [FJN+13] used a bucket size that guaranteed a majority of correct AND gates in each bucket. In this work we use the original bucketing technique of [NO09] that only requires a single correct component in each bucket, but requires a majority bucket of wire authenticator (WA) gadgets on each output wire. The purpose of a WA is to accept or reject a wire label as "valid" without revealing the semantic value on the wire, and as such a simple construction can be based on a hash function and C&C. A WA consists of a "soldering point" (homomorphic commitments to a Δ and a zero-key), along with an unordered pair $\{\mathcal{H}(K_i^0), \mathcal{H}(K_i^0 \oplus \Delta)\}$. A wire label K can be authenticated checking for membership $\mathcal{H}(K) \in \{\mathcal{H}(K_i^0), \mathcal{H}(K_i^0 \oplus \Delta)\}$. In order to defeat cheating a C&C step is carried out on the WAs to ensure that a majority of any WA bucket only accepts committed wire labels. The choice of using WAs in this work is motivated by the fact that DUPLO components can be of arbitrary size and are often much larger than a single gate. By requiring fewer such components in total, we therefore achieve much better overall performance as WAs are significantly cheaper to produce in comparison to garbled components.

Avoiding commitments to single bits. We also point out that the separate commitments to the zero-label K_i^0 and the indicator bit σ_i can be combined into a single commitment. The main idea is that the least significant bit of K_i^0 is always zero (being the wire label with color bit zero). Similarly, when using Free-XOR, the offset Δ must always have least significant bit 1. Hence in the solder values S and S^Δ, the evaluator knows *a priori* what the least significant bit will be. We can instead use the least significant bits of the K_i^0 commitments to store the indicator bit σ_i so that homomorphic openings convey $\sigma_i \oplus \sigma_j$. This approach saves s bits of communication per wire commitment over the naive approach of instantiating the bit-commitments using [FJNT16] using a bit-repetition code with length s.

In the online evaluation phase, the garbler decommits to the indicator bits of the evaluators designated input and output. In this case, the garbler does not want to decommit the entire wire label as this would potentially let the evaluator learn the global difference Δ (if the evaluator learned the opposite label through the OTs or evaluation). To avoid this, we have the garbler generate many commitments to values of the special form $R\|0$ for random $R \in \{0,1\}^{\kappa-1}$. Using the homomorphic properties of these commitments, this can be done efficiently by having the garbler decommit s random linear combinations of these commitments to ensure that *all* of them have the desired form with probability $1 - 2^{-s}$. Then when the garbler wants to decommit to a wire label's indicator bit only, it gives a homomorphic decommitment to the wire label XOR a mask $R\|0$, which hides everything but the indicator bit.

5.2 Improved Techniques for Circuit Inputs

We also present a new, more efficient technique for input recovery. The idea of input recovery [Lin13] is that if the sender in a 2PC protocol cheats, the receiver will learn the sender's input (and can hence compute the function output).

Within each DUPLO bucket, the cut-and-choose guarantees at least one correctly garbled component and a majority of correct output-wire authenticators. As such, the evaluator is guaranteed to learn, for each output wire of a component, either 1 or 2 *valid* garbled outputs. If only one garbled output is obtained, then it is guaranteed to be the correct one. Otherwise, the receiver learns both wire labels and hence the Free-XOR offset $\varDelta_c$ for that component. The receiver can then use the solder values to iteratively learn *both* wire labels on *all* wires in the circuit (at least all the wires in the connected component in which the sender cheated).

However, knowing both wire labels does not necessarily guarantee that the receiver learns their corresponding *truth values*. We need a mechanism so that the receiver learns the truth value for the sender's garbled inputs.

Our approach is to consider special input-components. These consist of an *empty garbled circuit* but homomorphic commitments to a zero-wire-label K and a Free-XOR offset $\varDelta$ that serve as soldering points. Suppose for every input to the circuit, we use such an input component that is soldered to other components. The sender gives his initial garbled input by homomorphically decommitting to either the zero wire-label K or $K \oplus \varDelta$. If the sender cheats within the computation, the receiver will learn $\varDelta$. The key novelty in our approach is to use **self-authenticating wire labels.** In an input-gadget, the false wire label must be $H(\varDelta)$ and the true wire label must be $H(\varDelta) \oplus \varDelta$ (the sender will still commit to whichever has color bit zero). Then when the sender cheats, the receiver learns $\varDelta$, and can determine whether the sender initially opened $H(\varDelta)$ (false) or $H(\varDelta) \oplus \varDelta$ (true).

This special form of wire labels can be checked in the cut-and-choose for input components. In the final circuit, we assemble input-components into buckets to guarantee that a majority within each bucket is correct. Then the receiver can extract a cheating sender's input according to the majority of input-components in a bucket.

5.3 Formal Description, Security

Our protocol implements secure reactive two-party computation [NR16], *i.e.*, the computation has several rounds of secret inputs and secret outputs, and future inputs and as well as the specification of future computations might depend on previous outputs.

To be more precise, let $\mathcal{F}$ denote the ideal functionality $\mathcal{F}_{\mathsf{R2PC}}^{\mathbb{L},\varPhi}$ in Fig. 9 on page 1040 in [NR16]. Recall that this functionality allows to specify a reactive computation by dynamically specifying the functionality of sub-circuits and how they are linked together. The command $(\textsc{Func}, t, f)$ specifies that the sub-circuit identified by t has circuit f. The command $(\textsc{Input}, t, i, x)$ gives input x to wire i on sub-circuit t. Only one party supplies x, the other party inputs $(\textsc{Input}, t, i, ?)$ to instruct $\mathcal{F}$ that the other party is allowed to give an input to the specified wire. The command defines the wire to have value x. The command $(\textsc{Link}, t_1, i_1, t_2, i_2)$ specifies that output wire i_1 of sub-circuit t_1 should be soldered on input wire i_2 of sub-circuit t_2. When an output value becomes defined to some x, this in turn defines the linked input wire to also have value x. The command $(\textsc{Garble}, t, f)$ evaluates the sub-circuit t. It assumes that all the input wires have already been defined. It runs f on these values and defines the output wires to the outputs of f. There are also output commands that allow to output the value of a wire to a given party. They may be called only on wires that had their value defined.

The set $\mathbb{L}$ allows to restrict the set of legal sequences of calls to the functionality. We need the restriction that all $(\textsc{Func}, t, f)$ commands are given before any other command. This allows us to compute how many times each f is used and do our preprocessing. The function $\varPhi$ allows to specify how much information about the inputs and outputs of $\mathcal{F}$ is allowed to leak to the adversary. We need the standard setting that we leak the entire sequence of inputs and outputs to the adversary, except that when an honest party has input $(\textsc{Input}, t, i, x)$, then we only leak $(\textsc{Input}, t, i, ?)$ and when an honest party has output $(\textsc{Output}, t, i, y)$, then we only leak $(\textsc{Output}, t, i, ?)$.

With many components, many buckets, and many 2PC executions, the formal description of our protocol is rather involved. It is therefore deferred to Appendix A while we in Appendix B prove the following theorem.

Theorem 1. *Our protocol implements $\mathcal{F}$ in the UC framework against a static, poly-time adversary.*

6 System Framework

In this section we give an overview of the DUPLO framework and our extension to the Frigate compiler that allows to transform a high-level C-style program into a set of boolean circuit components that can be fed to the DUPLO system for secure computation. We base our protocol on the recent TinyLEGO protocol [FJNT15], but adapted for supporting larger and distinct components. Our protocol has the the following high-level interface:

Setup A one-time setup phase that initializes the XOR-homomorphic commitment protocol.

PreprocessComponent(n, f) produces n garbled representations F_j of f that can be securely evaluated.

PrepareComponents(i) produces i input authenticators that can be used to securely transfer input keys from garbler G to evaluator E. In addition, for all F_j previously constructed using PreprocessComponent, this call constructs and attaches all required output authenticators. These gadgets ensure that only a single valid key will flow on each wire of all garbled components (otherwise the evaluator learns the generator's private input).

Build$(\mathcal{C})$ Takes a program $\mathcal{C}$ as input, represented as a DAG where nodes consist of the input/output wires of a set of (possibly distinct) components $\{f_i\}$ and edges consist of links from output wires to input wires for all of these f_i's. The Build call then looks up all previously constructed F_j for each f_i and stitches these together using the XOR-homomorphic commitments so that they together securely compute the computation specified by $\mathcal{C}$. This call also precomputes the required oblivious transfers (OTs) for transferring E's input securely.

Evaluate(x, y) Given the plaintext input x of garbler G and y of evaluator E, the parties can now compute a garbled output Z, representing the output of the $f(x, y)$. The system allows both parties to learn the full output, but also distinct output, *e.g.* G can learn the first half of $f(x, y)$ and E learn the second half.

Decode Finally the system allows the parties to decode their designated output. The reason why we have a dedicated decode procedure is to allow partial output decoding. Based on the decoded values the parties can then start a new secure computation on the remaining non-decoded output, potentially adding fresh input as well. The input provided and the new functionality to compute can thus depend on the partially decoded output. This essentially allows branching within the secure computation.

Following the terminology introduced in [NST17] we have that the Setup, PreprocessComponent, and PrepareComponents calls can be done independently of the final functionality $\mathcal{C}$. These procedures can therefore be used for function-independent preprocessing by restricting the functionality $\mathcal{C}$ to be expressible from a predetermined set of instructions. The Build procedure clearly depends on $\mathcal{C}$, but not on the inputs of the final computation, so this phase can implement function-dependent preprocessing. Finally the Evaluate and Decode procedures implement the online phase of the system and depend on the previous two phases to run.

For a detailed pseudocode description of the system as well as a proof of its security we refer the reader to Appendix A and Appendix B, respectively.

6.1 Implementation optimizations

As part of our work we developed a prototype implementation in C++ using the latest advances in secure computation engineering.[4] As the basis for our protocol we start from the libOTe library for efficient oblivious transfer extension [Rin]. As we in this work require UC XOR-homomorphic commitments to the input and output wires of all components we instantiate our protocol with the efficient construction of [FJNT16] which is implemented as the SplitCommit C++ library.[5]

As already mentioned, our protocol is described in detail in Appendix A. However, for reasons related to efficiency our actual software implementation deviates from the high-level description in several aspects

- In the homomorphic commitment scheme of [FJNT16], commitments to random values (chosen by the protocol itself) are cheaper than commitments to values chosen by the sender. Hence, whenever applicable we let the committed key-values be defined in this way. This optimization saves a significant amount of communication since the majority of commitments are to random values.
- Along the same lines we heavily utilize the batch-opening mechanism described in [FJNT16]. The optimization allows a sender to decommit to n values with total bandwidth $n\kappa+\mathcal{O}(s)$ as opposed to the naive approach which requires $\mathcal{O}(n\kappa s)$.
- In the PrepareComponents step we construct all output-wire key authenticators using a single global difference Δ_{ka}. This saves a factor $2x$ in terms of the required number of commitments and solderings, at the cost of an incorrect authenticator only getting caught with probability $1/2$ (as opposed to prob. 1 using distinct differences). However as the number of required key authenticators depends on the total number of output wires of all garbled components the effect of this difference in catching probability does not affect performance significantly when considerings many components.

In addition to the above optimizations, our implementation takes full advantage of modern processors' multi-core capabilities and instruction sets. We also highlight that our code leaves a substantially lighter memory footprint than the implementation of [NST17] which stores all garbled circuits and commitments in RAM. In addition to bringing down the required number of commitments on the protocol level, our implementation also makes use of disk storage in-between batches of preprocessed component types. This has the downside of requiring disk reads of the garbled components during the online phase, but we advocate that the added flexibility and possibility of streaming preprocessing is well worth this trade-off in performance.

6.2 Frigate Extension

The introduction of Fairplay [MNPS04], the first compiler targeted for secure computation (SC), has stimulated significant interest from the research community. Since then, a series of new compilers with enhanced performance and functionality have been proposed, such as CBMC [HFKV12], Obliv-C [ZE15], and ObliVM [LWN$^+$15]. Importantly, the state-of-the-art compiler, Frigate [MGC$^+$16], features a modular and extensible design that simplifies the circuit generation in secure computation. Relying on its rich language features, we provide an extension to the original Frigate framework, in which we divide the **specific** input program into distinct functions. We can then generate a circuit representation for each function which is fully independent from the circuit representation of other functions. Due to this independence

[4] Available at `https://github.com/AarhusCrypto/DUPLO`

[5] Available at `https://github.com/AarhusCrypto/SplitCommit`

we can easily garble each distinct function separately using the DUPLO framework and afterwards solder these back together such that they compute the original source program. As an additional improvement, which is tangential to the main thrust of this work, we construct an AES module that optimizes the number of uneven gates (all even gates can be garbled and evaluated without communication using *e.g.* [ZRE15]).

In the following, we describe the details of our compiler extension. Similar to the Frigate output format, our circuit output contains a set of input and output calls, gate operations, and function calls. The input and output calls consist of wires, which we enumerate and manage. We also use wires to represent declared variables in the source program. Each wire (or, rather its numeric id) is placed in a pool, and is ready for use whenever a new variable is introduced. Our function representation however differs from that of Frigate. In that work, each function reserves a specific set of wire values which requires no overlap among the functions' wires. As a result, Frigate's function representation is dependent on that of other functions. We remove this dependency by creating and managing separate wire pools for each function. In particular, every time a variable is introduced, our compiler searches for the free wires with the smallest indices in the pool of the current working function. Similarly to the original Frigate, our compiler will free the wires it can after each operation or variable assignment. Hence, our function is represented independently of other functions.

We now describe our strategy for constructing our optimized AES circuit. A key component of AES is the Rijndael S-Box [DR02] which is a fixed non-linear substitution table used in the byte substitution transformation and the key expansion routine. The circuit optimization in our AES-128 source program is described in the context of this S-Box. We note that if we generate the S-Box dynamically using the Frigate compiler, this will not optimize the number of uneven gates substantially. Hence, we create an AES-128 source program that embed a highly optimized S-Box circuit statically. To the best of our knowledge, [BP09] presents one of the most efficient S-Box circuit representation which contains only 32 uneven gates in a total of 115 gates. Therefore, we integrate this S-Box into our AES-128 source program, which allows our Frigate extension to optimize the number of uneven gates. For the *key-expanded* AES-128 circuit, which takes a 128-bit plaintext and ten 128-bit round keys as input and outputs a 128-bit ciphertext, this results in 5,120 uneven gates. This is almost a 2x reduction compared the AES-128 circuit originally reported in Frigate. Furthermore, our AES-128 circuit has 640 fewer uneven gates than the circuit reported in TinyGarble [SHS+15] which is the current best compiler written in Verilog. For completeness we note that for the *non-expanded* version of AES-128, our compiled circuit results in 6,400 uneven gates.

7 Performance

In order to evaluate the performance of our prototype we run a number of experiments on a single server with simulated network bandwidth and latency. The server has two 36-core Intel(R) Xeon(R) E5-2699 v3 2.30 GHz CPUs and 256 GB of RAM. That is, 36 cores and 128 GB of RAM per party. As both parties are run on the same host machine we simulate a LAN and WAN connection using the Linux *tc* command: a LAN setting with 0.02 ms round-trip latency, 1 Gbps network bandwidth; a WAN setting with 96 ms round-trip latency, 200 Mbps network bandwidth.

For both settings, the code was compiled using GCC-5.4. Throughout this section, we performed experiments with a statistical security parameter $s = 40$ and computational security parameter $k = 128$. The running times recorded are an average over 10 trials.

We demonstrate the scalability of our implementation by evaluating the following circuits:

AES-128 circuit consisting of 6,400 AND gates. The circuit takes a 128-bit key from one party and a 128-bit block from another party and outputs the 128-bit ciphertext to both.

(Note that this functionality is somewhat artificial for secure computation as the AES function allows decryption with the same key; thus the player holding the AES key can obtain the plaintext block. We chose to include the ciphertext output to the keyholder to measure and demonstrate the performance for the case where both parties receive output.)

CBC-MAC circuit with different number of blocks $m \in \{16, 32, 64, 128, 256, 1024\}$ using AES-128 as the block cipher. The circuit therefore consists of $6,400m$ AND gates. The circuit takes a 128-bit key from one party and m 128-bit blocks from another party and outputs a 128-bit block to both.

Mat-Mul circuit consisting of around 4.2 million AND gates. The circuit takes one 16×16 matrix of 32-bit integers from each party as input and outputs the 16×16 matrix product to both.

Random circuit consisting of 2^n AND gates for various n where topology of the circuit is chosen at random. The circuit takes 128-bit input from each party and outputs a 128-bit value to both.

7.1 Effect of Decomposition

In this section we show how DUPLO scales for the above-mentioned circuits, when considering subcomponents of varying size. As discussed in Section 1.1, we expect the performance of our protocol to be optimal for a subcomponent size somewhere inbetween the extremes of whole-circuit and gate-level C&C. We empirically validate this hypothesis by running two kinds of experiments, one for the randomly generated circuits and one for the real-world AES-128, CBC-MAC-16 and Mat-Mul circuits. The purpose of the random circuit experiment is to explore the trade-offs in overall performance between different decomposition strategies. For the latter experiment we aim to find their optimal decomposition strategy, both to see how this aligns to the random circuit experiment, but also for use in our later performance comparison in Section 7.2.

Random Circuits. In order to build a random circuit consisting of 2^n AND gates that is easily divisible into different subcomponent sizes we initially generate a number of smaller random circuit containing 2^t AND gates with 256 input wires and 128 output wires. This is done by randomly generating non-connected XOR and AND until exactly 2^t AND gates have been generated. Then for each of these generated gates i we assign their two input wires at random from the set of gates with index smaller than i (the gate id i is also the gate's output wire). Finally we solder 2^{n-t} copies of these components together into a final circuit C, thus consisting of 2^n AND gates overall. We consider $n \in \{10, 12, 14, 16, 18, 19, 20\}$ in this experiment, and for each of these we build a circuit of size 2^n using several values of t.

As we are only considering relative performance between different strategies in these experiments we run our implementation using a single thread for each party on the previously mentioned LAN setup.[6] We summarize our findings in Figure 1. The x-axis of the figure represents the continuum from whole-circuit C&C ($t = n$) towards gate-level C&C ($t = 0$). The overall trend of our experiments is strikingly clear, initially as the number of subcomponents increases (t decreases) the running time goes down as well due to our protocol taking advantage of the amortization benefits offered by the LEGO paradigm. However for all circuit sizes considered it is also apparent that at some point this benefit is outweighed by the overhead of soldering and committing to the increasing number of input/output wires between the components. It is at exactly this point (the vertex of each graph), in the sweet spot between

[6] For best absolute performance, we would always run our implementation using several threads per party.

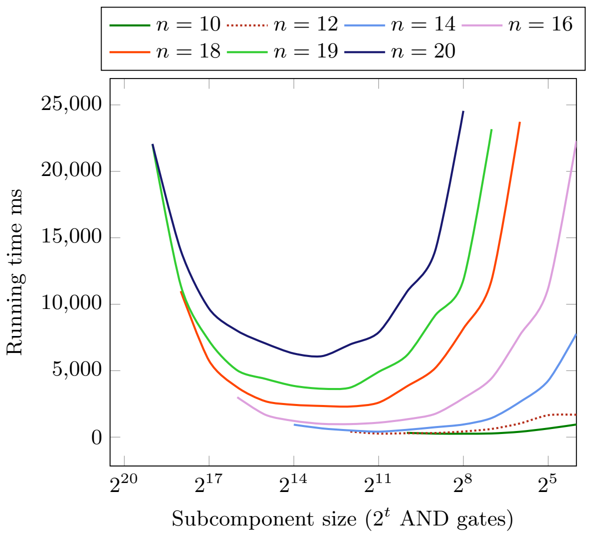

Fig. 1. DUPLO performance for random circuits consisting of 2^n AND gates divided into 2^{n-t} subcomponents.

substantial LEGO amortization and low soldering overhead, that DUPLO has it's optimal performance. We thus conclude that for an ideally decomposable circuit such as the ones generated in this experiment the viability of the DUPLO approach is apparent.

Real-world circuits. The experiments for the random circuits show that the DUPLO approach for C&C does have merit for circuits that can be divided into multiple identical subcomponents. Clearly, this is a very narrow class of functions so in addition we also evaluate our prototype on the previously mentioned real-world circuits in order to investigate their optimal decomposition strategy. We first describe our approach of dividing these circuits into subcomponents.

AES-128 We consider the following three strategies:

- Five kinds of subcomponents: each computing one of the functions of the AES algorithm, that is 1x Key Expansions (1,280 AND gates), 11x AddRoundKey, 10x SubBytes (512 AND gates), 10x ShiftRows, and 9x MixColumns.
- Three kinds of subcomponents: 1x Key Expansions and Initial Round (1,280 AND gates); 9x AES Round Functions (each 512 AND gates); 1x AES Final Round (512 AND gates).
- A single component consisting of the entire AES-128 circuit (6,400 AND gates), *i.e.* whole-circuit C&C.

CBC-MAC-16 We consider decomposing this circuit into a single subcomponent of varying size. In each case, the component contains $i \in \{16, 8, 4, 2, 1\}$ AES-128 blocks, meaning each of these consists of $6,400i$ AND gates.

Mat-Mul In order to multiply two matrices A, B use the block-matrix algorithm: We divide A, B into $m \times m$ 32-bit submatrices $A_{i,j}, B_{i,j}$ for $i, j \in [1, 16/m]$. To compute AB, the block entries $A_{i,k}$ are first multiplied by the block entries $B_{k,j}$ for $k \in [1, m]$, while summing the results over k. It is therefore the case that the experiment contains two different kinds of components, $m \times m$ 32-bit matrix product and $m \times m$ 32-bit matrix addition. In our experiment we consider block matrix sizes $m \in \{16, 8, 4, 2\}$ and the concrete number of AND gates for each kind of component are reported in Table 1.

Block Size	Component Size		Number Executions N		
	Mult	Add	1	32	128
2x2	8,192	124	**11,160**	7,815	7,554
4x4	65,536	496	14,847	**7,539**	**6,622**
8x8	524,288	1,984	52,334	9,615	7,324
16x16	4,194,304	0	351,002	11,338	9298

Table 1. Component sizes and amortized running time per execution for Mat-Mul (ms). Best performance marked in bold.

When performing $N = 1, 32, 128$ executions of AES-128 in parallel, we observe that our protocol performs best when considering the entire circuit as a single component. This is in contrast to what we observed in the random circuit experiment, but can be explained by the non-uniformity of the considered decomposition strategies. The fact that we split the AES-128 into three or five relatively small subcomponents, some of which are only used once, has a very negative influence on DUPLO performance as there is some overhead associated with preparing each component type while at the same time no LEGO-style amortization can be exploited when preparing only a single copy.

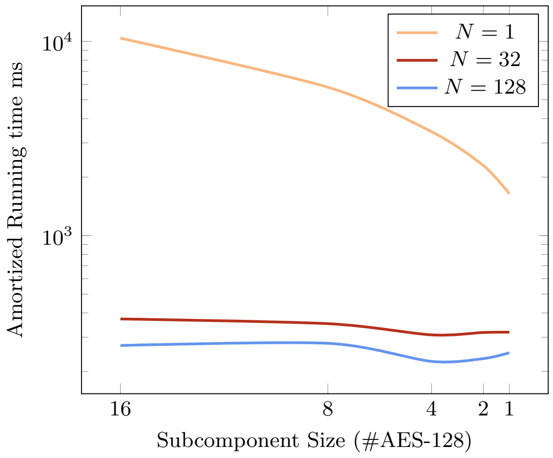

Fig. 2. DUPLO performance for $N = 1, 32, 128$ parallel executions of the CBC-MAC-16 circuit using different decomposition strategies.

For the CBC-MAC-16 circuit however whole-circuit C&C is not the optimal approach and we summarize the observed performance for the different decompositions in Figure 2. Here we see that the best strategy is to decompose the circuit into many *identical* subcomponents. The trend observed is similar to the random circuit experiments where initially it is best to optimize for many identical subcomponents. In particular for a single execution of CBC-MAC-16 it is best to decompose into 16 copies of the AES-128 circuit yielding around 5x performance increase over the whole-circuit approach. For the parallel executions (which contain overall many more AES-128 circuits) we can see that it is best to consider subcomponents consisting of 4xAES-128 circuits each. The lower relative performance difference between the strategies for the parallel executions is due to there being a minimum of N circuits for utilizing LEGO amortization, even for the whole-circuit approach. However as the number of total subcomponents grow it can be seen that there are savings to be had by grouping executions together.

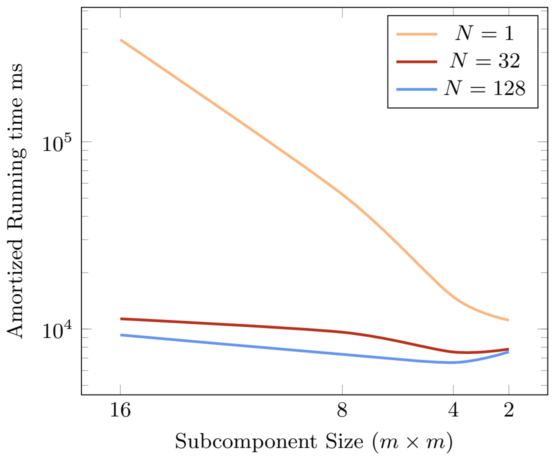

Fig. 3. DUPLO performance for $N = 1, 32, 128$ parallel executions of the Mat-Mul circuit using different decomposition strategies.

Finally for the Mat-Mul circuit we see a similar overall trend as in the CBC-MAC-16 experiment (Table 1 and Figure 3). Most notably is the performance increase for a single execution yielding around 31x by considering blocks of size 2×2 instead of a single whole-circuit 16×16. This experiment indeed highlights the performance potential of the DUPLO approach for large computations that can naturally be decomposed into distinct repeating subcomponents, in this case matrix product and matrix addition. This is in contrast to the previous AES-128 example where this approach was penalized. The difference however is that in the Mat-Mul experiment each subcomponent is repeated several times and therefore all benefit from LEGO amortization.

Experiment Discussion. The above real-world examples show that the DUPLO approach has merit, but the exact performance gains depend significantly on the circuit in question. As a general rule of thumb DUPLO performs best when the circuit can be decomposed into many identical subcomponents as can be seen from the CBC-MAC-16 and Mat-Mul experiments (the more the better). As there is no immediate way of decomposing the AES-128 circuit in this way, we see that performance suffers when the circuit cannot be decomposed into distinct *repeating* parts. However the Mat-Mul experiments show that decomposing the circuit into distinct circuits can certainly have merit, however it is crucial that each subcomponent is repeated a minimal number of times or the non-repeating part of the computation is relatively small.[7]

7.2 Comparison with Related Work

We also compared our prototype to three related high-performance open-source implementations of malicious-secure 2PC. All experiments use the same hardware configuration described

[7] This is not the case for the AES-128 circuit as the non-repeating part consists of around 40% of the entire computation.

at the beginning of this section. For all experiments we have tried tuning the calling parameters of each implementation to obtain the best performance.

When reporting performance of our DUPLO protocol, we split the offline part of the computation into an independent preprocessing (Setup + PreprocessComponent + PrepareComponents) whenever our analysis shows that dividing the computation into subcomponents is optimal — *i.e.*, when evaluating AES-128 we do not have any function-independent preprocessing since the optimal configuration is to let the component consist of the entire circuit. We summarize our measured timings for all the different protocols in Table 2, and now go into more detail:

Protocol	Setting	N	AES			CBC-MAC-16			Mat-Mul		
			Ind. Prep	Offline	Online	Ind. Prep	Offline	Online	Ind. Prep	Offline	Online
WMK	LAN	1	✗	(✗)	125	✗	(✗)	1,177	✗	(✗)	43,930
	WAN	1	✗	(✗)	2,112	✗	(✗)	11,443	✗	(✗)	368,190
RR	LAN	1	✗	198	8.83	✗	3,495	35.52	✗	120,200	913
		32	✗	67.77	3.60	✗	1,296	20.71	✗	36,437	1,247
		128	✗	40.70	2.86	✗	863	18.38	-	-	-
		1024	✗	24.90	3.06	✗	471	17.48	-	-	-
	WAN	1	✗	941	527	✗	12039	565	✗	467,711	1,550
		32	✗	311	472	✗	4202	471	✗	157,928	1,677
		128	✗	192	557	✗	2743	573	✗	-	-
		1024	✗	115	577	✗	1762	597	✗	-	-
NST	LAN	1	**1,506**	**22.34**	**2.54**	2,594	230	**18.14**	-	-	-
		32	**119**	**2.42**	**0.22**	965	38.63	**1.34**	-	-	-
		128	**75.64**	**2.08**	**0.16**	922	37.90	**0.87**	-	-	-
		1024	**60.48**	**1.85**	**0.14**	-	-	-	-	-	-
	WAN	1	**9,325**	223	195	13,812	699	219	-	-	-
		32	**599**	15.17	6.71	4,158	151	8.54	-	-	-
		128	**341**	12.75	6.24	3,810	148	7.15	-	-	-
		1024	**256**	11.81	5.56	-	-	-	-	-	-
DUPLO	LAN	1	✗	371	8.62	**799**	**29.83**	41.94	**10,268**	**569**	**118**
		32	✗	47.03	0.65	**303**	**10.18**	3.72	**7,124**	**331**	**83.73**
		128	✗	27.77	0.41	**213**	**11.54**	2.49	**6,260**	**303**	**58.65**
		1024	✗	17.58	0.30	**175**	**13.30**	**1.61**	-	-	-
	WAN	1	✗	7,391	585	**8,970**	1,370	620	**50,856**	**1,775**	**744**
		32	✗	347	19.37	**1,477**	**49.21**	23.62	**32,098**	**517**	**135**
		128	✗	148	5.55	**990**	**22.23**	8.85	**27,613**	**388**	**101**
		1024	✗	74.03	1.53	**733**	**15.93**	**3.83**	-	-	-

Table 2. All timings are ms per circuit. Best results are marked in bold. Cells with "✗" denote setting not supported. Cells with "-" denote program out of memory.

Better Amortization by Subdivision. The protocol of Rindal & Rosulek (RR in our tables) [RR16] is currently the fastest malicious-secure 2PC protocol in the multi-execution setting.

The protocol of Nielsen et al. (NST) [NST17] is the fastest that allow for function-independent preprocessing, using the LEGO paradigm.[8]

The general trend in Table 2 is that as the total complexity (combined cost of all computations) grows, the efficiency of the DUPLO approach becomes more and more apparent. For example, DUPLO is 1.5x times faster (counting total offline+online time) than RR whole-circuit C&C for 1024 AES-128 LAN. For the larger CBC-MAC-16 scenario, the difference 2.5x. For the even larger case of 32 Mat-Mul executions, the difference is 5x. Our experiments clearly confirm that DUPLO scales significantly better than state-of-the-art amortizing protocols.

When comparing to the LEGO C&C protocol of NST things are harder to compare as they use a much slower BaseOT implementation than we do (1200 ms vs. 200 ms) which especially matters for lower complexity computations. However even when accounting for this difference, in total time, our approach has 2-3x better total performance for AES-128. We note that if Ind. Prep. is applicable for an application then DUPLO cannot compete with NST for small computations, but as demonstrated from our CBC-MAC-16 experiments, once the computation reaches a certain size and we can decompose the target circuit into smaller subcomponents, DUPLO overtakes NST in performance by a factor 5x.

It is interesting to note that the online time of NST is vastly superior to RR and DUPLO, especially for small circuits (2-4x). This is due to the difference between whole-circuit C&C and gate-level C&C where the NST bucket size is relatively small (and thus online computation) even for a single circuit, whereas it needs to be 5-10x larger for the whole-circuit approach. As the number of executions increase we however see that this gap decreases significantly. We believe the reason why NST still outperforms DUPLO in all online running times is that the NST implementation is RAM only, whereas DUPLO writes components and solderings to disk in the offline phases and reads them in the online phase as needed. For RR we notice some anomalies for their online times that we cannot fully explain. We conclude that the throughput measured and reported in our experiments might not be completely fair towards the RR protocol, but might be explained by implementation decisions that work poorly for our particular scenarios. In any case, we do expect DUPLO to perform as fast or faster than RR in the online phase due to less online rounds and data transfer.

Setting	Protocol	CBC-MAC-32		CBC-MAC-64		CBC-MAC-128		CBC-MAC-256		CBC-MAC-1024	
		Offline	Online	Offline	Online	Offline	Online	Offline	Online	Offline	Online
LAN	**WMK**	(✗)	2,298	(✗)	4,539	(✗)	9,029	(✗)	18,003	(✗)	71,787
	DUPLO	**1,211**	**68.29**	**1,877**	**104**	**2,991**	**196**	**5,072**	**274**	**14,167**	**1,003**
WAN	**WMK**	(✗)	21,460	(✗)	41,114	(✗)	79,157	(✗)	155,995	(✗)	606,329
	DUPLO	**12,269**	**656**	**15,039**	**698**	**19,089**	**793**	**27,093**	**910**	**81,883**	**1,733**

Table 3. Comparison for CBC-MAC-XX. All timings are ms per circuit. Best results are marked in bold. "(✗)" denotes the setting is supported, but we only ran the "everything online" version of WMK.

[8] The recent 2PC protocol of [KRW17] appears to surpass NST in terms of performance in this setting, but as this implementation is not publicly available at the time of writing we do not consider it for these experiments.

Amortized-grade Performance for Single-Execution. The current fastest protocol for single-execution 2PC is due to Wang et al. (WMK) [WMK16]. When comparing to their protocol, we ran all experiments using the "everything online" version of their code since this typically gives the best overall running time. We stress however that the protocol also supports a function-dependent preprocessing phase, but since this is not the primary goal of that work we omit it here.

Unsurprisingly, the protocols designed for the multi-execution settings (including DUPLO) are significantly faster than WMK when considering several executions. However, even in the single-execution setting, we see that DUPLO scales better and eventually catches up to the performance of WMK for large computations. WMK is 3x faster than DUPLO when the subcomponent is an entire AES-128 circuit. Then, already for CBC-MAC-16 the ability to decompose this into 16 independent AES-128 circuits yields around 1.4x factor improvement over WMK. We further explore this comparison in Table 3, by evaluating even larger circuits in the single-execution setting. For larger CBC-MAC circuits, DUPLO is around 4.7x faster on LAN and 7.4x on WAN.

Protocol	#Execs	Ind. Prep	Dep. Prep	Online
WMK	1	✗	✗	9.66 MB
RR	32	✗	3.75 MB	25.76 kB
	128	✗	2.5 MB	21.31 kB
	1024	✗	1.56 MB	16.95 kB
NST	1	14.94 MB	**226.86 kB**	16.13 kB
	32	8.74 MB	**226.86 kB**	16.13 kB
	128	7.22 MB	**226.86 kB**	16.13 kB
	1024	6.42 MB	**226.86 kB**	16.13 kB
WRK	1	**2.86 MB**	570 kB	**4.86 kB**
	32	**2.64 MB**	570 kB	**4.86 kB**
	128	**2.0 MB**	570 kB	**4.86 kB**
	1024	**2.0 MB**	570 kB	**4.86 kB**
DUPLO	1	✗	12.94 MB	19.36 kB
	32	✗	2.60 MB	18.97 kB
	128	✗	1.96 MB	18.96 kB
	1024	✗	1.59 MB	18.96 kB

Table 4. Comparison of the data sent from constructor to evaluator AES-128 with $k = 128$ and $s = 40$. All numbers are per AES-128. Best results marked in bold.

Bandwidth Comparison As a final comparison we also consider the bandwidth requirements of the different protocols. In addition to the previous three protocols we here also include the recent work of Wang et al. (WRK) [KRW17]. To directly compare we report on the data required to transfer from constructor to receiver in Table 4 for different number of AES-128 executions. We stress that these numbers are all from the same AES-128 circuit [ST] and not from our optimized Frigate version. As already established for AES-128, DUPLO performs best by treating the entire circuit as a single component, hence we do not distinguish between Ind. Prep and Dep. Prep in the table. However we do stress that DUPLO only requires solderings from the input-wires to the output-wires of potentially large components, so for applicable settings we expect the Dep. Prep of DUPLO to be much lower than that of NST and WRK as they require solderings for each gate. It can be seen that for a single AES-128 component DUPLO cannot compare with the protocol of WRK in terms of overall bandwidth. This is natural as the replication factor is much lower for gate-level C&C in this case. However as the number of circuits grows we see that DUPLO's bandwidth requirement decreases significantly per AES-128 to a point where it is actually better than WRK by a factor 1.6x at 1024 executions. For the online phase it is clear that WRK's bandwidth is better than our protocol as we require decommitting the garbled input keys for the evaluator which induces some overhead. However we note that our implementation is not optimal in terms of online bandwidth in that we have chosen flexibility over minimizing rounds and bandwidth. For a dedicated application DUPLO's online bandwidth can be reduced by around 2x by combining the evaluate and decode phases and running batch-decommit of the evaluator input wires along with the output indicator bits.

References

[AHMR15] Arash Afshar, Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. How to efficiently evaluate RAM programs with malicious security. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 702–729. Springer, April 2015.

[AMPR14] Arash Afshar, Payman Mohassel, Benny Pinkas, and Ben Riva. Non-interactive secure computation based on cut-and-choose. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 387–404. Springer, May 2014.

[AO12] Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 681–698. Springer, December 2012.

[BCC88] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *J. Comput. Syst. Sci.*, 37(2):156–189, 1988.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, October 2012.

[BLN$^+$15] Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. High performance multi-party computation for binary circuits based on oblivious transfer. Cryptology ePrint Archive, Report 2015/472, 2015.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *STOC 1990*, pages 503–513. ACM Press, May 1990.

[BP09] Joan Boyar and Rene Peralta. New logic minimization techniques with applications to cryptology. Cryptology ePrint Archive, Report 2009/191, 2009.

[Bra13] Luís T. A. N. Brandão. Secure two-party computation with reusable bit-commitments, via a cut-and-choose with forge-and-lose technique - (extended abstract). In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 441–463. Springer, December 2013.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS 2001*, pages 136–145. IEEE Computer Society Press, October 2001.

[CDD$^+$16] Ignacio Cascudo, Ivan Damgård, Bernardo David, Nico Döttling, and Jesper Buus Nielsen. Rate-1, linear time and additively homomorphic UC commitments. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 179–207. Springer, 2016.

[CKKZ12] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the "free-XOR" technique. In Ronald Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 39–53. Springer, March 2012.

[DKL$^+$13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, September 2013.

[DLT14] Ivan Damgård, Rasmus Lauritsen, and Tomas Toft. An empirical study and some improvements of the MiniMac protocol for secure computation. In Michel Abdalla and Roberto De Prisco, editors, *SCN 2014*, volume 8642 of *LNCS*, pages 398–415. Springer, September 2014.

[DNNR16] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. Gate-scrambling revisited - or: The tinytable protocol for 2-party secure computation. Cryptology ePrint Archive, Report 2016/695, 2016.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, August 2012.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[DZ13] Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of Boolean circuits using preprocessing. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 621–641. Springer, March 2013.

[DZ16] Ivan Damgård and Rasmus Winther Zakarias. Fast oblivious AES A dedicated application of the MiniMac protocol. In *AFRICACRYPT 2016*, volume 9646 of *LNCS*, pages 245–264. Springer, 2016.

[FJN$^+$13] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. MiniLEGO: Efficient secure two-party computation from general assumptions. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 537–556. Springer, May 2013.

[FJNT15] Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. TinyLEGO: An interactive garbling scheme for maliciously secure two-party computation. Cryptology ePrint Archive, Report 2015/309, 2015.

[FJNT16] Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. On the complexity of additively homomorphic UC commitments. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part I*, volume 9562 of *LNCS*, pages 542–565. Springer, January 2016.

[FN13] Tore Kasper Frederiksen and Jesper Buus Nielsen. Fast and maliciously secure two-party computation using the GPU. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *ACNS 2013*, volume 7954 of *LNCS*, pages 339–356. Springer, June 2013.

[GLMY16] Adam Groce, Alex Ledger, Alex J. Malozemoff, and Arkady Yerukhimovich. CompGC: Efficient offline/online semi-honest two-party computation. Cryptology ePrint Archive, Report 2016/458, 2016.

[GMS08] Vipul Goyal, Payman Mohassel, and Adam Smith. Efficient two party and multi party computation against covert adversaries. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 289–306. Springer, April 2008.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *STOC 1987*, pages 218–229. ACM Press, May 1987.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security 2011*. USENIX Association, 2011.

[HFKV12] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ANSI C. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 772–783. ACM Press, October 2012.

[HKE13] Yan Huang, Jonathan Katz, and David Evans. Efficient secure two-party computation using symmetric cut-and-choose. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 18–35. Springer, August 2013.

[HKK$^+$14] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. Amortizing garbled circuits. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 458–475. Springer, August 2014.

[HMsG13] Nathaniel Husted, Steven Myers, abhi shelat, and Paul Grubbs. GPU and CPU parallelization of honest-but-curious secure two-party computation. In Charles N. Payne Jr., editor, *ACSAC 2013*, pages 169–178. ACM, 2013.

[KM15] Vladimir Kolesnikov and Alex J. Malozemoff. Public verifiability in the covert model (almost) for free. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part II*, volume 9453 of *LNCS*, pages 210–235. Springer, November / December 2015.

[KMRR15] Vladimir Kolesnikov, Payman Mohassel, Ben Riva, and Mike Rosulek. Richer efficiency/security trade-offs in 2PC. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part I*, volume 9014 of *LNCS*, pages 229–259. Springer, March 2015.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, August 2015.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS 2016*, pages 830–842. ACM Press, 2016.

[KRW17] Jonathan Katz, Samuel Ranellucci, and Xiao Wang. Authenticated garbling and communication-efficient, constant-round, secure two-party computation. Cryptology ePrint Archive, Report 2017/030, 2017.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, July 2008.

[KsS12] Benjamin Kreuter, abhi shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security 2012*. USENIX Association, 2012.

[KSS13] Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 549–560. ACM Press, November 2013.

[Lin13] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 1–17. Springer, August 2013.

[LOS14] Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multi-party computation for binary circuits. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 495–512. Springer, August 2014.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 52–78. Springer, May 2007.

[LP11] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 329–346. Springer, March 2011.

[LPS08] Yehuda Lindell, Benny Pinkas, and Nigel P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *SCN 2008*, volume 5229 of *LNCS*, pages 2–20. Springer, September 2008.

[LPSY15] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 319–338. Springer, August 2015.

[LR14] Yehuda Lindell and Ben Riva. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 476–494. Springer, August 2014.

[LR15] Yehuda Lindell and Ben Riva. Blazing fast 2PC in the offline/online setting with security for malicious adversaries. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 2015*, pages 579–590. ACM Press, October 2015.

[LSS16] Yehuda Lindell, Nigel P. Smart, and Eduardo Soria-Vazquez. More efficient constant-round multi-party computation from BMR and SHE. In Martin Hirt and Adam D. Smith, editors, *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part I*, volume 9985 of *Lecture Notes in Computer Science*, pages 554–581, 2016.

[LWN+15] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376, May 2015.

[MF06] Payman Mohassel and Matthew Franklin. Efficiency tradeoffs for malicious two-party computation. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 458–473. Springer, April 2006.

[MGBF14] Benjamin Mood, Debayan Gupta, Kevin R. B. Butler, and Joan Feigenbaum. Reuse it or lose it: More efficient secure computation through reuse of encrypted values. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 582–596. ACM Press, November 2014.

[MGC+16] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 112–127, March 2016.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay—a secure two-party computation system. In *USENIX Security 2004*. USENIX Association, 2004.

[MR13] Payman Mohassel and Ben Riva. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 36–53. Springer, August 2013.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, August 2012.

[NO09] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 368–386. Springer, March 2009.

[NR16] Jesper Buus Nielsen and Samuel Ranellucci. Reactive garbling: Foundation, instantiation, application. In *ASIACRYPT 2016, Part II*, LNCS, pages 1022–1052. Springer, December 2016.

[NST17] Jesper Buus Nielsen, Thomas Schneider, and Roberto Trifiletti. Constant round maliciously secure 2PC with function-independent preprocessing using LEGO. In *24. Annual Network and Distributed System Security Symposium (NDSS'17)*. The Internet Society, February 26-March 1, 2017. To appear.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, December 2009.

[Rab77] Michael O. Rabin. Digitalized signatures. Foundations of secure computation. In *Richard AD et al. (eds): Papers presented at a 3 day workshop held at Georgia Institute of Technology, Atlanta*, pages 155–166. Academic, New York, 1977.

[Rin] Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. `https://github.com/osu-crypto/libOTe`.

[RR16] Peter Rindal and Mike Rosulek. Faster malicious 2-party secure computation with online/offline dual execution. In *USENIX Security 2016*. USENIX Association, 2016.

[SHS+15] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*, pages 411–428. IEEE Computer Society Press, May 2015.

[sS11] abhi shelat and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 386–405. Springer, May 2011.

[sS13] abhi shelat and Chih-Hao Shen. Fast two-party secure computation with minimal assumptions. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 523–534. ACM Press, November 2013.

[ST] Nigel Smart and Stefan Tillich. Circuits of Basic Functions Suitable For MPC and FHE.

[TJ11] Henk C. A. Tilborg and Sushil Jajodia. *Encyclopedia of Cryptography and Security*. Springer Publishing Company, Incorporated, 2nd edition, 2011.

[WGMK16] Xiao Shaun Wang, S. Dov Gordon, Allen McIntosh, and Jonathan Katz. Secure computation of MIPS machine code. In *ESORICS 2016, Part II*, LNCS, pages 99–117. Springer, September 2016.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. Faster two-party computation secure against malicious adversaries in the single-execution setting. Cryptology ePrint Archive, Report 2016/762, 2016.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS 1986*, pages 162–167. IEEE Computer Society Press, October 1986.

[ZE15] Samee Zahur and David Evans. Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015.

[ZH17] Ruiyu Zhu and Yan Huang. Faster lego-based secure computation without homomorphic commitments. Cryptology ePrint Archive, Report 2017/226, 2017.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, April 2015.

A Protocol Details

We describe and analyse the protocol in the UC framework. We will here give an abstract description that lends itself to a security analysis. In Section 6.1 we describe some of the optimisations that were done in the implementation and why they do not affect the security analysis. We describe the protocol for two parties, the garbler G and the evaluator E. We will describe the protocol in the hybrid model with ideal functionalities $\mathcal{F}_{\mathsf{HCOM}}$ and $\mathcal{F}_{\mathsf{OT}}$ for xor-homomorphic commitment and one-out-of-two oblivious transfer. The precise description of the ideal functionalities are standard by now and can be found in [FJNT16] and [KOS15]. Here we will denote the use of the functionalities by some pseudo-code conventions. When using $\mathcal{F}_{\mathsf{HCOM}}$ it is G that is the committer and E that is the receiver. When G executes $\textsc{Commit}(\mathsf{cid}, x)$ for $\mathsf{cid} \in \{0,1\}^*$ and $x \in \{0,1\}^\kappa$, then $\mathcal{F}_{\mathsf{HCOM}}$ stores $(\mathsf{cid}, x_{\mathsf{cid}})$ (where $x_{\mathsf{cid}} = x$) and outputs cid to E. When G executes $\textsc{Open}(\mathsf{cid}_1, \ldots, \mathsf{cid}_c)$, where each cid_i was output to E at some point, then $\mathcal{F}_{\mathsf{HCOM}}$ outputs $(\mathsf{cid}_1, \ldots, \mathsf{cid}_c, \oplus_{i=1}^c x_{\mathsf{cid}_i})$ to E. When G executes an open command, then the commitment identifies $(\mathsf{cid}_1, \ldots, \mathsf{cid}_c)$ are always already known by E. If $\mathcal{F}_{\mathsf{HCOM}}$ outputs$(\mathsf{cid}'_1, \ldots, \mathsf{cid}'_c, \oplus_{i=1}^c x_{\mathsf{cid}_i})$ where some $\mathsf{cid}'_i \neq \mathsf{cid}_i$ then E always tacitly aborts the protocol. Similarly the cid used in the commit command is always known and E aborts if G uses a wrong one. When using $\mathcal{F}_{\mathsf{OT}}$ it is G that is the sender and E that is the receiver. We assume that we have access to a special OT which has a special internal state $\Delta \in \{0,1\}^\kappa$, which is chosen by G once and for all at the initialisation of the ideal functionality by executing $\textsc{OTinit}(\Delta)$. After that, when G executes $\textsc{OTsend}(\mathsf{id}, x_0)$ for $\mathsf{id} \in \{0,1\}^*$ and $x_0 \in \{0,1\}^\kappa$ and E executes $\textsc{OTreceive}(\mathsf{id}, b)$ for $b \in \{0,1\}$, then $\mathcal{F}_{\mathsf{OT}}$ outputs (id, x_b) to E, where $x_1 = x_0 \oplus \Delta$. If the protocol specifies that G is to execute $\textsc{OTreceive}(\mathsf{id}, b)$ and it does not or uses a wrong id, then E will always detect this and will tacitly abort.

When we instruct a party to send a value, we tacitly assume the receiver stores it under the same name when it is received.

When we instruct a party to check a condition, we mean that the party will abort if the condition is false.

When a variable like K_{id} is created in our pseudo-code, it can be accessed by another routine at the same party using the same identifier. Sometimes we use the **store** and **retrieve** key-words to explicitly do this. To save on notation, it will sometimes be done more implicitly, when it cannot lead to ambiguity. In general, if an uninitialised variable like K_{id} is used in a protocol, then there is an implicit "**retrieve** K_{id}" in the line before.

We assume that we have a free-xor garbling scheme $(\mathsf{Gb}, \mathsf{Ev})$ which has correctness, obliviousness and authenticity. We recall these notions now. The key length is some κ. The input to Gb is a poly-sized circuit C computing a function $C : \{0,1\}^n \to \{0,1\}^m$ along with $(K_1^0, \ldots, K_n^0, \Delta) \in (\{0,1\}^\kappa)^{n+1}$, where $\mathrm{lsb}(\Delta) = 1$. The output is $(L_1^0, \ldots, L_m^0) \in (\{0,1\}^\kappa)^m$ and a garbled circuit F. Here F is the garbled version of C. Define $K_i^1 = K_i^0 \oplus \Delta$. For $x \in \{0,1\}^n$ define $K^x = (K_1^{x_1}, \ldots, K_n^{x_n})$. This is the garbled input, i.e., the garbled version of x. Define $L_i^1 = L_i^0 \oplus \Delta$. For $y \in \{0,1\}^m$ define $L^y = (L_1^{y_1}, \ldots, L_m^{y_m})$. This is the garbled output. The input to Ev is a garbled circuit F and a garbled input $(K_1, \ldots, K_n) \in (\{0,1\}^\kappa)^n$. The output is $\perp$ or a garbled output $(L_1, \ldots, L_m) \in (\{0,1\}^\kappa)^m$. The scheme being free-xor means the inputs and outputs are of the above form. Correctness says that if you do garbled evaluation, you get the correct output. Obliviousness says that if you are given F but not given $(K_1^0, \ldots, K_n^0, \Delta)$, then the garbled input leaks no information on the plaintext input (or output). Authenticity says that if you are given only a garbled circuit for C and a garbled input for x, then you cannot compute the garbled output for any other value than the correct value $C(x)$. These notions have been formalized in [BHR12]. Here we recall them in the detail we need here and specialized to free-xor garbling schemes.

correctness $\forall x \in \{0,1\}^n$ and $\forall (K_1^0, \ldots, K_n^0, \Delta) \in (\{0,1\}^\kappa)^{n+1}$ with $\mathrm{lsb}(\Delta) = 1$ it holds for $(L_1^0, \ldots, L_m^0, F) = \mathsf{Gb}(K_1^0, \ldots, K_n^0, \Delta)$ that $\mathsf{Ev}(F, K^x) = L^{C(x)}$.

obliviousness For uniformly random $(K_1^0, \ldots, K_n^0, \Delta) \in (\{0,1\}^\kappa)^{n+1}$ with $\mathrm{lsb}(\Delta) = 1$ and $(\cdot, F) \leftarrow \mathsf{Gb}(K_1^0, \ldots, K_n^0, \Delta)$ and any $x_0, x_1 \in \{0,1\}^n$ it holds that (F, K^{x_0}) and (F, K^{x_1}) are computationally indistinguishable.

authenticity Let $\mathcal{A}$ be a probabilistic poly-time interactive Turing machine. Run $\mathcal{A}$ to get a circuit $C : \{0,1\}^n \to \{0,1\}^m$ and an input $x \in \{0,1\}^n$. Sample uniformly random $(K_1^0, \ldots, K_n^0, \Delta) \in (\{0,1\}^\kappa)^{n+1}$ with $\mathrm{lsb}(\Delta) = 1$ and $((L_1^0, \ldots, L_m^0), F) \leftarrow \mathsf{Gb}(K_1^0, \ldots, K_n^0, \Delta)$ and input (F, K^x) to $\mathcal{A}$. Let $y = C(x)$. Run $\mathcal{A}$ to get $(L_1, \ldots, L_n) \in (\{0,1\}^\kappa)^m$ and $y' \in \{0,1\}^m$. If $y' \neq y$ and $L = L^{y'}$, then $\mathcal{A}$ wins. Otherwise $\mathcal{A}$ loses. We require that the probability that any PPT $\mathcal{A}$ wins is negligible.

We will in fact require extended versions of these notions as we use a reactive garbling scheme in the sense of [NR16]. In a reactive garbling scheme one can make several independent garblings and then later solder an output wire id with keys $(K_{\mathsf{id}}^0, K_{\mathsf{id}}^1)$ onto an input wire id' with keys $(K_{\mathsf{id}'}^0, K_{\mathsf{id}'}^1)$ in another circuit. This involves releasing some information to the evaluator which allows the evaluator later to compute $K_{\mathsf{id}'}^b$ from K_{id}^b for either $b = 0$ or $b = 1$. The notion of reactive garbling scheme is given in [NR16]. We will use the reactive garbling scheme from [AHMR15]. We will later describe how to solder in [AHMR15] and we then recall the notion of reactive garbling scheme from [NR16] to the detail that we need in our proofs.

We finally assume that we have access to a programable random oracle $H : \{0,1\}^\kappa \to \{0,1\}^\kappa$. Note that this in particular implies that H is collision resistant.

We assume that we are to securely compute one circuit $\mathcal{C}$ which consist of sub-circuits C and solderings between input wires and output wires of these sub-circuits. We call the position in $\mathcal{C}$ in which a sub-circuit C is sitting a *slot* and each slot is identified by some identifier id. There is a public mapping from identifiers id to the corresponding sub-circuit C. If $C : \{0,1\}^n \to \{0,1\}^m$, then the inputs wires and output wires of the slot are identified by $\mathsf{id.in}.1, \ldots, \mathsf{id.in}.n$ and $\mathsf{id.out}.1, \ldots, \mathsf{id.out}.m$. Sub-circuits sitting at a slot are called functional sub-circuits. There are also some special sub-circuits:

- E in-gates, with $n = 0$ and $m = 1$. These are for letting E input a bit. The output wire is identified by $\mathsf{id.out.1}$.
- G in-gates, also with $n = 0$ and $m = 1$. These are for letting G input a bit. The output wire is identified by $\mathsf{id.out.1}$.
- E out-gates, with $n = 1$ and $m = 0$. These are the output gates of E. The input wire is identified by $\mathsf{id.in.1}$.
- G out-gates, with $n = 1$ and $m = 0$. These are the output gates of G. The input wire is identified by $\mathsf{id.in.1}$.

Besides a set of named sub-circuits, the circuit $\mathcal{C}$ also contains a set S of solderings $(\mathsf{id}_1, \mathsf{id}_2)$, where id_1 is the name of an output wire of a sub-circuit and id_2 is the name of an input wire of a sub-circuit. We require that all input wires of all sub-circuits are soldered to exactly one output wire of a sub-circuit and that there are no loops. This ensures we can plaintext evaluate the circuit as follows. For each in-gate id assign a bit x_{id} and say that $\mathsf{id.out.1}$ was evaluated. Now iteratively: 1) for each soldering $(\mathsf{id}_1, \mathsf{id}_2)$ where id_1 was evaluated, let $x_{\mathsf{id}_2} = x_{\mathsf{id}_1}$ and say id_2 was evaluated, and 2) for each sub-circuit where all input wires were evaluated, run C on the corresponding bits, assign the result to the output wires and say they are evaluated. This way all out-gates will be assigned a unique bit. The goal of our protocol is to let both parties learn their own output bits without learning any other information. We assume some given *evaluation order* of the sub-circuits that allows to plaintext evaluate in that order.

We assume that we have two functions $L^1, \alpha^1 : \mathbb{N} \to \mathbb{N}$ for setting the parameters of the cut-and-choose. Consider the following game parametrised by $n \in \mathbb{N}$. First the adversary picks $L = L^1(n)$ balls. Let $\alpha = \alpha^1(n)$. Some of them are green and some are red. The adversary picks the colours. Then we sample uniformly at random $L - \alpha n$ of the balls. If any of the sampled balls are red, the adversary immediately loses the game. Then we uniformly at random throw the remaining αn balls into n buckets of size α. The adversary wins if there is a bucket with only red balls. We assume that L^1 and α^1 have been fixed such that the probability that any adversary wins the game is 2^{-s}, where s is the security parameter. Note that the functions depend on s, but we ignore this dependence in the notation. We assume that we have two other functions $L^2, \alpha^2 : \mathbb{N} \to \mathbb{N}$. We consider a game similar to the above, but where the adversary wins if all sampled balls are green and there is a bucket with a majority of red balls. We assume that L^2 and α^2 have been fixed such that the probability that any adversary wins the game is 2^{-s}.

Overview of Notation

- id: generic identifier, just a bit-string naming an object.
- Δ_{id}: the *difference* with identifier id. Defined to be the value in the commitment with identifier $\mathsf{id.dif}$. It should hold that $\mathrm{lsb}(\Delta_{\mathsf{id}}) = 1$.
- σ_{id}: the *indicator* bit with identifier id. Defined to be the value in the commitment with identifier $\mathsf{id.ind}$.
- $K_{\mathsf{id}}^{\sigma_{\mathsf{id}}}$: *base-key* with identifier id. Defined to be the value in the commitment with identifier $\mathsf{id.base}$. It should hold that $\mathrm{lsb}(K_{\mathsf{id}}^{\sigma_{\mathsf{id}}}) = 0$.
- $K_{\mathsf{id}}^{1-\sigma_{\mathsf{id}}}$: *esab-key* with identifier id. Defined to be $K_{\mathsf{id}}^{\sigma_{\mathsf{id}}} \oplus \Delta_{\mathsf{id}}$.
- K_{id}^0: 0-*key* with identifier id.
- K_{id}^1: 1-*key* with identifier id.
- K_{id}: key held by E. It should hold that $K_{\mathsf{id}} \in \{K_{\mathsf{id}}^0, K_{\mathsf{id}}^1\}$.
- $L^1(n)$: total number of objects to create when one component should be good per bucket and n buckets are needed.

– $\alpha^1(n)$: bucket size when one component should be good per bucket and n buckets are needed.
– $L^2(n)$: as above, but majority in each bucket is good.
– $\alpha^2(n)$: as above, but majority in each bucket is good.
– $\mathrm{Par}(\mathsf{id})$: mapping from input wire id to the unique parent output wire. This is well-defined given the soldering set S.
– rco: A special wire index used in recovering inputs of a corrupted G.

Main Structure The main structure of the protocol will be as follows.

```
function Main($\mathcal{C}$)
    PreProcessKA()
    PreProcessInKA()
    PreProcessOTInit()
    PreProcessSub()
    AssembleSubs()
    AttachInKAs()
    for all sub-circuit id in evaluation order do
        if id is a G in-gate then InputG(id)
        else if id is an E in-gate then InputE(id)
        else if id is a G in-gate then OutputG(id)
        else if id is a E out-gate then OutputE(id)
        else EvSubs(id)
        end if
    end for
end function
```

We assume that E knows an input bit x_{id} for each E in-gate id before it is evaluated and that G knows an input bit x_{id} for each G in-gate id before it is evaluated. The inputs are allowed to depend adaptively on previous outputs. At the end of the protocol E knows an output bit y_{id} for each E out-gate id and G knows an output bit y_{id} for each G out-gate id.

During the pre-processing G will commit to key material for all wires. The keys K^0_{id} and K^1_{id} will be well defined from these committed values, even if G is corrupt. We then implement the input protocols and the evaluation protocols such that it is guaranteed that for each wire, E will learn $K_{\mathsf{id}} \in \{K^0_{\mathsf{id}}, K^1_{\mathsf{id}}\}$.

We then implement the input protocols such that it is guaranteed that for each G-input gate id the evaluator will learn some $K_{\mathsf{id}} \in \{K^0_{\mathsf{id}}, K^1_{\mathsf{id}}\}$. This holds even if G or E is corrupted. If G is honest, it is guaranteed that $K_{\mathsf{id}} = K^{x_{\mathsf{id}}}_{\mathsf{id}}$. If G is corrupted, then x_{id} is defined by $K_{\mathsf{id}} = K^{x_{\mathsf{id}}}_{\mathsf{id}}$. Furthermore, for each E-input gate E will learn $K_{\mathsf{id}} \in \{K^0_{\mathsf{id}}, K^1_{\mathsf{id}}\}$. This holds even if G or E is corrupted. If E is honest, it is guaranteed that $K_{\mathsf{id}} = K^{x_{\mathsf{id}}}_{\mathsf{id}}$. If E is corrupted, then x_{id} is defined by $K_{\mathsf{id}} = K^{x_{\mathsf{id}}}_{\mathsf{id}}$. This ensures that after the input protocols, an input bit x_{id} is defined for each input wire, called the plaintext value of the wire. This allows us to mentally do a plaintext evaluation of the circuits, which gives us a plaintext bit for each output wire and input wire of all components. We denote the bit defined for wire id by x_{id}. We call this the correct plaintext value of the wire. Note that this value might be known to neither E nor G. However, by security of the input protocols E will learn the correct key $K^{x_{\mathsf{id}}}_{\mathsf{id}}$ for in-gates. We then implement the evaluation protocol such that E iteratively will also learn the correct keys $K^{x_{\mathsf{id}}}_{\mathsf{id}}$ for all internal wires id. For the G-output wires, the evaluator E will just send $K^{x_{\mathsf{id}}}_{\mathsf{id}}$ to G who knows $(K^0_{\mathsf{id}}, K^1_{\mathsf{id}})$ and can decode to x_{id}. By authenticity E cannot force an incorrect output. For the E-output wires, the evaluator E will be given the indicator bit for the keys which will allow to compute exactly x_{id} from $K^{x_{\mathsf{id}}}_{\mathsf{id}}$. That the evaluator learns nothing else will follow from obliviousness of the garbling scheme.

We first describe some small sub-protocols and then later stitch them together to the sub-protocol used above. During the presentation of the sub-protocols we will argue correctness of the protocols, i.e., that they compute the correct keys $K_{\mathsf{id}}^{x_{\mathsf{id}}}$. In the following section we will then give a more detailed security analysis of the protocol.

Key Authenticators and Input The following protocol is used to assign key material to an identifier id.

```
function $\textsc{GenWire}(\mathsf{id}, K, \Delta)$                    ▷ Require: $\mathrm{lsb}(\Delta) = 1$
    G: $K_{\mathsf{id}}^0 \leftarrow K$                                  ▷ the 0-key
    G: $\Delta_{\mathsf{id}} \leftarrow \Delta$                              ▷ the difference
    G: $K_{\mathsf{id}}^1 \leftarrow K_{\mathsf{id}}^0 \oplus \Delta_{\mathsf{id}}$                          ▷ the 1-key
    G: $\sigma_{\mathsf{id}} \leftarrow \mathrm{lsb}(K_{\mathsf{id}}^0)$                       ▷ the indicator bit
    G: $\textsc{Commit}(\mathsf{id.dif}, \Delta_{\mathsf{id}})$
    G: $\textsc{Commit}(\mathsf{id.ind}, \sigma_{\mathsf{id}})$
    G: $\textsc{Commit}(\mathsf{id.base}, K_{\mathsf{id}}^{\sigma_{\mathsf{id}}})$
end function
```

The key K_{id}^b will be used to represent the plaintext value b. From $\mathrm{lsb}(\Delta) = 1$ it follows that $\mathrm{lsb}(K_{\mathsf{id}}^0) = \mathrm{lsb}(K_{\mathsf{id}}^1) \oplus 1$. We set $\sigma_{\mathsf{id}} = \mathrm{lsb}(K_{\mathsf{id}}^0)$. So, if $\mathrm{lsb}(K_{\mathsf{id}}^0) = 0$, then $\sigma_{\mathsf{id}} = 0$ and hence $\mathrm{lsb}(K_{\mathsf{id}}^{\sigma_{\mathsf{id}}}) = 0$. And if $\mathrm{lsb}(K_{\mathsf{id}}^0) = 1$, then $\sigma_{\mathsf{id}} = 1$ and hence $\mathrm{lsb}(K_{\mathsf{id}}^{\sigma_{\mathsf{id}}}) = \mathrm{lsb}(K_{\mathsf{id}}^1) = \mathrm{lsb}(K_{\mathsf{id}}^0) \oplus 1 = 0$. So in both cases

$$\mathrm{lsb}(K_{\mathsf{id}}^{\sigma_{\mathsf{id}}}) = 0 \ ,$$

as required. From setting $\sigma_{\mathsf{id}} = \mathrm{lsb}(K_{\mathsf{id}}^0)$ it also follows that $\sigma_{\mathsf{id}} \oplus 1 = \mathrm{lsb}(K_{\mathsf{id}}^1)$, which implies that

$$\mathrm{lsb}(K_{\mathsf{id}}^b) = b \oplus \sigma_{\mathsf{id}} \ ,$$

i.e., the last bit in the key is a one-time pad encryption of the plaintext value of the key with the indicator bit. In particular, given a key and the indicator bit, one can compute the plaintext value of the key as

$$b = \mathrm{lsb}(K_{\mathsf{id}}^b) \oplus \sigma_{\mathsf{id}} \ .$$

The next protocol allows to verify key material associated with an identifier.

```
function $\textsc{VerWire}(\mathsf{id})$
    G: $\textsc{Open}(\mathsf{id.dif})$; E: receive $\Delta_{\mathsf{id}}$
    E: check $\mathrm{lsb}(\Delta_{\mathsf{id}}) = 1$
    G: $\textsc{Open}(\mathsf{id.ind})$; E: receive $\sigma_{\mathsf{id}}$
    G: $\textsc{Open}(\mathsf{id.base})$; E: receive $K_{\mathsf{id}}^{\sigma_{\mathsf{id}}}$
    E: check $\mathrm{lsb}(K_{\mathsf{id}}^{\sigma_{\mathsf{id}}}) = 0$
    E: $K_{\mathsf{id}}^{1-\sigma_{\mathsf{id}}} \leftarrow K_{\mathsf{id}}^{\sigma_{\mathsf{id}}} \oplus \Delta_{\mathsf{id}}$
    E: store $K_{\mathsf{id}}^0, K_{\mathsf{id}}^1, \Delta_{\mathsf{id}}, \sigma_{\mathsf{id}}$
end function
```

We say that the key material associated with an identifier is *correct* if it would pass the verification protocol given that G opens all commitments. This just means that when K_{id}^0, Δ_{id} and σ_{id} are defined to be the values in the respective commitments and K_{id}^1 is defined to be $K_{\mathsf{id}}^0 \oplus \Delta_{\mathsf{id}}$, then

$$\mathrm{lsb}(\Delta_{\mathsf{id}}) = 1$$

and

$$\mathrm{lsb}(K_{\mathsf{id}}^{\sigma_{\mathsf{id}}}) = 0 \ .$$

Note the key material generated with $\textsc{GenWire}$ is correct. For brevity we will just say that id is *correct* when the key material associated with id is correct.

The following protocol reveals information that is used to solder two independent key materials. It will allow to translate a key for id_1 to a key for id_2.

```
function GenSold($\mathsf{id}_1, \mathsf{id}_2$)
    G: Open($\mathsf{id}_1.\mathsf{ind}, \mathsf{id}_2.\mathsf{ind}$); E: receive $\sigma_{\mathsf{id}_1,\mathsf{id}_2}$
    if $\sigma_{\mathsf{id}_1,\mathsf{id}_2} = 0$ then
        G: Open($\mathsf{id}_1.\mathsf{base}, \mathsf{id}_2.\mathsf{base}$)
    else
        G: Open($\mathsf{id}_1.\mathsf{base}, \mathsf{id}_2.\mathsf{base}, \mathsf{id}_2.\mathsf{dif}$)
    end if
    E: receive $K_{\mathsf{id}_1,\mathsf{id}_2}$
    G: Open($\mathsf{id}_1.\mathsf{dif}, \mathsf{id}_2.\mathsf{dif}$)
    E: receive $\Delta_{\mathsf{id}_1,\mathsf{id}_2}$
    check $\mathrm{lsb}(\Delta_{\mathsf{id}_1,\mathsf{id}_2}) = 0$
    check $\mathrm{lsb}(K_{\mathsf{id}_1,\mathsf{id}_2}) = \sigma_{\mathsf{id}_1,\mathsf{id}_2}$
end function
```

The last two checks ensure that if one of the key materials are correct, then the other one is correct too. To see this, assume that id_1 is correct. This just means that

$$\mathrm{lsb}(\Delta_{\mathsf{id}_1}) = 1$$

$$\mathrm{lsb}(K_{\mathsf{id}_1}^{\sigma_{\mathsf{id}_1}}) = 0 \ .$$

From $\mathrm{lsb}(\Delta_{\mathsf{id}_1}) = 1$, $\mathrm{lsb}(\Delta_{\mathsf{id}_1,\mathsf{id}_2}) = 0$ and $\Delta_{\mathsf{id}_1,\mathsf{id}_2} = \Delta_{\mathsf{id}_1} \oplus \Delta_{\mathsf{id}_2}$ it follows that

$$\mathrm{lsb}(\Delta_{\mathsf{id}_2}) = 1 \ .$$

We have by construction that

$$K_{\mathsf{id}_1,\mathsf{id}_2} = K_{\mathsf{id}_1}^{\sigma_{\mathsf{id}_1}} \oplus K_{\mathsf{id}_2}^{\sigma_{\mathsf{id}_2}} \oplus \sigma_{\mathsf{id}_1,\mathsf{id}_2}\Delta_{\mathsf{id}_2} \ .$$

Since we already established that $\mathrm{lsb}(\Delta_{\mathsf{id}_2}) = 1$ and we assumed that $\mathrm{lsb}(K_{\mathsf{id}_1}^{\sigma_{\mathsf{id}_1}}) = 0$, we have that

$$\mathrm{lsb}(K_{\mathsf{id}_1,\mathsf{id}_2}) = 0 \oplus \mathrm{lsb}(K_{\mathsf{id}_2}^{\sigma_{\mathsf{id}_2}}) \oplus \sigma_{\mathsf{id}_1,\mathsf{id}_2} \ .$$

Since E checks that $\mathrm{lsb}(K_{\mathsf{id}_1,\mathsf{id}_2}) = \sigma_{\mathsf{id}_1,\mathsf{id}_2}$, it follows that

$$\mathrm{lsb}(K_{\mathsf{id}_2}^{\sigma_{\mathsf{id}_2}}) = 0 \ .$$

Hence id_2 is correct. Showing that id_1 is correct if id_2 is correct follows a symmetric argument.

Note then that if both key materials are correct, then

$$\begin{aligned} K_{\mathsf{id}_1,\mathsf{id}_2} &= K_{\mathsf{id}_1}^{\sigma_{\mathsf{id}_1}} \oplus K_{\mathsf{id}_2}^{\sigma_{\mathsf{id}_2}} \oplus \sigma_{\mathsf{id}_1,\mathsf{id}_2}\Delta_{\mathsf{id}_2} \\ &= K_{\mathsf{id}_1}^0 \oplus \sigma_{\mathsf{id}_1}\Delta_{\mathsf{id}_1} \oplus K_{\mathsf{id}_2}^0 \oplus \sigma_{\mathsf{id}_2}\Delta_{\mathsf{id}_2} \oplus \sigma_{\mathsf{id}_1,\mathsf{id}_2}\Delta_{\mathsf{id}_2} \\ &= K_{\mathsf{id}_1}^0 \oplus \sigma_{\mathsf{id}_1}\Delta_{\mathsf{id}_1} \oplus K_{\mathsf{id}_2}^0 \oplus \sigma_{\mathsf{id}_1}\Delta_{\mathsf{id}_2} \\ &= K_{\mathsf{id}_1}^0 \oplus K_{\mathsf{id}_2}^0 \oplus \sigma_{\mathsf{id}_1}(\Delta_{\mathsf{id}_1} \oplus \Delta_{\mathsf{id}_2}) \\ &= K_{\mathsf{id}_1}^0 \oplus K_{\mathsf{id}_2}^0 \oplus \sigma_{\mathsf{id}_1}\Delta_{\mathsf{id}_1,\mathsf{id}_2} \ . \end{aligned}$$

We use this later.

The following protocol shows how to use the soldering information. It assumes that E already knows a key K_{id} for wire id. This key is either K_{id}^0 or K_{id}^1, but E might not know the plaintext value, so we use the generic name K_{id} for the key.

```
function EvSold($\mathsf{id}_1, \mathsf{id}_2, K$)
```

```
    E: return $K \oplus K_{\mathsf{id}_1,\mathsf{id}_2} \oplus \mathrm{lsb}(K)\Delta_{\mathsf{id}_1,\mathsf{id}_2}$
end function
function EvSold($\mathsf{id}_1, \mathsf{id}_2$)
    E: retrieve $K_{\mathsf{id}_1}$
    E: $K_{\mathsf{id}_2} \leftarrow$ EvSold($\mathsf{id}_1, \mathsf{id}_2, K_{\mathsf{id}_1}$)
end function
```
Note that if both key material is correct and $K_{\mathsf{id}_1} = K^b_{\mathsf{id}_1} = K^0_{\mathsf{id}_1} \oplus b\Delta_{\mathsf{id}_1}$, then

$$\begin{aligned}&K_{\mathsf{id}_1} \oplus K_{\mathsf{id}_1,\mathsf{id}_2} = \\ &(K^0_{\mathsf{id}_1} \oplus b\Delta_{\mathsf{id}_1}) \oplus (K^0_{\mathsf{id}_1} \oplus K^0_{\mathsf{id}_2} \oplus \sigma_{\mathsf{id}_1}\Delta_{\mathsf{id}_1,\mathsf{id}_2}) = \\ &K^0_{\mathsf{id}_2} \oplus b\Delta_{\mathsf{id}_1} \oplus \sigma_{\mathsf{id}_1}\Delta_{\mathsf{id}_1,\mathsf{id}_2}\ .\end{aligned}$$

It is easy to see that when the key materials are correct then $\mathrm{lsb}(K_{\mathsf{id}_1}) = b \oplus \sigma_{\mathsf{id}_1}$, from which it follows that

$$\sigma_{\mathsf{id}_1}\Delta_{\mathsf{id}_1,\mathsf{id}_2} \oplus \mathrm{lsb}(K_{\mathsf{id}_1})\Delta_{\mathsf{id}_1,\mathsf{id}_2} = b\Delta_{\mathsf{id}_1,\mathsf{id}_2}\ .$$

So,

$$\begin{aligned}K_{\mathsf{id}_2} &= K^0_{\mathsf{id}_2} \oplus b\Delta_{\mathsf{id}_1} \oplus b\Delta_{\mathsf{id}_1,\mathsf{id}_2} \\ &= K^0_{\mathsf{id}_2} \oplus b\Delta_{\mathsf{id}_2} \\ &= K^b_{\mathsf{id}_2}\ ,\end{aligned}$$

so $K^b_{\mathsf{id}_1}$ is mapped to $K^b_{\mathsf{id}_2}$, as intended.

Lemma 1 (correctness of soldering). *If either id_1 is correct or id_2 is correct and id_1 has been soldered onto id_2 without* E *aborting, then* both *of them are correct and*

$$\mathrm{EvSold}(\mathsf{id}_1, \mathsf{id}_2, K^b_{\mathsf{id}_1}) = K^b_{\mathsf{id}_2}$$

for $b = 0, 1$.

The next protocol is used to generate key authenticators.

```
function GenKeyAuth(id)
    G: $K^0_{\mathsf{id}} \leftarrow \{0,1\}^\kappa$
    G: $\Delta_{\mathsf{id}} \leftarrow \{0,1\}^{\kappa-1} \times \{1\}$
    G: GenWire($\mathsf{id}, K^0_{\mathsf{id}}, \Delta^0_{\mathsf{id}}$)
    G: $A_{\mathsf{id}} \leftarrow \{H(K^0_{\mathsf{id}}), H(K^1_{\mathsf{id}})\}$
    G: send $A_{\mathsf{id}}$
end function
```
Given a value $A = \{h_1, h_2\}$ and a key K we write $A(K) = \top$ if $H(K) \in A$. Otherwise we write $A(K) = \bot$. This protocol allows to verify a key authenticator.

```
function VerKeyAuth(id)
    VerWire(id)
    E: check $A_{\mathsf{id}} = \{H(K^0_{\mathsf{id}}), H(K^1_{\mathsf{id}})\}$
end function
```
We call a key authenticator with identifier id *correct* if it would pass the verification algorithm. Note that if it is correct, then $A_{\mathsf{id}} = \{H(K^0_{\mathsf{id}}), H(K^1_{\mathsf{id}})\}$ and E knows K^0_{id} and K^1_{id} as $K^{\sigma_{\mathsf{id}}}_{\mathsf{id}}$ and Δ_{id} were input to the commitment functionality. It therefore follows from the collision resistance of H that if $A(K) = \top$ for a key K computed in polynomial time, then $K \in \{K^0_{\mathsf{id}}, K^1_{\mathsf{id}}\}$ except with negligible probability. Furthermore, if in addition $A(K') = \top$ and $K' \neq K$, then $K \oplus K' = \Delta_{\mathsf{id}}$.

When we generate key authenticators for input gates, we will use the special form that $K^0_{\mathsf{id}} = H(\Delta_{\mathsf{id}})$.

```
function GenInKeyAuth(id)
    $\Delta_{\mathsf{id}} \leftarrow \{0,1\}^{\kappa-1} \times \{1\}$
    $K^0_{\mathsf{id}} \leftarrow H(\Delta_{\mathsf{id}})$
    GenWire(id, $K^0_{\mathsf{id}}, \Delta^0_{\mathsf{id}}$)
    $A_{\mathsf{id}} \leftarrow \{H(K^0_{\mathsf{id}}), H(K^1_{\mathsf{id}})\}$
    G sends $A_{\mathsf{id}}$
end function
```

To verify this special form we use this protocol.

```
function VerInKeyAuth(id)
    VerKeyAuth(id)
    E: check $K_{\mathsf{id}} = H(\Delta_{\mathsf{id}})$
end function
```

Note that if we are given a key K^b_{id} and Δ_{id} for an input gate, then we can compute b as follows. First compute $K^0_{\mathsf{id}} = H(\Delta_{\mathsf{id}})$. If $K^b_{\mathsf{id}} = K^0_{\mathsf{id}}$, then $b = 0$. Otherwise $b = 1$.

The following sub-protocol prepares a lot of key authenticators and uses cut-and-choose to verify that most are correct. The unopened key authenticators are put into buckets with a majority of correct ones in each bucket.

```
function PreProcessKA
    $\ell \leftarrow$ #output wires of all functional sub-circuits
    Let $L = L^2(\ell)$                                    ▷ # KAs generated
    Let $\alpha_{\mathsf{ka}} = \alpha^2(\ell)$                                  ▷ bucket size
    $\forall_{i=1}^{L}$ : GenKeyAuth(preka.$i$)
    E: Sample $V \subset [L]$ uniform of size $L - \alpha_{\mathsf{ka}}\ell$.
    E: send $V$
    $\forall_{i \in V}$ : VerKeyAuth(preka.$i$)
    for all functional sub-circuits id do
        for all $j = 1, \ldots, m_{\mathsf{id}}$ do
            pick $\alpha_{\mathsf{ka}}$ uniform, fresh KAs $i \notin V$
            rename them to ids id.ka.$1, \ldots,$ id.ka.$\alpha_{\mathsf{ka}}$.
            $\forall_{i=2}^{\alpha_{\mathsf{ka}}}$ : GenSold(id.ka.1, id.ka.$i$)
        end for
    end for
end function
```

We call id.ka.$1, \ldots,$ id.ka.α_{ka} a KA bucket, and we identify it by id.ka. We call id.ka KM correct if the key material associated with each id.ka.j is correct. We call it KA correct if it is KM correct and furthermore a majority of the KAs are correct, as defined above. By definition of L^2 and α^2 it follows that for each id.ka there will be a majority of id.ka.j for which the key material is correct, so there is in particular at least one for which this is true (except with negligible probability). By Lemma 1 this implies that the bucket is KM correct. By definition of L^2 and α^2 it then follows that it is additionally KA correct. We get that:

Lemma 2 (robustness of KA buckets). *If* E *is honest and* G *is honest or corrupted, then except with negligible probability each KA bucket id.ka is KA correct.*

We also use protocols PreProcessInKA, which work exactly as PreProcessKA, except that in PreProcessInKA we let ℓ be the number of input wires in $\mathcal{C}$, we pre-process the key authenticators using GenInKeyAuth instead, we verify using VerInKeyAuth, and we associated the unopened key authenticators to the identifiers of the input wires instead. For an input wire id the identifiers of the key authenticator will be id.ka.$1, \ldots,$ id.ka.α_{inka}.

The following protocol evaluates a key authenticator. It selects from a set of keys, namely the keys that are accepted by a majority of the key authenticators. If G is corrupted, then E might end up in the situation where it learns both a 0-key and a 1-key for a wire, if both of these are in the input key set. In that case we use a special recovery procedure RECOVER. This procedure will recover the plaintext input of E and use it to do a plaintext evaluation of the circuit instead of garbled evaluation. How this is done is described later.

```
function EvKAs(id, $\mathcal{K}_{\mathsf{id}}$)
    $\alpha \leftarrow \alpha_{\mathsf{ka}}$                              ▷ if generated using PreProcessKA
    $\alpha \leftarrow \alpha_{\mathsf{inka}}$                            ▷ if gen. using PreProcessInKA
    $\forall_{i=1}^{\alpha} : A_i \leftarrow A_{\mathsf{id.ka}.i}$                ▷ get key authenticators
    $\mathcal{L} \leftarrow \emptyset$
    for $K \in \mathcal{K}_{\mathsf{id}}$ do
        $K_1 = K$
        $\forall_{i=2}^{\alpha} : K_i = \text{EvSold}(\mathsf{id.ka}.1, \mathsf{id.ka}.i, K)$
        if $\#\{i \in \{1, \dots, \alpha\} \mid A_i(K_i) = \top\} > \alpha/2$ then
            $\mathcal{L} \leftarrow \mathcal{L} \cup \{K\}$
        end if
    end for
    if $\mathcal{L} = \{K\}$ then return $K$
    else if $\mathcal{L} = \{K_0, K_1\}$ then
        $\Delta \leftarrow K_0 \oplus K_1$
        Recover(id, $\Delta$)
    else abort
    end if
end function
```

Note that if $\mathsf{id.ka}$ is KA correct, then a majority of the key authenticators are correct and all the key materials are correct. Therefore, if a key is put in $\mathcal{L}$, then it is was accepted by at least one correct key authenticator. So, if a key K is in $\mathcal{L}$, then by the properties of correct key authenticators and Lemma 1, it follows that $K \in \{K^0_{\mathsf{id.ka}.1}, K^1_{\mathsf{id.ka}.1}\}$ except with negligible probability. So, assuming that $K^b_{\mathsf{id.ka}.1} \in \mathcal{K}_{\mathsf{id}}$ the outcome of the protocol is either to return $K^b_{\mathsf{id.ka}.1}$ or to call the recover protocol with the correct $\Delta = \Delta_{\mathsf{id.ka}.1}$, except with negligible probability.

Lemma 3 (robustness of EvKAs). *If E is honest and G is honest or corrupted, the following holds except with negligible probability. If for some $b \in \{0,1\}$ it holds that $K^b_{\mathsf{id.ka}.1} \in \mathcal{K}_{\mathsf{id}}$, then* EvKAs *returns $K^b_{\mathsf{id.ka}.1}$ or calls* Recover*(id, Δ) with $\Delta = \Delta_{\mathsf{id.ka}.1}$. If for no $b \in \{0,1\}$ it holds that $K^b_{\mathsf{id.ka}.1} \in \mathcal{K}_{\mathsf{id}}$, then* EvKAs *aborts.*

The following protocol allows E to learn $K^{x_{\mathsf{id}}}_{\mathsf{id}}$ for a G in-gate, where G has input x_i, without E learning x_i.

```
function InputG(id)                                    ▷ id is an ID of a G in-gate
    G: retrieve the input bit $x_{\mathsf{id}}$ for id
    G: $K \leftarrow K^{x_{\mathsf{id}}}_{\mathsf{id.ka}.1}$
    G: send $K$
    E: $K_{\mathsf{id}} \leftarrow \text{EvKAs}(\mathsf{id}, \{K\})$
    E: store $K_{\mathsf{id}}$
end function
```

Definition 1. *If G is corrupted and E is honest, then after an execution of* InputG*(id), define x_{id} as follows. If $\mathsf{id.ka}$ is KA correct, then define x_{id} by $K^{x_{\mathsf{id}}}_{\mathsf{id.ka}.1} = K_{\mathsf{id}}$. Otherwise let $x_{\mathsf{id}} = \perp$.*

By Lemma 3, if G sends a key not from $\{K^0_{\mathsf{id.ka}.1}, K^1_{\mathsf{id.ka}.1}\}$, then the procedure aborts. Otherwise it outputs that key.

Lemma 4 (robustness of InputG). *If* G *is corrupted and* E *is honest, then after an execution of* $\textsc{InputG}(\mathsf{id})$ *that did not abort, it holds except with negligible probability that* $x_{\mathsf{id}} \in \{0,1\}$ *and it holds for the key* K_{id} *then stored by* E *that* $K_{\mathsf{id}} = K_{\mathsf{id}}^{x_{\mathsf{id}}}$*.*

It is more complicated to give input to E as G is not to learn which key was received. To prepare for this oblivious input delivery we set up the oblivious transfer functionality such that G is also committed to the Δ chosen for the OT.

```
function PREPROCESSOTINIT
    G: $\Delta_{\mathsf{ot}} \leftarrow \{0,1\}^\kappa$
    G: $\textsc{Commit}(\mathsf{ot}, \Delta_{\mathsf{ot}})$
    G: $\textsc{OTInit}(\Delta_{\mathsf{ot}})$
    for $i \in [s]$ do
        G: $R_i \leftarrow \{0,1\}^\kappa$, $\textsc{OTsend}(R_i, \mathsf{ot}_i)$
        G: $\textsc{Commit}(\mathsf{ot}_i, R_i)$
        E: $b_i \leftarrow \{0,1\}, R_{b_i} \leftarrow \textsc{OTreceive}(b_i, \mathsf{ot}_i)$
        E: send $(R_{b_i}, b_i)$
        G: receive $(\bar{R}_i, \bar{b}_i)$; check $\bar{R}_i = R_{\bar{b}_i}$
        if $\bar{b}_i = 0$ then
            G: $\textsc{Open}(\mathsf{ot}_i)$
        else
            G: $\textsc{Open}(\mathsf{ot}_i, \mathsf{ot})$
        end if
        E: receive $\tilde{R}_i$
        E: check $\tilde{R}_i = R_{b_i}$
    end for
end function
```

The $\textsc{PreProcessOTInit}$ procedure ensures that the commitment to the identifier ot is a commitment to the difference Δ_{ot} which G chose for the OT functionality. It was proven in [NST17] that the protocol is sound except with probability 2^{-s} and that it is straight-line zero-knowledge when the simulator controls the commitment functionality.

The following protocol allows E to learn $K_{\mathsf{id}}^{x_{\mathsf{id}}}$ for an E in-gate, where E has input x_i, without G learning x_i and without E learning anything else.

```
function INPUTE(id)                                   ▷ id is an ID of an E in-gate
    G: $R_{\mathsf{ot}_{\mathsf{id}}} \leftarrow \{0,1\}^\kappa$, $\textsc{OTsend}(R_{\mathsf{ot}_{\mathsf{id}}}, \mathsf{ot}_{\mathsf{id}})$
    G: $\textsc{Commit}(\mathsf{ot}_{\mathsf{id}}, R_{\mathsf{ot}_{\mathsf{id}}})$
    E: $b_{\mathsf{ot}_{\mathsf{id}}} \leftarrow \{0,1\}, R_{b_{\mathsf{ot}_{\mathsf{id}}}} \leftarrow \textsc{OTreceive}(b_{\mathsf{ot}_{\mathsf{id}}}, \mathsf{ot}_{\mathsf{id}})$
    E: retrieve the input bit $x_{\mathsf{id}}$ for id
    E: send $f_{\mathsf{id}} = x_{\mathsf{id}} \oplus b_{\mathsf{ot}_{\mathsf{id}}}$
    G: $e_{\mathsf{id}} = f_{\mathsf{id}} \oplus \sigma_{\mathsf{id}}$
    E: $\mathsf{id}' \leftarrow \mathsf{id.ka.1}$
    if $e_{\mathsf{id}} = 0$ then
        G: $\textsc{Open}(\mathsf{id}', \mathsf{ot}_{\mathsf{id}})$
    else
        G: $\textsc{Open}(\mathsf{id}', \mathsf{ot}_{\mathsf{id}}, \mathsf{ot})$
    end if
    E : receive $D = K_{\mathsf{id}'} \oplus R_{\mathsf{ot}_{\mathsf{id}}} \oplus e_{\mathsf{id}} \Delta_{\mathsf{ot}}$
    G: $\textsc{Open}(\mathsf{id}'.\mathsf{dif}, \mathsf{ot})$; E : receive $S_{\mathsf{id}} = \Delta_{\mathsf{id}'} \oplus \Delta_{\mathsf{ot}}$
    G: $\textsc{Open}(\mathsf{id.ind})$; E : receive $\sigma_{\mathsf{id}}$
    E: $K = D \oplus R_{b_{\mathsf{ot}_{\mathsf{id}}}} \oplus (x_{\mathsf{id}} \oplus \sigma_{\mathsf{id}}) S_{\mathsf{id}}$
    E: $K_{\mathsf{id}'} \leftarrow \textsc{EvKAs}(\mathsf{id}, \{K\})$
    E: **check** $\mathrm{lsb}(K_{\mathsf{id}'}) = x_{\mathsf{id}} \oplus \sigma_{\mathsf{id}}$
    E: **store** $K_{\mathsf{id}'}$
  **end function**
```

Then identify the output wire $\mathsf{id}.1$ with $\mathsf{id.ka}.1$, such that $K_{\mathsf{id}} = K_{\mathsf{id}'}$. If G in the above commits honestly to the value $R_{\mathsf{ot}_{\mathsf{id}}}$ he also inputs to the OT functionality then we indeed see that

$$\begin{aligned}
K_{\mathsf{id}} &= D \oplus R_{b_{\mathsf{ot}_{\mathsf{id}}}} \oplus (x_{\mathsf{id}} \oplus \sigma_{\mathsf{id}}) S_{\mathsf{id}} \\
&= (K_{\mathsf{id}'} \oplus R_{\mathsf{ot}_{\mathsf{id}}} \oplus e_{\mathsf{id}} \Delta_{\mathsf{ot}}) \oplus R_{b_{\mathsf{ot}_{\mathsf{id}}}} \oplus (x_{\mathsf{id}} \oplus \sigma_{\mathsf{id}})(\Delta_{\mathsf{id}'} \oplus \Delta_{\mathsf{ot}}) \\
&= (K_{\mathsf{id}'} \oplus b_{\mathsf{ot}_{\mathsf{id}}} \Delta_{\mathsf{ot}} \oplus e_{\mathsf{id}} \Delta_{\mathsf{ot}}) \oplus (x_{\mathsf{id}} \oplus \sigma_{\mathsf{id}})(\Delta_{\mathsf{id}'} \oplus \Delta_{\mathsf{ot}}) \\
&= K_{\mathsf{id}'} \oplus (x_{\mathsf{id}} \oplus \sigma_{\mathsf{id}}) \Delta_{\mathsf{ot}} \oplus (x_{\mathsf{id}} \oplus \sigma_{\mathsf{id}})(\Delta_{\mathsf{id}'} \oplus \Delta_{\mathsf{ot}}) \\
&= K_{\mathsf{id}'} \oplus (x_{\mathsf{id}} \oplus \sigma_{\mathsf{id}}) \Delta_{\mathsf{id}'} \\
&= K_{\mathsf{id}'}^{x_{\mathsf{id}}} = K_{\mathsf{id}}^{x_{\mathsf{id}}} \ .
\end{aligned}$$

If instead G cheats and commits to a value $\bar{R} \neq R_{\mathsf{ot}_{\mathsf{id}}}$ then

$$K = K_{\mathsf{id}'}^{x_{\mathsf{id}}} \oplus F \ ,$$

where

$$F = \bar{R} \oplus R_{\mathsf{ot}_{\mathsf{id}}} \ .$$

Note that when G is honest, then $F = 0$ and therefore $K = K_{\mathsf{id}'}^{x_{\mathsf{id}}}$. If G uses $F \notin \{0, \Delta_{\mathsf{id}'}\}$, then

$$K \notin \{K_{\mathsf{id}'}^0, K_{\mathsf{id}'}^1\} \ ,$$

so by Lemma 3 it holds that except with negligible probability the procedure aborts. Importantly, this happens independent of the value of x_{id}. If G uses $F = \Delta_{\mathsf{id}'}$, then

$$K = K_{\mathsf{id}'}^{x_{\mathsf{id}}} \oplus \Delta_{\mathsf{id}'} = K_{\mathsf{id}'}^{1 \oplus x_{\mathsf{id}}} \ .$$

so by Lemma 3 it holds that

$$K_{\mathsf{id}'} = K_{\mathsf{id}'}^{1 \oplus x_{\mathsf{id}}} \ .$$

As we have ensured that $\mathrm{lsb}(\Delta_{\mathsf{id}'}) = 1$ for all KA buckets except with negligible probability (by Lemma 2) it follows from the check $\mathrm{lsb}(K_{\mathsf{id}}) = x_{\mathsf{id}} \oplus \sigma_{\mathsf{id}}$ that when $K = K_{\mathsf{id}'}^{1 \oplus x_{\mathsf{id}}}$, then the procedure aborts. So, all in all, if a cheating E uses $F \neq 0$, then the procedure aborts except with negligible probability, independently of the value of x_{id}.

Lemma 5 (robustness of InputE). *The following holds except with negligible probability. If* G *is corrupted and* E *is honest, then an execution of* INPUTE(*id*) *will abort or not independently of the value of x_{id}. Furthermore, when it does not abort, then it holds for the key K_{id} stored by* E *that $K_{id} = K_{id}^{x_{id}}$.*

Functional Sub-Circuits The following protocol generates a garbled circuit of circuit C and generates the key material for all input wires and output wires. All these wires share the same difference.

```
  **function** GENSUB($\mathsf{id}, C$)                                   ▷ $C : \{0,1\}^n \to \{0,1\}^m$
    G: $(K_1, \ldots, K_n) \leftarrow (\{0,1\}^\kappa)^n$
    G: $\Delta_{\mathsf{id}} \leftarrow \{0,1\}^{\kappa-1} \times \{1\}$
    G: $(L_1, \ldots, L_m, F_{\mathsf{id}}) \leftarrow \mathsf{Gb}(K_1, \ldots, K_n, \Delta_{\mathsf{id}})$
    G: **send** $F_{\mathsf{id}}$
    $\forall_{i=1}^n$ : GENWIRE($\mathsf{id.in}.i, K_i, \Delta_{\mathsf{id}}$)
```

```
        $\forall_{i=1}^{m}$ : GenWire(id.out.$i$, $L_i$, $\Delta_{\mathsf{id}}$)
    end function
```

The following protocol allows to verify that the garbling was done correctly.

```
    function VerSub(id, $C$)                                    ▷ $C : \{0,1\}^n \to \{0,1\}^m$
        $\forall_{i=1}^{n}$ : VerWire(id.in.$i$)
        $\forall_{i=1}^{m}$ : VerWire(id.out.$i$)
        E: $\Delta_{\mathsf{id}} \leftarrow \Delta_{\mathsf{id.in.1}}$
        E: $\forall_{i=2}^{n}$ : check $\Delta_{\mathsf{id.in.}i} = \Delta_{\mathsf{id}}$
        E: $\forall_{i=1}^{m}$ : check $\Delta_{\mathsf{id.out.}i} = \Delta_{\mathsf{id}}$
        E: $\forall_{i=1}^{n}$ : $K_i \leftarrow K^0_{\mathsf{id.in.}i}$
        E: $\forall_{i=1}^{m}$ : $L_i \leftarrow K^0_{\mathsf{id.out.}i}$
        E: check $(L_1, \dots, L_m, F_{\mathsf{id}}) = \mathsf{Gb}(K_1, \dots, K_n, \Delta_{\mathsf{id}})$
    end function
```

We call a generated garbled sub-circuit with identifier id *correct* if it would pass the above verification protocol. Notice that if it is correct, then for all $x \in \{0,1\}^n$ and $y = C(x)$ it holds that $\mathsf{Ev}(F_{\mathsf{id}}, K^x_{\mathsf{id.in}}) = K^y_{\mathsf{id.out}}$, which shows that the following protocol works as intended.

```
    function EvSub(id)
        E: $\forall_{i=1}^{n}$ : $K_i \leftarrow$ retrieve $K_{\mathsf{id.in.}i}$
        E: $(L_1, \dots, L_m) \leftarrow \mathsf{Ev}(F_{\mathsf{id}}, K_1, \dots, K_n)$
        E: $\forall_{i=1}^{m}$ : store $K_{\mathsf{id.out.}i} \leftarrow L_i$
    end function
```

Lemma 6 (correctness of EvSub). *If $K_{id.in.i} = K^{x_i}_{id.in.i}$ for $i = 1, \dots, n$ and id is correct, then after an execution of* EvSub(*id*) *it holds that $K_{id.out.i} = K^{y_i}_{id.out.i}$ for $i = 1, \dots, m$, where $y = C(x)$.*

The following protocol will generate garblings of all the needed types of sub-circuits. It will generate more than needed. Some of these will be opened and checked. The rest will be assigned at random into buckets. Each slots id will be assigned some α_{id} unopened circuits.

```
    function PreProcessSub
        for all sub-circuit types $C$ do
            Let $\ell$ be the number of times $C$ is used.
            Let $L = L^1(\ell)$                                   ▷ #circuits generated
            Let $\alpha_{\mathsf{id}} = \alpha^1(\ell)$                                       ▷ bucket size
            $\forall_{i=1}^{L}$ : GenSub($C$.pre.$i$, $C$)
            E: Sample $V \subset [L]$ uniform of size $L - \alpha_{\mathsf{id}}\ell$.
            E: send $V$
            $\forall_{i \in V}$ : VerSub($C$.pre.$i$, $C$)
            for all slots id where $C$ occurs do
                pick $\alpha_{\mathsf{id}}$ uniform, fresh circuits $i \notin V$
                rename them to have ids id.1, ..., id.$\alpha_{\mathsf{id}}$.
                $\forall_{i=2}^{\alpha_{\mathsf{id}}}$ : GenSoldSub(id.1, id.$i$)
            end for
        end for
    end function
```

By *fresh* we mean that no circuit is assigned to more than one slot. We used this sub-protocol.

```
    function GenSoldSub(id$_1$, id$_2$)
        $\forall_{i=1}^{n}$ : GenSold(id$_1$.in.$i$, id$_2$.in.$i$)
        $\forall_{i=1}^{m}$ : GenSold(id$_2$.out.$i$, id$_1$.out.$i$)
        $\forall_{i=1}^{m}$ : GenSold(id.1.out.$i$, id.1.out.$i$.ka.1)
```

```
  end function
```

After all sub-circuits have been preprocessed the below procedure can be applied to stitch them all together to compute the final functionality $\mathcal{C}$.

```
  function AssembleSubs
      for all functional sub-circuits id do
          $\forall_{i=1}^{n}$ : $\mathsf{id}_{\mathsf{par}.i} \leftarrow \mathrm{Par}(\mathsf{id.1.in}.i)$
          $\forall_{i=1}^{n}$ : $\textsc{GenSold}(\mathsf{id}_{\mathsf{par}.i}, \mathsf{id.1.in}.i)$
      end for
  end function
```

If the linking of components are done adaptively, the above soldering is done only when needed.

The following procedure is used to evaluate a bucket of garbled sub-circuits that have been soldered together. Since each sub-circuit might give different output keys (if G is corrupted) the output for each output wire might not be a single key, but a set of keys. We therefore use the associated KAs to reduce this set to one key. If the reduced set contains two keys we will again call the recovery mechanism, which we are still to describe.

```
  function EvSubs(id)
                                                          ▷ Evaluate first circuit in bucket
      $\forall_{j=1}^{n}$ : $\mathsf{id}_{\mathsf{par}.j} \leftarrow \mathrm{Par}(\mathsf{id.1.in}.j)$
      $\forall_{j=1}^{n}$ : $\textsc{EvSold}(\mathsf{id}_{\mathsf{par}.j}, \mathsf{id.1.in}.j)$
      $\forall_{j=1}^{n}$ : retrieve $K_{\mathsf{id.1.in}.j}$
      $\textsc{EvSub}(\mathsf{id.1})$
      $\forall_{j=1}^{m}$ : retrieve $K_{\mathsf{id.1.out}.j}$
      $\forall_{j=1}^{m}$ : $\textsc{EvSold}(\mathsf{id.1.out}.j, \mathsf{id.1.out}.j.\mathsf{ka.1})$
      $\forall_{j=1}^{m}$ : $\mathcal{K}_j \leftarrow \{K_{\mathsf{id.1.out}.j.\mathsf{ka.1}}\}$                  ▷ key sets
                                                   ▷ Evaluate remaining circuits in bucket
      for $i = 2, \dots, \alpha_{\mathsf{id}}$ do E:
          $\forall_{j=1}^{n}$ : $\textsc{EvSold}(\mathsf{id.1.in}.j, \mathsf{id}.i.\mathsf{in}.j)$
          $\textsc{EvSub}(\mathsf{id.i})$
          $\forall_{j=1}^{m}$ : $\textsc{EvSold}(\mathsf{id}.i.\mathsf{out}.j, \mathsf{id.1.out}.j)$
          $\forall_{j=1}^{m}$ : $\textsc{EvSold}(\mathsf{id.1.out}.j, \mathsf{id.1.out}.j.\mathsf{ka.1})$
          $\forall_{j=1}^{m}$ : $\mathcal{K}_j \leftarrow \mathcal{K}_j \cup \{K_{\mathsf{id.1.out}.j.\mathsf{ka.1}}\}$
      end for
      $\forall_{j=1}^{m}$ : $K_{\mathsf{id.1.out}.j} \leftarrow \textsc{EvKAs}(\mathsf{ka}_j, \mathcal{K}_j)$
  end function
```

Lemma 7 (robustness of EvSubs). *If* G *is corrupt and* E *is honest and the sub-circuit for* id *is* C*, then the following holds except we negligible probability. If* $K_{\mathit{id}_{\mathit{par}.j}} = K^{x_j}_{\mathit{id}_{\mathit{par}.j}}$ *for* $j = 1, \dots, n$*, then after an execution of* $\textsc{EvSubs}(\mathit{id})$ *that did not call* $\textsc{Recover}$ *it holds that* $K_{\mathit{id.1.out}.j} = K^{y_j}_{\mathit{id.1.out}.j}$ *for* $j = 1, \dots, m$*, where* $y = C(x)$*.*

Proof. All the following statements hold except with negligible probability, assuming the premise of the above lemma. As we argued when we established Lemma 3, we can argue that all the key material used in EvSubs is correct. This follows from Lemma 1 as all the key material sits in buckets that are soldered and each bucket contains at least one correct key material by construction. It then follows from Lemma 1 that $K_{\mathsf{id.1.in}.j} = K^{x_i}_{\mathsf{id.1.in}.j}$. By definition of L^1 and α^1 we have that at least one circuit $\mathsf{id}.i$ is correct. By Lemma 1 and Lemma 3 it therefore follows that $K_{\mathsf{id.1.out}.j} \in \mathcal{K}_j$. By Lemma 3 it then follows that either Recover is called, or $K_{\mathsf{id.1.out}.j} = K^{y_j}_{\mathsf{id.1.out}.j}$. □

Output The following protocol allows E to get an output.

```
function OutputE(id)                                   ▷ id: ID of an E output gate
    E: retrieve soldering $(\mathsf{id}_1, \mathsf{id.out}.1)$ from $\mathcal{C}$.
    E: retrieve $K_{\mathsf{id}_1}$.
    G: Open$(\mathsf{id}_1.\mathsf{ind})$; E: receive $\sigma_{\mathsf{id}_1}$
    E: $y_{\mathsf{id}} \leftarrow \mathrm{lsb}(K_{\mathsf{id}_1}) \oplus \sigma_{\mathsf{id}_1}$.
end function
```

Lemma 8 (robustness of OutputE). *If G is corrupt and E is honest and $K_{id_1} = K^b_{id_1}$, then after an execution of* OutputE *that does not abort, it holds that $y_{id} = b$. Furthermore, whether or not the protocol aborts is independent of y_{id}.*

The first part of the lemma follows from $\mathrm{lsb}(K^b_{\mathsf{id}_1}) = b \oplus \sigma_{\mathsf{id}_1}$. The second part is obvious as the protocol aborts only if the commitment is not opened, which is the choice of G, and G does not know y_{id}.

The following protocol allows G to get an output.

```
function OutputG(id)                                   ▷ id: ID of an G output gate
    E: retrieve soldering $(\mathsf{id}_1, \mathsf{id.out}.1)$ from $\mathcal{C}$
    E: retrieve $K_{\mathsf{id}_1}$
    E: send $K_{\mathsf{id}_1}$
    G: receive $K_{\mathsf{id}_1}$
    G: check $K_{\mathsf{id}_1} \in \{K^0_{\mathsf{id}_1}, K^1_{\mathsf{id}_1}\}$
    G: $y_{\mathsf{id}} \leftarrow b$ where $K^b_{\mathsf{id}_1} = K_{\mathsf{id}_1}$.
end function
```

Correctness follows from authenticity of the garbling scheme. Even a corrupt E will know some key $K_{\mathsf{id}_1} \in \{K^0_{\mathsf{id}_1}, K^1_{\mathsf{id}_1}\}$ and this by correctness of the circuits and solderings is the right one. Sending another key $K' \in \{K^0_{\mathsf{id}_1}, K^1_{\mathsf{id}_1}\}$ would break authenticity. Security against E follows from obliviousness.

The following results is immediate by inductively applying the above lemmas.

Lemma 9 (robustness without recover). *If G is corrupt and E is honest and the protocol does not abort and does not call* Recover*, then the following holds except with negligible probability. For each input gate id the evaluator holds $K_{id} = K^{x_i}_{id}$. For E-in-gates x_i is the correct input of E. Furthermore, for each output gate id′, the evaluator E holds $K_{id'} = K^{y_i}_{id'}$ where y_i is the plaintext value obtained by evaluating $\mathcal{C}$ in plaintext on the input values x_{id}. Furthermore, the probability that the protocol aborts is independent of the inputs of E.*

Notice that it might not be the case that whether or not Recover is called is independent of the inputs of E. It is in fact easy for a corrupt G to construct garblings for which we go into recovery mode if and only if some internal wire has plaintext value 1. This is called a selective attack.

Recovery The description of the above procedures for evaluation and output assume that Recover has not been called. We now describe what happens when Recover is called. Recall that Recover is called in EvKAs when that procedure learns both the 0-key K^0_{id} and the 1-key K^1_{id} for some wire id. The issue is that the procedure then cannot know which key is the right one, as it does not know the plaintext value x_{id} nor does it know which key is the 0-key.

When the procedure is called, it is called as Recover(id, Δ), where $\mathsf{id.ka}.1, \ldots, \mathsf{id.ka}.\alpha$ are the identifier of key authenticators that have been produced using PreprocessKA or PreprocessInKA. Furthermore, Δ is the difference for the key authenticator id.ka.1

Let $\mathcal{E}$ be the set of identifiers id for which id is an in-gate. This implies that if $\mathsf{id} \in \mathcal{E}$, then $\mathsf{id.ka}.1, \ldots, \mathsf{id.ka}.\alpha_{\mathsf{inka}}$ are identifiers of input key authenticators. By construction a majority of them are correct except with negligible probability.

Let $\mathcal{I}$ be the set of identifiers id with which RECOVER could be called but which are not in $\mathcal{E}$, these are internal wires (non-input wires) with associated KA buckets. This implies that if $\mathsf{id} \in \mathcal{I}$, then $\mathsf{id.ka}.1, \ldots, \mathsf{id.ka}.\alpha_{\mathsf{inka}}$ are identifiers of key authenticators. By construction a majority of them are correct except with negligible probability.

As a simple motivating example assume that recover is called with an identifier from $\mathcal{E}$. The following procedure shows how this allows to recover the plaintext value x_{id}.

```
function RECOVERINPUTBIT($\mathsf{id}, \Delta$)                ▷ $\Delta = \Delta_{\mathsf{id.ka}.1}$
    $\mathsf{id}' \leftarrow \mathsf{id.ka}$
    $\Delta_1 \leftarrow \Delta$
    $\alpha \leftarrow \alpha_{\mathsf{inka}}$
    $\forall_{j=1}^{\alpha}$ retrieve $K_j \leftarrow K_{\mathsf{id}'.j}$;
    $\forall_{j=2}^{\alpha}$ retrieve $\Delta_{\mathsf{id}'.1,\mathsf{id}'.j}$; $\Delta_j \leftarrow \Delta_{\mathsf{id}'.1,\mathsf{id}'.j} \oplus \Delta_1$
    if $\#\{j \in \{1, \ldots, \alpha\} \mid H(\Delta_j) = K_j\} > \alpha/2$ then
        $x_{\mathsf{id}} \leftarrow 0$
    else $x_{\mathsf{id}} \leftarrow 1$
    end if
end function
```

Lemma 10. *If* G *is corrupt and* E *is honest, the following holds except with negligible probability. If $K_{id} = K_{id}^b$ and $\Delta = \Delta_{id.ka.1}$, then after an execution of* RECOVERINPUTBIT(id, Δ) *it holds that $x_{id} = b$.*

Proof. The following statements hold except with negligible probability. By premise there exists b such that $K_j = K_j^b$ for all j. By Lemma 2 it holds that $\Delta_j = \Delta_{\mathsf{id}'.j}$ for all j. This implies that for the majority of correct input key authenticators it holds that $H(\Delta_j) = K_j$ if and only if $K_j = K_j^0$. Hence, if $b = 0$, it will hold for a majority of j that $H(\Delta_j) = K_j$, and if $b = 1$ it will hold for a majority of j that $H(\Delta_j) \neq K_j$. □

Consider then the case where RECOVER is called with $\mathsf{id} \notin \mathcal{E}$. We want to ensure that if G is caught cheating, then E can recover all inputs of G and then evaluate $\mathcal{C}$ in plaintext. For this we only have to ensure that if RECOVER is called with $\mathsf{id} \in \mathcal{I}$, then it can call RECOVERINPUTBIT with all $\mathsf{id} \in \mathcal{E}$. To facilitate this, we are going to create on special wire rco and solder all $\mathsf{id} \in \mathcal{I}$ onto rco and solder rco onto all $\mathsf{id} \in \mathcal{E}$. For security reasons we do not do full solderings, we only release the difference between the Δ values. That way, if we learn the difference for any $\mathsf{id} \in \mathcal{I}$, we can learn the difference for all $\mathsf{id} \in \mathcal{E}$, and we are done.

The following procedure will be run together with all the other preprocessing protocols.

```
function PREPROCESSRECOVERY
    G: $\Delta_{\mathsf{rco}} \leftarrow \{0,1\}^{\kappa}$
    G: COMMIT(rco, $\Delta_{\mathsf{rco}}$)
    G: $\forall \mathsf{id} \in \mathcal{I} \cup \mathcal{E}$: OPEN(id.ka.1.dif, rco)
    E: $\forall \mathsf{id} \in \mathcal{I} \cup \mathcal{E}$: receive $\Delta_{\mathsf{id,rco}}$
    G: $\forall \mathsf{id} \in \mathcal{E}$: OPEN(rco, id.ka.1.dif)
    E: $\forall \mathsf{id} \in \mathcal{E}$: receive $\Delta_{\mathsf{rco,id}}$
end function
```

The following procedure uses the Δ-solderings to recover all the inputs.

```
function RECOVERINPUTBITS($\mathsf{id}, \Delta$)               ▷ $\Delta = \Delta_{\mathsf{id.ka}.1}$
    retrieve $\Delta_{\mathsf{id,eecov}}$
    $\Delta_{\mathsf{rco}} \leftarrow \Delta_{\mathsf{id,rco}} \oplus \Delta$
```

```
        for all $\mathsf{id}' \in \mathcal{E}$ do
            retrieve $\varDelta_{\mathsf{rco},\mathsf{id}'}$
            $\varDelta_{\mathsf{id}'} \leftarrow \varDelta_{\mathsf{rco},\mathsf{id}'} \oplus \varDelta_{\mathsf{rco}}$
            RecoverInputBit($\mathsf{id}', \varDelta_{\mathsf{id}'}$)
        end for
    end function
```

The following result is immediate.

Lemma 11. *If* G *is corrupt and* E *is honest, the following holds except with negligible probability. If $K_{id'} = K_{id'}^{b_{id'}}$ for all input gates id' and $\varDelta = \varDelta_{id.ka.1}$, then after an execution of* RecoverInputBits$(id, \varDelta)$ *it holds that $x_{id'} = b_{id'}$ for all input gates id'.*

Notice that RecoverInputBits not only computes x_{id} for all input gates. It can also compute the key K_{id}^0 and the difference $\varDelta_{\mathsf{id}}$. From the inputs x_{id} it can compute the correct plaintext value x_{id} for *all* wires id. From the keys K_{id}^0 and the differences $\varDelta_{\mathsf{id}}$ it can use Gb iteratively to compute also the correct key K_{id}^0 and the correct difference $\varDelta_{\mathsf{id}}$ for all subsequence wires id, as it has all the information it needs to compute the garblings the way G ought to have done it if it started from the computed inputs keys and differences. This will in particular allow E to compute for each G-output gate id the plaintext output x_{id} and the key $K_{\mathsf{id}} = K_{\mathsf{id}}^0 \oplus x_{\mathsf{id}}\varDelta_{\mathsf{id}}$. This is how the outputs will be computed in recovery mode.

```
    function Recover($\mathsf{id}, \varDelta$)                          ▷ $\varDelta = \varDelta_{\mathsf{id.ka}.1}$
        RecoverInputBits($\mathsf{id}, \varDelta$)
        go to recovery mode
    end function
    function OutputG($\mathsf{id}$)                                      ▷ In recovery mode
        E: retrieve soldering $(\mathsf{id}_1, \mathsf{id.out}.1)$ from $\mathcal{C}$
        E: retrieve $x_{\mathsf{id}_1}$
        E: retrieve $K_{\mathsf{id}_1}^0$
        E: retrieve $\varDelta_{\mathsf{id}_1}^0$
        E: send $K_{\mathsf{id}_1}^{x_{\mathsf{id}_1}}$
    end function
    function OutputE($\mathsf{id}$)                                      ▷ In recovery mode
        E: retrieve soldering $(\mathsf{id}_1, \mathsf{id.out}.1)$ from $\mathcal{C}$.
        E: retrieve $x_{\mathsf{id}_1}$.
        E: $y_{\mathsf{id}} \leftarrow x_{\mathsf{id}_1}$.
    end function
```

The following result follows from the above discussion.

Lemma 12 (robustness with recover). *If* G *is corrupt and* E *is honest and the protocol calls* Recover*, then the following holds except with negligible probability. For each input gate id the evaluator holds $K_{id} = K_{id}^{x_i}$. For* E*-in-gates x_i is the correct input of* E*. Furthermore, for each output gate id', the evaluator* E *holds $K_{id'} = K_{id'}^{y_i}$ where y_i is the plaintext value obtained by evaluating $\mathcal{C}$ in plaintext on the input values x_{id}. Furthermore, the probability that the protocol aborts is independent of the inputs of* E*.*

By combining Lemma 9 and Lemma 12 we get:

Theorem 2 (robustness). *If* G *is corrupt and* E *is honest and the protocol does not abort, then the following holds except with negligible probability. For each input gate id the evaluator holds $K_{id} = K_{id}^{x_i}$. For* E*-in-gates x_i is the correct input of* E*. Furthermore, for each output gate id', the evaluator* E *holds $K_{id'} = K_{id'}^{y_i}$ where y_i is the plaintext value obtained by evaluating $\mathcal{C}$ in plaintext on the input values x_{id}. Furthermore, the probability that the protocol aborts is independent of the inputs of* E*.*

Recall the issue with the selective attack that a corrupt G can ensure that Recover is called based on for instance the value of an internal wire. We now see that this is handled by making sure that in recovery mode, we return the exact same values to G as we would when we are not in recovery mode,

B Analyis

Our proof follows the proofs in [NST17] and [AHMR15] closely. Redoing the proofs in the full, glorious detail in the UC model would have us reiterate much of the proofs in these papers. We will instead sketch the overall structure of the formal proof and point to [NST17] and [AHMR15] for the details. We realize that this means that only the reader which is familiar with the UC model and and the mentioned papers may completely verify the proofs. However, since the UC model and our the proof techniques are standard by now we find this a reasonable level of proof detail.

B.1 Corrupt G

We first prove security for the case where G is corrupt and E is honest.

Without going into the details of the UC framework, let us just recall that the proof tasks as usual are as follows. When E is corrupted and G is honest, we should present a poly-time simulator $\mathcal{S}$. It plays the role of E in the protocol. But as opposed to E it is *not* being given the inputs x_{E} of E. Instead it has access to an oracle $\mathcal{O}_{x_{\mathsf{E}}}(\cdot)$ containing x_{E}. The simulator might once supply a possible set of inputs x_{G} of G to the oracle and learn the outputs $y_{\mathsf{G}} = \mathcal{O}_{x_{\mathsf{E}}}(x_{\mathsf{G}})$ that G would have if $\mathcal{C}$ was evaluated on the x_{E} in the oracle and the provided x_{G}.

The simulator $\mathcal{S}$ proceeds as follows. It runs as the honest E would do, but with two modifications. 1) It uses dummy inputs $x_{\mathsf{id}} = 0$ for all E-in-gates and 2) there is a modification in OutputG, which we describe below.

For all G-in-gates id it inspects the commitment functionality and learns K^0_{id} and $\varDelta_{\mathsf{id}}$ and computes K^1_{id} from these. From all G-in-gates it can then retrieve $K_{\mathsf{id}} = K^{x_{\mathsf{id}}}_{\mathsf{id}}$ which is well defined by Theorem 2. This defined the inputs x_{id} of G. Then it calls its oracle with those input bits of G and for all G-out-gates id it receives from the oracle $\mathcal{O}$ the output y_{id} obtained by running $\mathcal{C}$ in plaintext with those inputs x_{id} and the unknown input bits of E. If the protocol aborts, then the simulator aborts too. If the protocol reaches an execution of OutputG, then the simulator will send the key $K^{x_{\mathsf{id}_1}}_{\mathsf{id}_1}$ computed as in recovery mode. Note that it can do this as it knows all the keys and therefore can compute $K^0_{\mathsf{id}_1}$ and $K^1_{\mathsf{id}_1}$ exactly as in recovery mode, and it was given y_{id} from the oracle. It follows directly from Theorem 2 that the real protocol and the simulation aborts with the same probability and that when they do not abort, then the key returned to E from the simulator is the same that the honest E would have sent, except with negligible probability.

B.2 Corrupt E

We then prove security for the case where G is honest and E is corrupt.

Without going into the details of the UC framework, let us just recall that the proof tasks as usual are as follows. When G is corrupted and E is honest, we should present a poly-time simulator $\mathcal{S}$. It plays the role of G in the protocol. But as opposed to G it is *not* being given the input x_{G} of G. Instead it has access to an oracle $\mathcal{O}_{x_{\mathsf{G}}}(\cdot)$ containing x_{G}. The simulator might once supply a possible input x_{E} of E to the oracle and learn the outputs $y_{\mathsf{E}} = \mathcal{O}_{x_{\mathsf{G}}}(x_{\mathsf{E}})$ that E would have if $\mathcal{C}$ was evaluated on the x_{G} in the oracle and the provided x_{E}.

The simulator $\mathcal{S}$ proceeds as follows. It runs as the honest G would do, but with two modifications. 1) It uses dummy inputs $x'_{\mathsf{id}} = 0$ for all G-in-gates and 2) there is a modification in OUTPUTE, which we describe below.

For all E-in-gates id it inspects the commitment functionality and learns K^0_{id} and Δ_{id} and computes K^1_{id} from these. From all E-in-gates it can then retrieve $K_{\mathsf{id}} = K^{x_{\mathsf{id}}}_{\mathsf{id}}$ which is well defined as G is honest and therefore followed the protocol. This defined the inputs x_{id} of E. Then it calls its oracle with those input bits of E and for all E-out-gates id it receives from the oracle the output y_{id} obtained by running $\mathcal{C}$ in plaintext with those inputs x_{id} and the unknown input bits of G.

If the protocol aborts, then the simulator aborts too.

If the protocol reaches an execution of OUTPUTE, then the simulator will retrieve the output y_{id} learned from $\mathcal{O}_{x_{\mathsf{G}}}(x_{\mathsf{E}})$. Then it retrieves the key K_{id_1} and computes

$$\sigma'_{\mathsf{id}_1} = \mathrm{lsb}(K_{\mathsf{id}}) \oplus y_{\mathsf{id}} \ .$$

Then it runs OUTPUTE as in the protocol. To understand why we make this change, recall that $\mathcal{S}$ ran G with dummy inputs, so it might be the case that $K_{\mathsf{id}.1}$ encodes a different output than y_{id}. Therefore, sending $\sigma_{\mathsf{id}.1}$ might result in E getting a wrong output, which would allow it to learn that it is in the simulation.

We should now argue that the value seen by G in the simulation and in the real execution are computationally indistinguishable. Notice that if we ran the simulation but using the real inputs x_{E} for E instead of dummy inputs, then in all executions of OUTPUTE it would be the case that $\sigma_{\mathsf{id}_1} = \mathrm{lsb}(K_{\mathsf{id}}) \oplus y_{\mathsf{id}} = \sigma_{\mathsf{id}_1}$ for all E-out-gates. Therefore there is not really a modification of the commitment functionality when OUTPUTE is simulated. Therefore, the simulation run with real inputs is just the the real protocol. This means that it is sufficient to argue that the values seen by G in the simulation with dummy input x'_{E} and with real input x_{E} are computationally indistinguishable.

We do that via a reduction to obliviousness of the reactive garbling scheme and the fact that H is a random oracle. The reduction will have access to the real input of x_{E} and an oracle producing garbled circuits along with encodings of either dummy inputs or the real inputs. It will then augment these to make them look like a run of the simulation with dummy inputs or real inputs.

In a bit more detail, in the obliviousness game in [NR16] we will have access to an oracle $\mathcal{O}_b$ for a uniformly random bit b. We can give $\mathcal{O}_b$ a command of the form $(\mathsf{garble}, \mathsf{id}, C)$. Then it garbles C and gives us F_{id}, keeping the keys secret. If we later give the command $(\mathsf{reveal}, \mathsf{id})$, then we are given all keys used in garbling F_{id}. We can also give the command $(\mathsf{link}, \mathsf{id}_1, i_1, \mathsf{id}_2, i_2)$. The oracle will release the information used to solder output wire i_1 in F_{id_1} onto input wire i_2 in F_{id_2} as specified in GENSOLD. We can also give the command $(\mathsf{input}, \mathsf{id}, i, x_0, x_1)$. The oracle will compute the keys K^0 and K^1 for input wire i in F_{id}. It then gives us K^{x_b}. There are some natural restrictions. There is not allowed to be any loops in linking wires between circuits. No identifier is allowed to occur in both a REVEAL and a LINK command. No wire (id_2, i_2) is allow to occur in two different call to $(\mathsf{input}, \mathsf{id}_2, i_2, \cdot, \cdot)$ or two calls to $(\mathsf{link}, \cdot, \cdot, \mathsf{id}_2, i_2)$. No input wire is allowed to occur in both a call $(\mathsf{input}, \mathsf{id}_2, i_2, \cdot, \cdot)$ and a call $(\mathsf{link}, \cdot, \cdot, \mathsf{id}_2, i_2)$. At the end we have to make a guess at b. The security definition says that no poly-time adversary can guess b with better than negligible probability. The definition of authenticity say that the adversary cannot for some wire id compute both K^0_{id} and K^1_{id} given the information it receives in the game.

In fact, the definition in [NR16] does not have the REVEAL command. We add this command here, getting a notion of adaptive, reactive garbling. It is straight forward to verify that the scheme in [AHMR15] is an adaptive, reactive garbling scheme in the above sense. This is achieved by using the strong programable random oracle model.

We need to add an additional command $(\mathsf{difdif}, \mathsf{id}_1\mathsf{id}_2)$. In response to this the oracle will release $\Delta_{\mathsf{id}_1} \oplus \Delta_{\mathsf{id}_2}$, where Δ_{id_i} is the difference used to garble F_{id_i}. No identifier is allowed to occur in both a DIFDIF and a REVEAL command, i.e., one is only allowed to learn $\Delta_{\mathsf{id}_1} \oplus \Delta_{\mathsf{id}_2}$ when both Δ_{id_1} and Δ_{id_2} are unknown. It is straight forward to verify that the scheme in [AHMR15] is an adaptive, reactive garbling scheme even when this command is added. The scheme would trivially still be secure if the same Δ was used on two unrevealed garblings, i.e., if $\Delta_{\mathsf{id}_1} = \Delta$ and $\Delta_{\mathsf{id}_2} = \Delta$ for random Δ. To see this, consider garbling two circuits C_1 and C_2 by garbling the circuit $C_1 \| C_2$ which runs the two circuits in parallel. This would exactly provide garblings of C_1 and C_2 with a common Δ. It is straight forward to go through the proof of [AHMR15] and verify that first garbling using independent Δ_{id_1} and Δ_{id_2} and then releasing $\Delta_{\mathsf{id}_1} \oplus \Delta_{\mathsf{id}_2}$ does not break the security. To see this note that in the proof, when id_1 and id_2 are unrevealed, then the distribution of Δ_{id_i} is statistically close to uniform, except that $\mathrm{lsb}(\Delta_{\mathsf{id}_i}) = 1$. Furthermore, the security depends only on Δ_{id_i} being statistically close to uniform, not that the Δ_{id_i} are independent, which is what allows that $\Delta_{\mathsf{id}_1} = \Delta_{\mathsf{id}_2}$. This means we can simulate $\Delta_{\mathsf{id}_1} \oplus \Delta_{\mathsf{id}_2}$ by a uniformly random value Δ with $\mathrm{lsb}(\Delta) = 0$, as it is still ensured that each Δ_{id_i} has full entropy given Δ.

We will go over each of the sub-protocols and argue how to simulate the values sent to E.

When the simulation is run with dummy inputs for G, we call it the *dummy mode*. When the simulation is run with real inputs for G, we call it the *real mode*.

When we say that a protocol is trivial to simulate it means that the protocol sends no values to E, so there are no values to simulate.

$\triangleright$ $\mathsf{GenWire}(\mathsf{id}, K, \Delta)$: When this sub-simulator is called the key material $(K^0_{\mathsf{id}}, \Delta_{\mathsf{id}})$ is already defined and sitting inside $\mathcal{O}$ as part of some circuit. This also defines K^1_{id} and σ_{id}. The simulator is not given these values. We then simulate the commitments by sending E notifications that the commitments were done. These value are clearly the same in the dummy and the real mode. Notice that by this way of simulating, it holds that for each wire id which has associated key material in the protocol, the oracle $\mathcal{O}_b$ will hold this key material, so we can work on it using the interface of $\mathcal{O}_b$.

$\triangleright$ $\mathsf{VerWire}(\mathsf{id})$: Here we send all the key material to E. The simulator will ask its oracle $\mathcal{O}_b$ to reveal the circuit that $(K^0_{\mathsf{id}}, \Delta_{\mathsf{id}})$ is part of. This will give it the needed key material. Then it patches the commitment functionality to open to those values before decommitting.

$\triangleright$ $\mathsf{GenSold}(\mathsf{id}_1, \mathsf{id}_2)$: The values sent to E here are exactly the soldering values of the reactive garbling scheme [AHMR15]. The simulator can therefore request to get the values from its oracle.

$\triangleright$ $\mathsf{EvSold}(\mathsf{id}_1, \mathsf{id}_2, K)$: Trivial.

$\triangleright$ $\mathsf{GenKeyAuth}(\mathsf{id})$: The simulator will first ask its oracle $\mathcal{O}_b$ to make a garbling of a circuit with one input wire and one output wire and one gate which is the identity gate. Such a garbling consist simply of K^0_{id} and Δ_{id}. So now the key material $(K^0_{\mathsf{id}}, \Delta_{\mathsf{id}})$ is defined and sitting inside $\mathcal{O}$ as part of some circuit. We call the simulator for GENWIRE on these. We additionally have to simulate the value $A_{\mathsf{id}} = \{H(K^0_{\mathsf{id}}), H(K^1_{\mathsf{id}})\}$. This is complicated by the fact that $\mathcal{S}$ does not know K^0_{id} or K^1_{id}, as the key material is sitting inside $\mathcal{O}_b$. Recall, however, that we assume a programable random oracle. We can therefore simply sample two uniformly random value $h, h' \leftarrow \{0,1\}^\kappa$ and let $A_{\mathsf{id}} = \{h, h'\}$. If E later queries H in an unrevealed key authenticator on its key $K_{\mathsf{id}} \in \{K^0_{\mathsf{id}}, K^1_{\mathsf{id}}\}$, then we return h. Except with negligible probability it will never query on the other key, as this would break authenticity. Notice that we did not pick whether $H(K^0_{\mathsf{id}}) = h$ or $H(K^0_{\mathsf{id}}) = h'$. We do not need to do this as A_{id} is sent as a set. This is important as we do not know the value of K_{id}.

$\triangleright$ $\mathsf{VerKeyAuth}(\mathsf{id})$: Here we first simulate VERWIRE and learn K^0_{id} and K^1_{id}. Then define $H(K^0_{\mathsf{id}}) := h$ or $H(K^1_{\mathsf{id}}) := h'$. If the oracle had been called on K^0_{id} or K^1_{id} before, this might give an inconsistent simulation. However, if the oracle had been called on K^0_{id} or K^1_{id} before, then E guessed one of the keys without being given any information about the key. Since the keys are uniformly random this happens with negligible probability.

$\triangleright$ $\mathsf{GenInKeyAuth}(\mathsf{id})$: This protocol is simulated as GenKeyAuth, but we additionally have to ensure that $H(\varDelta_{\mathsf{id}}) = K^0_{\mathsf{id}}$. The values $\varDelta_{\mathsf{id}}$ and K^1_{id} are not known to $\mathcal{S}$, as they are sitting inside $\mathcal{O}_b$. But we can still define that $H(\varDelta_{\mathsf{id}}) = K^0_{\mathsf{id}}$. We will only have to return K^0_{id} if H is ever evaluated on $H(\varDelta_{\mathsf{id}})$. If id is never verified, this would involve E guessing $\varDelta_{\mathsf{id}}$ after being given only one key, which would break authenticity of the garbling scheme. For the case where id is verified, see below.

$\triangleright$ $\mathsf{VerInKeyAuth}(\mathsf{id})$: Simulated as $\mathsf{VerInKeyAuth}$. This lets $\mathcal{S}$ learn K^0_{id} and $\varDelta_{\mathsf{id}}$. Then it programs H to $H(\varDelta_{\mathsf{id}}) = K^0_{\mathsf{id}}$. If the oracle had been called on $\varDelta_{\mathsf{id}}$ before, this might give an inconsistent simulation. However, if the oracle had been called on $\varDelta_{\mathsf{id}}$ before the wire it verified, then E guessed this value without being given any information about $\varDelta_{\mathsf{id}}$. Since $\varDelta_{\mathsf{id}}$ has $\kappa - 1$ bits of min-entropy this happens with negligible probability.

$\triangleright$ $\mathsf{PreProcessKA}()$: The protocol is simulated by running honestly and simulating all the calls to GenKeyAuth, VerKeyAuth and GenSold. No additional values are sent.

$\triangleright$ $\mathsf{EvKAs}(\mathsf{id}, \mathcal{K}_{\mathsf{id}})$: Trivial.

$\triangleright$ $\mathsf{InputG}(\mathsf{id})$: Let $x'_{\mathsf{id}} = 0$ be the input bit of G in dummy mode and let x_{id} be the input bit of G in real mode. Ask the oracle $\mathcal{O}_b$ to get the encoded input for $\mathsf{id.ka.1}$. Submit the bit-pair $(x'_{\mathsf{id}}, x_{\mathsf{id}})$. The simulator learns $K = K^x_{\mathsf{id.ka.1}}$, where $x = x'_{\mathsf{id}}$ if $b = 0$ and $x = x_{\mathsf{id}}$ if $b = 1$. Send K to E. The distribution of K is exactly as in dummy mode when $b = 0$ and exactly as in real mode when $b = 1$.

$\triangleright$ $\mathsf{PreProcessOTInit}()$: It was shown in [NST17] how to simulate this protocol. This can be done without knowing $\varDelta_{\mathsf{ot}}$. This is important as we need to implicitly define $\varDelta_{\mathsf{ot}}$ below.

$\triangleright$ $\mathsf{InputE}(\mathsf{id})$: When running this protocol the simulator will pick $R_{b_{\mathsf{ot}_{\mathsf{id}}}}$ uniformly a random and output this to E from the OT. This is needed as the simulator does not know $\varDelta_{\mathsf{ot}}$. Now E receives the values D, σ_{id} an S_{id} along with some notification values there are trivial to simulate. It is therefore enough to show how to simulate these three values. Let $b_{\mathsf{ot}_{\mathsf{id}}}$ be the choice bit of E in the call to the OT. The simulator can inspect the ideal functionality and learn this value. Let f_{id} be the bit sent by E. Let $x_{\mathsf{id}} \leftarrow f_{\mathsf{id}} \oplus b_{\mathsf{ot}_{\mathsf{id}}}$. Define this to be the input of E. Ask the oracle $\mathcal{O}_b$ to get the encoded input for id'. Submit the bit-pair $(x_{\mathsf{id}}, x_{\mathsf{id}})$. The simulator learns $K = K^{x_{\mathsf{id}}}_{\mathsf{id}'}$. By definition we have that $\mathrm{lsb}(K_{\mathsf{id}'}) = x_{\mathsf{id}} \oplus \sigma_{\mathsf{id}}$, which allows to simulate σ_{id} as

$$\sigma_{\mathsf{id}} = \mathrm{lsb}(K_{\mathsf{id}'}) \oplus x_{\mathsf{id}} \ ,$$

as the simulator knows $K_{\mathsf{id}'}$ and x_{id}. To simulate S_{id} we send a uniformly random value. This will implicitly define $\varDelta_{\mathsf{ot}}$ by $\varDelta_{\mathsf{ot}} = S_{\mathsf{id}} \oplus \varDelta_{\mathsf{id}'}$, where the simulator does not know $\varDelta_{\mathsf{id}'}$ as it is sitting inside $\mathcal{O}$. A little more care must be done. The above simulation of S_{id} works for the first call to InputG. For the next call (with identifier $\widehat{\mathsf{id}}$ say) we should ensure that $\widehat{\varDelta}_{\mathsf{ot}} = S_{\widehat{\mathsf{id}}} \oplus \varDelta_{\widehat{\mathsf{id}}'}$ becomes defined to the same $\varDelta_{\mathsf{ot}}$ as when we simulated for id. This means that we should ensure that $S_{\mathsf{id}} \oplus \varDelta_{\mathsf{id}'} = S_{\widehat{\mathsf{id}}} \oplus \varDelta_{\widehat{\mathsf{id}}'}$, which is equivalent to $S_{\widehat{\mathsf{id}}} = S_{\mathsf{id}} \oplus \varDelta_{\mathsf{id}'} \oplus \varDelta_{\widehat{\mathsf{id}}'}$. To ensure this the simulator ask $\mathcal{O}_b$ for $\varDelta_{\mathsf{id}'} \oplus \varDelta_{\widehat{\mathsf{id}}'}$ using the difdif-command and computes $S_{\widehat{\mathsf{id}}}$ as required. By inspection of the protocol we see that $K = D \oplus R_{b_{\mathsf{ot}_{\mathsf{id}}}} \oplus (x_{\mathsf{id}} \oplus \sigma_{\mathsf{id}}) S_{\mathsf{id}}$. This allows to simulate D as

$$D = K \oplus R_{b_{\mathsf{ot}_{\mathsf{id}}}} \oplus (x_{\mathsf{id}} \oplus \sigma_{\mathsf{id}}) S_{\mathsf{id}} \ ,$$

as the simulator can compute K, $R_{b_{\mathsf{ot}_{\mathsf{id}}}}$, x_{id}, σ_{id} and S_{id} at the time where it needs to send D. It compute K as $K = K_{\mathsf{id}'}$. It picked $R_{b_{\mathsf{ot}_{\mathsf{id}}}}$ itself. It already computed x_{id} and σ_{id} and S_{id}.

$\triangleright$ $\mathsf{GenSub}(\mathsf{id}, C)$: We simulate by asking $\mathcal{O}_b$ to generate and give us F_{id}. This also defines key materials $(K_i, \Delta_{\mathsf{id}})_{i=1}^{n}$ and $(L_i, \Delta_{\mathsf{id}})_{i=1}^{m}$ sitting inside $\mathcal{O}$. We call the simulator for GenWire to simulate these.

$\triangleright$ $\mathsf{VerSub}(\mathsf{id}, C)$: We simulate by asking $\mathcal{O}_b$ to reveal F_{id}. This gives us the key materials $(K_i, \Delta_{\mathsf{id}})_{i=1}^{n}$ and $(L_i, \Delta_{\mathsf{id}})_{i=1}^{m}$ sitting inside $\mathcal{O}$. We call the simulator for VerWire with these.

$\triangleright$ $\mathsf{EvSub}(\mathsf{id})$: Trivial.

$\triangleright$ $\mathsf{PreProcessSub}()$: This protocol including the call to GenSoldSub is simulated by simulating the calls to GenSub, VerSub and GenSold as described above. No further messages are sent.

$\triangleright$ $\mathsf{AssembleSubs}()$: This protocol is simulated by simulating the calls to GenSold as described above. No further messages are sent.

$\triangleright$ $\mathsf{EvSubs}(\mathsf{id})$: Trivial.

$\triangleright$ $\mathsf{OutputE}(\mathsf{id})$: Recall that here the simulator will retrieve the output y_{id} learned from $\mathcal{O}_{x_{\mathsf{G}}}(x_{\mathsf{E}})$ and the key K_{id} held by E and send $\sigma'_{\mathsf{id}_1} = \mathrm{lsb}(K_{\mathsf{id}}) \oplus y_{\mathsf{id}}$ to E. We do the same in the reduction.

$\triangleright$ $\mathsf{OutputG}(\mathsf{id})$: Trivial.

$\triangleright$ $\mathsf{RecoverInputBit}$: Trivial.

$\triangleright$ $\mathsf{PreProcessRecovery}$: Here we have to give away all the differences $\Delta_{\mathsf{rco},\mathsf{id}}$ and $\Delta_{\mathsf{id},\mathsf{rco}}$.

Consider the first $\mathsf{id} \in \mathcal{E} \cup \mathcal{I}$. We need to compute $\Delta_{\mathsf{id},\mathsf{rco}} = \Delta_{\mathsf{id.ka.1}} \oplus \Delta_{\mathsf{rco}}$ for an unknown $\Delta_{\mathsf{id.ka.1}}$. We do this by picking $\Delta_{\mathsf{id},\mathsf{rco}}$ uniformly at random and defining

$$\Delta_{\mathsf{rco}} = \Delta_{\mathsf{id.ka.1}} \oplus \Delta_{\mathsf{id},\mathsf{rco}} \ .$$

This is possible as we never have to release Δ_{rco} so we do not need to know it. Consider the remaining $\widehat{\mathsf{id}} \in \mathcal{E} \cup \mathcal{I}$. We need to compute

$$\begin{aligned}\Delta_{\widehat{\mathsf{id}},\mathsf{rco}} &= \Delta_{\widehat{\mathsf{id}}.\mathsf{ka.1}} \oplus \Delta_{\mathsf{rco}} \\ &= \Delta_{\widehat{\mathsf{id}}.\mathsf{ka.1}} \oplus \Delta_{\mathsf{id.ka.1}} \oplus \Delta_{\mathsf{id},\mathsf{rco}} \ .\end{aligned}$$

We can do this as we can get $\Delta_{\widehat{\mathsf{id}}.\mathsf{ka.1}} \oplus \Delta_{\mathsf{id.ka.1}}$ using the DIFDIF-command and we know $\Delta_{\mathsf{id},\mathsf{rco}}$. We compute the values $\Delta_{\mathsf{rco},\mathsf{id}}$ similarly.

$\triangleright$ $\mathsf{RecoverInputBits}$: Trivial.

It now follows that if $b = 0$, then the distribution of the reduction is exactly that of the dummy model and if $b = 1$, then the distribution of the reduction is exactly that of the real model. Since we only did allowed queries to $\mathcal{O}_b$, it follows that the dummy mode and the real are computationally indistinguishable.