Stacked Garbling for Disjunctive Zero-Knowledge Proofs

David Heath(✉) and Vladimir Kolesnikov(✉)

Georgia Institute of Technology, Atlanta, GA, USA
{heath.davidanthony,kolesnikov}@gatech.edu

Abstract. Zero-knowledge (ZK) proofs (ZKP) have received wide attention, focusing on non-interactivity, short proof size, and fast verification time. We focus on the fastest total proof time, in particular for large Boolean circuits. Under this metric, Garbled Circuit (GC)-based ZKP (Jawurek et al., [JKO], CCS 2013) remained the state-of-the-art technique due to the low-constant linear scaling of computing the garbling.

We improve GC-ZKP for proof statements with conditional clauses. Our communication is proportional to the longest branch rather than to the entire proof statement. This is most useful when the number m of branches is large, resulting in up to factor $m\times$ improvement over JKO.

In our proof-of-concept **illustrative application**, prover P demonstrates knowledge of a bug in a codebase consisting of *any number* of snippets of **actual C code**. Our computation cost is linear in the size of the codebase and communication is *constant in the number of snippets*. That is, we require only enough communication for a single largest snippet!

Our **conceptual contribution** is *stacked garbling for ZK*, a privacy-free circuit garbling scheme that can be used with the JKO GC-ZKP protocol to construct more efficient ZKP. Given a Boolean circuit $\mathcal{C}$ and computational security parameter κ, our garbling is $L \cdot \kappa$ bits long, where L is the length of the longest execution path in $\mathcal{C}$. All prior concretely efficient garbling schemes produce garblings of size $|\mathcal{C}| \cdot \kappa$. The computational cost of our scheme is not increased over prior state-of-the-art.

We implement our GC-ZKP and demonstrate significantly improved ($m\times$ over JKO) ZK performance for functions with branching factor m. Compared with recent ZKP (STARK, Libra, KKW, Ligero, Aurora, Bulletproofs), our scheme offers much better proof times for larger circuits (35-1000× or more, depending on circuit size and compared scheme).

For our illustrative application, we consider four C code snippets, each of about 30–50 LOC; one snippet allows an invalid memory dereference. The entire proof takes 0.15 s and communication is 1.5 MB.

Keywords: Garbled circuits · Inactive branch elimination · ZK · Proof of C bugs

---

Electronic supplementary material The online version of this chapter (https://doi.org/10.1007/978-3-030-45727-3_19) contains supplementary material, which is available to authorized users.

A. Canteaut and Y. Ishai (Eds.): EUROCRYPT 2020, LNCS 12107, pp. 569–598, 2020.
https://doi.org/10.1007/978-3-030-45727-3_19

1 Introduction

Zero-knowledge (ZK) proofs (ZKP) have a number of practical applications; reducing their cost is an active research direction. Many efficient schemes were recently proposed, focusing on small proofs and fast verification. These works are largely motivated by blockchain applications [AHIV17, BCR+19, BBB+18, WTs+18, XZZ+19, BBHR19, etc.] and also by post-quantum signatures [CDG+17, KKW18].

Our focus, in contrast, is on the classical setting of *fastest total proof time*, including (possibly interactive) proof generation, transmission, and verification. In this total-time metric, Yao's garbled circuits (GC) is the fastest and one of the most popular techniques for proving general NP statements (expressed as Boolean circuits) in ZK. GC offers low-overhead linear prover complexity, while other techniques' provers are either superlinear or have high constants.

[JKO13] and [FNO15] demonstrate how to use GC for ZK without the costly cut-and-choose technique, while [ZRE15] proposes an efficient garbling technique that requires only 1 cryptographic ciphertext per AND gate in the ZK setting. As a result, GC-ZKP can process 20 million AND gates per second or more on a regular laptop (XOR gates are essentially free [KS08]). Unfortunately, while the computational cost of GC-ZKP is low, the communication is high. Even a fast 1 Gbps LAN can support only ≈6 million AND gates per second (XOR gates are free in communication). While this rate is higher than all recent NIZK systems, further communication improvements would make the approach even stronger.

In this work we achieve such a communication improvement. We reduce the cost of sending a GC when the proof statement contains logically disjoint clauses (i.e. conditional branches in `if` or `switch` constructs). In particular, if a logical statement contains disjoint clauses, then the cost to transmit the GC is bounded by the size of the largestclause rather than the total size of all clauses.

Our key idea is that the proof verifier (who is the GC generator) garbles from seeds all the clauses and then XORs together, or *stacks*, the garblings before sending them to the prover for evaluation. The prover receives via OT the seeds for the inactive clauses, reconstructs their garblings, and then XORs them out to obtain the target clause's garbling. By stacking the garblings, we decrease the cost to transmit the GC from the verifier to the prover.

In Sect. 3, we formally present our approach as a garbling scheme, which we call Privacy-Free Stacked (PFS) garbling. Accompanying proofs are in Sect. 4. We implement our approach in C++ and evaluate its performance against state-of-the-art techniques in Sect. 6 (see also discussion in Sect. 1.6).

1.1 Use Cases: Hash Trees and Existence of Bugs in Program Code

Our technique is useful for proving in ZK one of several statements.

Consider proving arbitrary statements in ZK, represented as Boolean circuits. These can be straightline programs or, more generally and quite typically, will include logical combinations of basic clauses. Several lines of work consider ZK of general functions, including MPC-in-the-head, SNARKs/STARKs, JKO, Sigma protocols [CDS94]; the latter specifically emphasizes proving disjoint statements, e.g., [Dam10, CDS94, CPS+16, GK14].

We now briefly present our two main applications (cf. Sects. 6 and 7):

App 1: Existence of Bugs. Our most exciting application allows a prover P to demonstrate knowledge of a bug in a potentially large codebase. We stress that ours is not a full-strength automated application, but rather a proof of concept. Still, we are able to handle C code with pointers, standard library calls, and simple data structures (see Sect. 7).

We consider a number of code snippets motivated by real code used in operating systems, standard algorithms, etc. The snippets we consider contain between 30 and 50 lines of code, but this number can be easily increased. We manually instrument each snippet with program assertions. Each snippet outputs a single bit that indicates if any assertion failed, and hence whether there is a bug.

We used and extended the EMP toolkit [WMK16] to compile instrumented snippets to Boolean circuits. Now, P can demonstrate she knows an input to a snippet, resulting in output 1. We envision that the mechanical tasks of instrumenting a codebase and splitting it into snippets will be automated in a practical tool; we leave further development as important and imminent future work.

Our approach excels in this use case because it features (1) high concrete performance and (2) communication that is constant in the number of code snippets. We further elaborate on this use case in Sect. 7.

App 2: Merkle Tree Membership. We wish to compare the performance of our PFS garbling to recent ZKP systems. We therefore consider a typical application considered in the literature: proof of membership in a Merkle tree.

Specifically, Alice wishes to assert properties of her record R embedded in a certificate signed by one of several acceptable authorities (CAs). Each CA A_i includes a number of different players' records $R_1^i, ..., R_n^i$ in a Merkle tree, and securely publishes its root. Alice receives her record R_j^k (which may embed a secret) and the Merkle tree hashes on the path to root. Now, Alice can prove statements in ZK about R_j^k with respect to any set of the published roots. CAs may use different hash functions for Merkle trees, or, in general, differ in other aspects of the proof, thus creating a use case for proving one of many clauses. In Sect. 6, we compare our performance to recent work based on this use case.

1.2 Key Contributions

- Conceptual contribution: A novel GC technique, which we call *stacked*, or PFS, garbling, requiring garbled material linear in the longest execution path, rather than in the full size of the circuit. Specifically, the same material sent from the verifier to the prover can represent the execution of any of the disjoint clauses. Note, `Free IF` technique [Kol18] *does not* work in our setting.
- High concrete performance, improving over the state-of-the art baseline (JKO+half-gates) approximately by the function branching factor; improvement over recent SNARKs is 35× − 1000× or more, depending on function size, branching, and compared scheme. Our technique has low RAM requirements (146 MB for 7M gate circuit).

- A proof of concept system that allows proving knowledge of a bug in C code. We use realistic C code snippets, which include pointers and standard library calls, and prove a bug related to incorrect use of **`sizeof()`** on a pointer.

1.3 Preliminaries

Free IF *review:* First, we review Kolesnikov's Free IF approach [Kol18]. Free IF decouples circuit topology (i.e. wire connections among the gates) from cryptographic material used to evaluate gates (i.e. encrypted gate tables). While a topology is needed to evaluate a circuit, it is assumed to be conveyed to the evaluator, Eval, separately from the garbled tables, or by implicit agreement between the participants Eval and GC generator Gen.

Let $\mathcal{S} = \{\mathcal{C}_1, ..., \mathcal{C}_m\}$ be a set of Boolean circuits. Let (only) Gen know which circuit in $\mathcal{S}$ is evaluated, and let $\mathcal{C}_t$ be this target circuit. The key idea of [Kol18] is that Gen constructs cryptographic material for $\mathcal{C}_t$, but does *not* construct material for the other circuits. Let $\widehat{\mathcal{C}}$ be the constructed cryptographic material. The circuits in $\mathcal{S}$ may have varying topologies, but $\widehat{\mathcal{C}}$ is a collection of garbled tables that can be interpreted as the garbling of any of these topologies. Eval knows $\mathcal{S}$, but does not know which circuit is the target. For each $\mathcal{C}_i \in \mathcal{S}$, Eval interprets $\widehat{\mathcal{C}}$ as cryptographic material for $\mathcal{C}_i$ and evaluates, obtaining garbled output. Only the output labels of $\mathcal{C}_t$ encrypt truth values; the other output labels are garbage. Eval cannot distinguish the garbage labels from the valid labels, and hence cannot distinguish which of the circuits in $\mathcal{S}$ is the target circuit $\mathcal{C}_t$.

Next, Eval obliviously propagates (only) the target output labels to Gen via an *output selection* protocol. As input to the protocol, Eval provides all output labels (including the garbage outputs), and Gen provides the index t as well as $\mathcal{C}_t$'s zero labels on output wires. The output selection protocol outputs (re-encoded) labels corresponding to the output of $\mathcal{C}_t$.

While our technique is different, PFS garbling is inspired by the key ideas from Free IF: (1) Separating the topology of a circuit from its garbled tables and (2) using the same garbling to securely evaluate several topologies.

Superficially, both [Kol18] and we omit inactive clauses when one of the players (Gen in [Kol18] and Eval in our work) knows the target clause. Indeed, in GC ZK, Gen *must not* know the evaluated branch. This is a critical distinction that requires a different approach. We present this new approach in this work.

Garbled Circuits for Zero Knowledge: Until the work of Jawurek et al. [JKO13], ZK research focused on proofs of algebraic statements. Generic ZKP techniques were known, but were based on generic NP reductions and were inefficient. [JKO13] provides an efficient generic ZKP technique based on garbled circuits.

The construction works as follows: The Verifier, V, and the Prover, P, run a passively-secure Yao's GC protocol, where V acts as the circuit generator and P acts as the circuit evaluator. The agreed upon Boolean circuit, $\mathcal{C}$, is an encoding of the proof relation where (1) the input is a witness supplied by P, (2) the output is a single bit, and (3) if the output bit is 1, then the witness satisfies the relation. V garbles $\mathcal{C}$ and sends the garbling to P. P evaluates the GC and sends V the output label. The security of Yao's protocol (namely the authenticity property [BHR12]) ensures that a computationally bounded P can only produce the correct output label by running the circuit with a valid witness as input. By computing $\mathcal{C}$, P and V have achieved a ZK proof in the honest verifier setting.

A malicious V can violate ZK security by sending an invalid circuit or invalid OT inputs, which can leak P's inputs. [JKO13] solves this as follows: P does not immediately send the output label to V, but instead commits to it. Then V sends the seed used to generate the GC. P uses the seed to verify that the GC was honestly constructed. If so, P can safely open the commitment to the output label, completing the proof. [JKO13] consider a generalization of the above that does not require V to construct GCs from seeds. Instead, they define the notion of *verifiable* garbling. Verifiability prevents V from distinguishing different witnesses used by the prover, and therefore from learning something about P's input. Specifically, a garbling scheme is verifiable if there is a verification procedure such that even a malicious V cannot create circuits that both (1) satisfy the procedure and (2) output different values depending on the evaluator's witness.

In this work, we deal with explicit randomness and generate GCs from seeds. It is possible to generalize our work to the verifiable formulation of [JKO13].

Subsequent to the [JKO13] work, [FNO15] observes that weaker *privacy-free* garbling schemes are sufficient for the ZK construction of [JKO13]. [FNO15] construct a more efficient privacy-free garbling, whose cost is between 1 and 2 ciphertexts per AND gate. Zahur et al. [ZRE15] present a privacy-free variant of their half gates scheme, which requires only 1 ciphertext per AND gate, and is compatible with the JKO/FNO schemes. In our implementation, we use these state-of-the-art constructions. Because our work leverages the protocol from [JKO13], we will include their protocol in the full version of this paper.

1.4 High-Level Approach

Our main contribution is a new ZKP technique in the [JKO13] paradigm. The key characteristic of our construction is that for proof relations with disjoint clauses (i.e. conditional branches), communication is bounded by the size of the largest clause rather than the total size of the clauses. In Sect. 3, we present our approach in technical detail as a garbling scheme which can be plugged into the [JKO13] protocol. For now, we explain our approach at a high level.

Consider the proof of a statement represented by a Boolean circuit $\mathcal{C}$ with conditional evaluation of one of several clauses. In Sect. 1.3, we reviewed existing work that demonstrates how to efficiently evaluate $\mathcal{C}$ if the circuit *generator* knows the active clause. However, the [JKO13] ZK approach requires the generator to be V. Unfortunately, V has no input and therefore does not know the target clause. Instead, P must select the target clause.

As a naïve first attempt, P can select 1-out-of-m garbled circuits via OT. However, this involves transferring all GC clauses, resulting in no improvement.

Instead, we propose the following idea, inspired by a classic two-server private information retrieval approach [CGKS95].[1] Let $\mathcal{S} = \{\mathcal{C}_1, .., \mathcal{C}_m\}$ be the set of

---

[1] [CGKS95] includes a PIR protocol where two non-colluding servers separately respond to a client's two random, related queries by XORing elements of their result sets (and the client XORs out the true answer).

circuits implementing clauses of the ZK relation. Let $\mathcal{C}_t \in \mathcal{S}$ be the target clause that P wants to evaluate. For simplicity, suppose all clauses $\mathcal{C}_i$ are of the same size, meaning that they each generate GCs of equal size. Our approach naturally generalizes to clauses of different sizes (we discuss this in more detail in Sect. 3.7). The players proceed as follows.

V generates m random seeds $s_1..s_m$ and generates from them m GCs, $\widehat{\mathcal{C}}_1..\widehat{\mathcal{C}}_m$. V then computes $\widehat{\mathcal{C}} = \bigoplus \widehat{\mathcal{C}}_1..\widehat{\mathcal{C}}_m$ and sends $\widehat{\mathcal{C}}$ to P. Informally, computing $\widehat{\mathcal{C}}$ can be understood as *stacking* the different garbled circuits for space efficiency.

The key idea is that we will allow P to reconstruct (from seeds received via OT) all but one of the stacked GCs and then XOR these reconstructions out to retrieve the target GC, which P can evaluate with the witness she has. We must prevent P from receiving all m GCs and thus forging the proof. To do so, we introduce the notion of a 'proof of retrieval' string $\textsc{PoR}$. P receives $\textsc{PoR}$ via OT only when she *does not* choose to receive a clause seed. P proves that she has not forged the proof by showing that she knows $\textsc{PoR}$. This is put together as follows.

V generates a random proof of retrieval string $\textsc{PoR}$. For each $i \in \{1..m\}$, the players run 1-out-of-2 OT, where V is the sender and P is the receiver. Players use *committing* OT for this phase [KS06]. For the ith OT, V's input is a pair $(s_i, \textsc{PoR})$. P selects 0 as her input in all instances, except for instance t, where she selects 1. Therefore P receives $\textsc{PoR}$ and seeds $s_{i \neq t}$, from which P can reconstruct all GCs $\widehat{\mathcal{C}}_{i \neq t}$. P reconstructs the garbled material for the target clause by computing $\widehat{\mathcal{C}}_t = \widehat{\mathcal{C}} \oplus (\bigoplus_{i \neq t} \widehat{\mathcal{C}}_i)$.

Now, P received the garbling of the target clause, but we have not yet described how P receives input encodings for the target clause. We again simplify by specifying that each clause must have the same number, n, of input bits. Our approach generalizes to clauses with different numbers of inputs, as we discuss in Sect. 3.7. V's random seed s_i is used to generate the n pairs of input labels for each corresponding clause $\widehat{\mathcal{C}}_i$. Let X_i be the vector of n label pairs used to encode the input bits for clause i. V generates m such vectors, $X_1..X_m$. As an optimization similar to stacking the garbled circuits, V computes $X = \bigoplus X_1..X_m$. V and P now perform n committing 1-out-of-2 OTs, where in each OT V provides the two (stacked) possible input labels for a bit (a label corresponding to 0 and to a 1) and P provides the bit for that input. P uses the seeds obtained in the first step to reconstruct each $X_{i \neq t}$ and computes $X_t = X \oplus (\bigoplus_{i \neq t} X_i)$.

P now has the garbling $\widehat{\mathcal{C}}_t$ and appropriate input labels X_t. Therefore, P can evaluate $\widehat{\mathcal{C}}_t$ with the input labels and receive a single output label Y_t. For security, we must prevent V from learning t, so we must hide which clause P received output from. We accomplish this by allowing P to compute the correct output label for *every* clause. Recall that P has the seeds for every non-target clause. P can use the garblings constructed from these seeds to obtain the output labels $Y_{i \neq t}$. P computes $Y = \textsc{PoR} \oplus (\bigoplus Y_1..Y_m)$ and commits to this value (as suggested in [JKO13]). Next, V opens all commitments made during rounds of OT. From this, P checks that $\textsc{PoR}$ is consistent across all seed OTs and obtains the final seed s_t. P checks that the circuits are properly constructed by regarbling them from the seeds (and checking the input labels and garbled material) and, if so, completes the proof by opening the output commitment.

1.5 Generality of Top-Level Clauses

Our approach optimizes for *top-level* clauses. That is, possible execution paths of the proof relation must be represented by separate clauses. Top-level clauses are general: Even nested conditionals can be represented by performing program transformations that lift inner conditionals into top-level conditionals.

Unfortunately, over-optimistically lifting conditionals can sometimes lead to an exponential number of clauses. In particular, if two conditionals occur sequentially in the relation, then the number of possible execution paths is the product of the number of paths through both conditionals. Of course, it is not necessary to fully lift all conditionals in a program; individual clauses can include (unstacked) conditional logic. Our approach will yield improvement for any separation of top level clauses. Improving the described protocol to handle nested and sequential conditionals directly is a potential direction for improvement.

We emphasize that the notion of top-level clauses matches nicely with the target use case of proving the existence of program bugs: Programs can be split into various snippets, each of which may contain a bug. Each snippet can then be presented as a top-level proof clause.

1.6 Related Work

Our work is a novel extension of GC-based ZK [JKO13] which we reviewed in Sect. 1.3. Here we review other related work and provide brief comparisons in Sect. 1.7. We focus on recent concrete-efficiency protocols.

ZK. ZKP [GMR85,GMW91] is a fundamental cryptographic primitive. ZK proofs of knowledge (ZKPoKs) [GMR85,BG93,DP92] allow a prover to convince a verifier, who holds a circuit C, that the prover knows an input, or *witness*, w for which $C(w) = 1$. There are several flavors of ZK proofs. In this work we do not distinguish between computational and information-theoretic soundness, and thus refer to both arguments and proofs simply as 'proofs.'

ZK proofs were investigated both theoretically and practically in largely non-intersecting bodies of work. Earlier practical ZK protocols focused on algebraic relations, motivated mainly by signatures and identification schemes, e.g. [Sch90, CDS94]. More recently, these two directions have merged. Today, ZKPoKs and non-interactive ZKPoK (NIZKPoK) for arbitrary circuits are efficient in practice. Two lines of work stand out:

Garbled RAM combines GC with ORAM to repeatedly perform individual processor cycles instead of directly computing the program as a circuit [LO13]. Because the circuit needed to handle a cycle has fixed size, this groundbreaking technique has cost proportional to the program execution rather than to the full program. Garbled RAM must interface the GC with ORAM, making it not concretely efficient. While our approach is not as general as Garbled RAM, we achieve high concrete efficiency for conditions.

Efficient ZK from MPC. Ishai et al. (IKOS) [IKOS07], introduced the 'MPC-in-the-head' approach. Here, the prover emulates MPC evaluation of $C(w)$ among several virtual players, where w is secret-shared among the players. The verifier checks that the evaluation outputs 1 and asks the prover to open the views of some virtual players. A prover who does not have access to w must cheat to output 1; opening random players ensures a cheating prover is caught with some probability. At the same time, ZK is preserved because (1) not all virtual players are opened, (2) the witness is secret shared among the virtual players, and (3) MPC protects the inputs of the unopened virtual players.

Based on the IKOS approach, Giacomelli et al. [GMO16] implemented a protocol called ZKBoo that supports efficient NIZKPoKs for arbitrary circuits. Concurrently, Ranellucci et al. [RTZ16] proposed a NIZKPoK with similar asymptotics. Chase et al. [CDG+17] introduced ZKB++, which improves the performance of ZKBoo; they also showed that ZKB++ could be used to construct an efficient signature scheme based on symmetric-key primitives alone. Katz et al. [KKW18] further improved the performance of this approach by using MPC with precomputation. A version of the [CDG+17] scheme called *Picnic* [ZCD+17] was submitted to the NIST post-quantum standardization effort. The Picnic submission was since updated and is now based on [KKW18].

Ligero [AHIV17] offers proofs of length $O(\sqrt{|\mathcal{C}|})$, and asymptotically outperforms ZKBoo, ZKB++ and [KKW18] in communication. The break-even point between [KKW18] and Ligero depends on function specifics, and is estimated in [KKW18] to be ≈100K gates.

SNARKs/STARKs. Succinct non-interactive arguments of knowledge (SNARK) [GGPR13, PHGR13, BCG+13, CFH+15, Gro16] offer proofs that are particularly efficient in both communication and verification time. They construct proofs that are shorter than the input itself. Prior work demonstrated the feasibility of ZK proofs with size sublinear in the input [Kil92, Mic94], but were concretely inefficient. Earlier SNARKs require that their public parameters be generated and published by some semi-trusted party. This disadvantage motivated development of STARKs (succinct transparent arguments of knowledge) [BBHR18]. STARKs do not require trusted set up and rely on more efficient primitives. STARKs are succinct ZKP, and thus are SNARKs. In this work, we do not separate them; rather we see them as a body of work focused on sublinear proofs. Thus, Ligero [AHIV17], which is an MPC-in-the-head ZKP, is a SNARK.

In our comparisons, we focus on JKO, [KKW18], and recent SNARKs Ligero, Aurora, Bulletproofs [BBB+18], STARK [BBHR19], and Libra [XZZ+19].

1.7 Comparison with Prior Work

We present detailed experiment results in Sect. 6; here we reiterate that our focus and the main metric is *fastest total proof time*, including (possibly interactive) proof generation, transmission and verification. In this total-time metric, GC is the fastest technique for proving statements expressed as Boolean circuits. This is because GC offers low-overhead linear prover complexity, while other techniques' provers are superlinear, have high constants, or both.

In Sect. 1.1, we presented an exciting application where a prover demonstrates knowledge of a program bug. However, for comparison with prior work, the Merkle hash tree evaluation is most convenient, since many other works report on it. In Sect. 6, we implement our GC-ZK of Merkle hash tree and compare the results to JKO (which we reimplement as JKO was measured on older hardware), as well as to a variety of modern ZKP systems: KKW, Ligero, Aurora, Bulletproofs, STARK, and Libra.

As expected, our total time is improved over [JKO13] by a factor approximately equal to the branching factor. Indeed, our communication cost is linear in the longest execution path, while [JKO13,KKW18] are linear in $|\mathcal{C}|$, and our constants are similar to that of [JKO13] and significantly smaller than [KKW18].

Our total time outperforms current SNARKS by $35\times - 1,000\times$ or more. Like JKO, and unlike KKW and SNARKs, our technique is interactive and requires higher bandwidth.

2 Notation

The following are variables related to a given disjoint proof statement:

- t is the *target* index. It specifies the clause for which the prover has a witness.
- m is the number of clauses.
- n is the number of inputs. Unless stated otherwise, each clause has n inputs.

We simplify much of our notation by using $\oplus$ to denote a slight generalization of XOR: Specifically, if one of the inputs to XOR is longer than the other, the shorter input is padded by appending 0s until both inputs have the same length. We use $\bigoplus x_i..x_j$ as a vectorized version of this length-aware XOR:

$$\bigoplus x_i..x_j = x_i \oplus x_{i+1} \oplus \ldots x_{j-1} \oplus x_j$$

We discuss in Sect. 3.7 that this generalization is not detrimental to security in the context of our approach.

$x \mid\mid y$ refers to the concatenation of strings x and y. We use κ as the computational security parameter. We use V, he, him, his, etc. to refer to the verifier and P, she, her, etc. to refer to the prover. We use . for namespacing; `pack.proc` refers to a procedure `proc` defined as part of the package `pack`.

3 Our Privacy-Free Stacked Garbling Construction

We optimize the performance of ZK proofs for circuits that include disjoint clauses. In this section, we present our approach in technical detail.

We present our approach as a *verifiable garbling scheme* [BHR12,JKO13]. A verifiable garbling scheme is a tuple of functions conforming to a specific interface and satisfying certain properties such that protocols can be defined with the garbling scheme left as a parameter. Thus, new garbling schemes can

```
Proc $\mathit{Stack.Gb}(1^\kappa, f, R)$:
    $(f_1..f_m) \leftarrow f$
    $(\textsc{PoR}||s_1..s_m) \leftarrow R$
    for $i \in 1..m$ do
        $(F_i, e_i, d_i) \leftarrow \mathsf{Base.Gb}(1^\kappa, f_i, s_i)$
    $F \leftarrow f \;||\; \left(\bigoplus F_1..F_m\right)$
    $d \leftarrow \textsc{PoR} \oplus \left(\bigoplus d_1..d_m\right)$
    $e \leftarrow \textsc{PoR}||s_1..s_m||e_1..e_m$
    return $(F, e, d)$
```

```
Proc $\mathit{Stack.En}(e, x)$:
    $(\textsc{PoR} \;||\; s_1..s_m \;||\; e_1..e_m) \leftarrow e$
    $(t \;||\; x_t) \leftarrow x$
    for $i \in 1..m$ do
        if $i \neq t$ then
            $r_i \leftarrow s_i$
        else
            $r_i \leftarrow \textsc{PoR}$
        $X_i \leftarrow \mathsf{Base.En}(e_i, x_t)$
    $X \leftarrow r_1..r_m \;||\; \left(\bigoplus X_1..X_m\right)$
    return $X$
```

```
Proc $\mathit{Stack.De}(Y, d)$:
    $y \leftarrow Y = d$
    return $y$
```

```
Proc $\mathit{Stack.ev}(f, x)$:
    $(f_1..f_m) \leftarrow f$
    $(t \;||\; x_t) \leftarrow x$
    $y \leftarrow \mathsf{Base.ev}(f_t, x_t)$
    return $y$
```

```
Proc $\mathit{Stack.Ev}(F, X, x)$:
    $(f_1..f_m \;||\; F) \leftarrow F$
    $(r_1..r_m \;||\; X) \leftarrow X$
    $(t \;||\; x_t) \leftarrow x$
    for $i \in 1..m$ do
        if $i \neq t$ then
            $(F_i, e_i, d_i) \leftarrow \mathsf{Base.Gb}(1^\kappa, f_i, r_i)$
            $X_i \leftarrow \mathsf{Base.En}(e_i, x_t)$
        else
            $(F_i, d_i, X_i) \leftarrow (\mathbf{0}, \mathbf{0}, \mathbf{0})$
    $F_t \leftarrow F \oplus \left(\bigoplus F_1..F_m\right)$
    $X_t \leftarrow X \oplus \left(\bigoplus X_1..X_m\right)$
    $Y_t \leftarrow \mathsf{Base.Ev}(F_t, X_t)$
    $Y \leftarrow Y_t \oplus \left(\bigoplus d_1..d_m\right) \oplus r_t$
    return $Y$
```

```
Proc $\mathit{Stack.Ve}(f, F, e)$:
    $(\textsc{PoR} \;||\; s_1..s_m \;||\; \cdot) \leftarrow e$
    $(F', e', d') \leftarrow$
      $\mathsf{Stack.Gb}(1^\kappa, f, \textsc{PoR} \;||\; s_1..s_m)$
    return $e = e' \wedge F = F'$
```

Fig. 1. PFS garbling scheme `Stack`. `Stack` is defined as six procedures: Stack.Gb, Stack.Ev, Stack.ev, Stack.En, Stack.De, and Stack.Ve.

be easily plugged into existing protocols. That is, a garbling scheme *does not* specify a protocol. Instead, it specifies a modular building block.

We specify an efficient verifiable garbling scheme, where the function encoding, F, is proportional to the longest program execution path, rather than to the entire program[2]. Our scheme satisfies the security properties required by existing

---

[2] To be more precise, in the notation of Kolesnikov [Kol18], the function encoding $F = (T, E)$ consists of function topology T (thought of as the Boolean circuit) and cryptographic material E (e.g., garbled tables). In our work, the cryptographic material E is proportional to the longest execution path.

For the reader familiar with the BHR notation, we provide the following discussion. In BHR, the function encoding F must (implicitly) include a full description of the function, i.e., it must include a description of each clause. In this sense, F is also proportional to the full size of the function. However, compared to the cryptographic material needed for the longest clause, this function description (which can be thought of as a Boolean circuit $\mathcal{C}$ computing f) is small. Formally, the size of the circuit description is constant in κ. Most importantly, implementations can assume that circuit descriptions are known to both players, and therefore need not transmit them (or treat them separately).

ZK constructions [JKO13, FNO15]. This results in an efficient ZK scheme whose communication is proportional to the longest program execution path.

A verifiable garbling scheme is a tuple of six algorithms:

$$(\mathsf{ev}, \mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De}, \mathsf{Ve})$$

The first five algorithms define a garbling scheme [BHR12], while the sixth adds verifiability [JKO13]. In the ZK context, a garbling scheme can be seen as a specification of the functionality computed by V and P. Loosely speaking, V uses Gb to construct the garbled circuit sent to P. V defines input labels by using En, and decodes the output label received from P by using De. P uses Ev to compute the garbled circuit with encrypted inputs and uses Ve to check that the circuit was honestly constructed. Finally, ev provides a reference against which the other algorithms can be compared. The key idea is that if (1) a garbling is constructed using Gb, (2) the inputs are encoded using En, (3) the encoded output is computed using Ev, and (4) the output is decoded using De, then the resulting bit should be the same as calling ev directly.

A verifiable garbling scheme must satisfy the formal definitions of **correctness**, **soundness**, and **verifiability**. We present these definitions, as well as formal proofs that our scheme satisfies these properties in Sect. 4.

Since we are primarily concerned with reducing the cost of disjoint clauses, we can offload the remaining work (i.e. processing a single clause) to another garbling scheme. Therefore, our scheme is parameterized over another garbling scheme, Base. We place the following requirements on this underlying scheme:

- The scheme must be **correct** and **sound**.
- The scheme must be *projective* [BHR12]. In a projective garbling scheme, each bit of the prover's input is encoded by one of two cryptographic labels. The truth value of that bit is used to decide which label the prover will receive. Projectivity allows us to stack input labels from different clauses. We can lift this requirement by compromising on efficiency: The verifier can send an input encoding for *each* clause rather than a stacked encoding.
- The scheme must output a single cryptographic label and decoding must be based on an equality check of this label. This property is important because it allows us to stack the output labels from each clause. Again, we can lift this requirement by compromising efficiency: The prover can send each output label rather than the stacked value.

These requirements are reasonable and are realized by existing schemes, including state-of-the-art privacy-free half gates [ZRE15].

In the following text, we describe our construction, the PFS verifiable garbling scheme `Stack`. Pseudocode for each of our algorithms is given in Fig. 1.

3.1 Reference Evaluation

ev maps the computed function f and an input x to an output bit. Informally, ev provides a specification that the garbled evaluation can be compared to: The garbled evaluation should yield the same output as running ev. In our setting, the input can be split into a clause selection index t and the remaining input. $\mathsf{Stack.ev}$ delegates to $\mathsf{Base.ev}$ on the t-th clause. For many practical choices of Base (including privacy-free half gates) the procedure $\mathsf{Base.ev}$ simply applies the function to the input: That is, it returns $f(x)$.

3.2 Garble

Gb maps the given function, f, to a garbled function F, an encoding string e, and a decoding string d. At a high level, Gb corresponds to the actions taken by V to construct the proof challenge for P. Typically, e contains input labels (conveyed to P via OT), F contains cryptographic material needed to evaluate the individual logic gates, and in the ZK setting d contains a single label corresponding to a secret that will convince the verifier that the prover has a witness. The objective of the prover is to use her witness to construct d.

Gb is usually described as an algorithm with implicit randomness. However, for the purposes of our scheme it is important that Gb is explicitly parameterized over its randomness. Gb takes as parameters the unary string 1^κ, the desired function f, and a random string, R. It generates a three-tuple of strings, (F, e, d).

At a high level, $\mathsf{Stack.Gb}$ (Fig. 1) delegates to $\mathsf{Base.Gb}$ for each clause and XORs[3] the obtained garbling strings, thus reducing the GC length to that of a single (largest) clause. First, it deconstructs f into its various clauses and extracts from the randomness (1) m different random seeds and (2) the random string PoR which we refer to as the *proof of retrieval*. The proof of retrieval is a security mechanism that allows our approach to cleanly interact with existing MPC protocols. Later, in Sect. 3.3 we will see that the prover receives via OT the garbling seed for each of m clauses, except for the target clause. PoR prevents P from simply taking *all* m seeds and trivially constructing a proof (we enforce that if P takes all seeds, then she will not obtain PoR). Next, each seed is used to garble its respective clause using the underlying scheme ($\mathsf{Stack.Gb}$ line 5). The cryptographic material from each clause is XORed together and concatenated with the function description[4] ($\mathsf{Stack.Gb}$ line 6). This is a key step in our approach: Since the cryptographic material has been XORed together, we have reduced the cost of sending the garbling F compared to sending each garbling separately. Similarly, the output labels from each clause are XORed together. The PoR string is also XORed onto the latter value. Finally, the encoding string e contains PoR, each random string s_i, and each encoding string e_i.

---

[3] As discussed in Sect. 2, by XOR we mean length-aware XOR, where shorter clauses are padded with zeros so that all clauses are bitstrings of the same length.

[4] Including the function description f is a formality to fit the BHR interface. In practice, f is often known to both parties and need not be explicitly handled/transmitted.

3.3 Encode

En maps the encoding string, e, and the function input, x, to an encoded input, X. En describes which input encoding the verifier should send to the prover. Typically, En is implemented by OT.

Stack.En ensures that the prover receives (1) the proof of retrieval string PoR, (2) each random seed $s_{i \neq t}$, and (3) stacked garbled inputs for the target clause. Section 3.2 described how e contains PoR, $s_1..s_m$, and $e_1..e_m$.

First, Stack.En deconstructs e into the above parts. It also deconstructs the circuit input into t (the target clause index) and x_t (the input for the target clause). Next, a vector of secrets, $r_1..r_m$ is constructed. This vector contains PoR and $s_{i \neq t}$. Finally, we use the underlying scheme to construct m encodings of x_t and XOR the encodings together (Stack.En line 10). Stack.En outputs the vector of secrets and the stacked input encodings.

We remark that Stack.En defines the encoding functionality, not an implementation. As mentioned earlier, Stack.En is implemented using OT. Our implementation realizes this functionality in the following way:

- For each clause, V generates n pairs of labels, one pair for each bit and one label for each configuration of that bit.[5]
- V stacks these labels, yielding n pairs of stacked labels.
- For each $i \in 1..m$, V constructs the pair (s_i, PoR).
- Now, P and V participate in $m + n$ executions of 1-out-of-2 OT, such that P receives PoR, non-target seeds, and stacked garbled inputs according to En.

By running this protocol, V obliviously transfers encoded input, including the seeds and PoR, to P.

3.4 Evaluate

Ev maps an encoded function, F, and encoded inputs, X, to the encoded output, Y. In the ZK setting we (as do [JKO13] and [FNO15]) allow Ev to take the unencoded input, x, as a parameter (in practice Ev is run by P who knows the witness). Informally, Ev describes the actions of the prover to construct a proof string, given the garbling of the function and input labels.

The bulk of the work done by Stack.Ev is concerned with 'undoing' the stacking of the encoded functions $F_1..F_m$ and of the encoded inputs $X_1..X_m$, in order to extract the encoded function F_t, and inputs X_t for the target clause. First, Stack.Ev deconstructs all inputs into their constituent parts. It then uses the random strings included in the encoded input to re-garble each non-target clause by calling Base.Gb (Stack.Ev line 7). Note that since Base.Gb is called with the same random strings in both Stack.Ev and Stack.Gb, the resulting encodings are the

---

[5] In fact, since we use half gates we can use the Free XOR extension [KS08]. Therefore, each clause has only one label for each input bit and one global Δ value that separates 0 bit labels from 1 bit labels. Our implementation stacks the Δ from each clause as part of the stacked projective garbling.

same. Stack.Ev cannot call Base.Gb on the target clause because the input encoding does not include the corresponding random string. Instead, r_t is the proof of retrieval PoR. Stack.Ev XORs out the garblings of the non-target clauses to obtain the encoded function (Stack.Ev line 11) and encoded input (Stack.Ev line 12) for the target clause. Now, the prover can use F_t and X_t to compute the output Y_t by calling Base.Ev. Finally, the prover XORs together Y_t, $d_1..d_m$, and PoR and returns the result.

3.5 Decode

De maps an encoded output, Y, and an output encoding string, d, to an unencoded output. In the ZK setting, both Y and d are labels encoding a single bit. Stack.De checks that the values are the same, and if so returns 1 (and 0 if not).

3.6 Verify

Ve maps an input function f, the garbled function F, and the encoding string e to a bit. Informally, the function should return 1 if (F, e) is correctly constructed.

Stack.Ve extracts the proof of retrieval PoR and input seeds $s_1..s_m$ from e. It uses these strings to garble the computed functions and checks that it indeed matches the provided garbling.

In our implementation, we take advantage of an optimization available in Stack.Ve. To verify V's messages, the prover must reconstruct the garbling of each clause. However, the prover *already* garbled each circuit except the target while computing Ev, so we simply reuse these already computed values and only garble the target during verification. This is noteworthy because our approach not only transmits less information, it involves less computation on the part of P as well: Under previously defined ZK garbling schemes (e.g. [ZRE15]), P must both garble and evaluate *every* clause. Under our scheme the prover needs to garble every clause, but need only evaluate the target clause.

3.7 Generalizing to Diverse Clauses

In Sect. 1.4, we simplified the discussion by presenting our approach as handling clauses of the same size and with the same number of inputs. However, our formal presentation does not need these simplifications. Here, we discuss generalization to clauses with different sizes and numbers of inputs.

Our approach supports clauses of various sizes. The only implementation detail that relates to the size of the clauses is the XOR stacking of the garbled material from each clause (Stack.Gb line 6 and Stack.Ev line 11). In Sect. 2, we describe how we use $\oplus$ to denote a *length-aware* variant of XOR (i.e. the shorter string is padded with 0 s). Therefore, there is no correctness concern with stacking mismatched length of material. The only potential concern is security. Our proofs formally alleviate this concern; informally, stacking material is secure because we can safely allow the prover to obtain material for each clause F_i.

Indeed, even sending each clause F_i separately is secure, although inefficient. Giving P access to the garbled material provides no aid in constructing a proof. Specifically, only having a witness and *running* the garbled circuit will allow P to construct the correct Y_t. Therefore, clause stacking does not hinder security.

We support clauses with different numbers of inputs. Regardless of her clause choice t, the prover will append the input string x_t with 0 s until x_t is appropriate for an input of length n. This is secure for a similar reason as having cryptographic material of different lengths. Our technique allows P to learn every input encoding $X_{i \neq t}$ and therefore to learn X_t. This is desirable: We must allow P to learn X_t in order to evaluate the target clause on their input.

4 Proofs of Security

Jawurek et al. [JKO13] introduced a methodology for using garbling schemes to build maliciously secure ZKP protocols. In this section, we prove that our construction satisfies the [JKO13] requirements. Thus, we can directly leverage the work of [JKO13] to construct a maliciously secure ZKP scheme with efficient disjoint clause handling.

[JKO13] requires the garbling scheme to be **correct**, **sound**, and **verifiable**. We use slightly simpler formulations of these definitions presented in [FNO15], a follow-up work on [JKO13].

We now explicitly state the definitions of these properties in our notation. We prove our garbling scheme `Stack` (Fig. 1) satisfies each property (Theorems 1 to 3) if the underlying scheme Base is **correct** and **sound** (We do not require Base to be verifiable, since we explicitly manage the scheme's randomness).

4.1 Correctness

Correctness ensures that the prover can construct a valid proof if she, in fact, has a valid witness. More precisely, Definition 1 states that if a garbling is constructed by calling Gb, then Ev will *always*[6] yield the correct output label, d, when called with the encoding of a valid witness. Recall, we work with explicit randomness. Thus, Gb takes a random string R as an additional input.

Definition 1 (Correctness). *A garbling scheme is **correct** if for all* $n = \mathsf{poly}(\kappa)$, *all functions* $f : \{0,1\}^n \rightarrow \{0,1\}$, *all inputs* $x \in \{0,1\}^n$ *such that* $\mathsf{ev}(f, x) = 1$, *and all random strings* $R \in_R \{0,1\}^\kappa$*:*

$$(F, e, d) = \mathsf{Gb}(1^\kappa, f, R) \Rightarrow \mathsf{Ev}(F, \mathsf{En}(e, x), x) = d$$

Theorem 1. *If the underlying garbling scheme* Base *is correct, then the garbling scheme* `Stack` *(Fig. 1) is correct (Definition 1).*

---

[6] In the full version of this paper, we will discuss *probabilistic* correctness and the changes to our approach that are necessary to account for this probabilistic notion.

Proof. By correctness of the underlying garbling scheme. $\mathsf{Stack.Gb}$ constructs the output label d by XORing together the output label of each clause, d_i, and the proof of retrieval string, PoR. Therefore, it suffices to show that a prover, P, with satisfying input obtains each d_i and PoR. Recall that P's input includes the bits that select a clause, t, concatenated with her remaining input x. We show that she obtains each output label d_i and PoR in three steps:

1. P obtains d_i for all $i \neq t$ by garbling f_i. This is immediate from the fact that P receives every seed s_i for $i \neq t$ as a part of her encoded input ($\mathsf{Stack.En}$, line 6). P garbles clause f_i with seed s_i and obtains d_i ($\mathsf{Stack.Ev}$, line 7).
2. P obtains d_t by evaluating f_t on her input x. We show this in three parts: (1) P obtains the garbling of the selected clause, F_t, (2) P obtains encoded inputs for the selected clause, X_t, and (3) P computes d_t.
   First, $\mathsf{Stack.Gb}$ constructs the XOR sum of the garbling of each clause, F_i ($\mathsf{Stack.Gb}$, line 6). Therefore, to show that P obtains F_t, it suffices to show that she obtains F_i for all $i \neq t$ and F. F is given as a parameter to $\mathsf{Stack.Ev}$ and so is trivially available. P obtains the garblings of all clauses F_i by calling $\mathsf{Stack.Gb}$ with the seeds in her encoded input.
   Second, $\mathsf{Stack.En}$ constructs X by XORing together the encodings of each clause X_i ($\mathsf{Stack.En}$, line 10). Similar to the previous step, P computes each X_i by garbling clause i with s_i. She then uses the encoding e_i to compute $X_i = \mathsf{Base.En}(e_i, x)$ ($\mathsf{Stack.Ev}$, line 8). She XORs these encodings with X to get the appropriate input for clause t, X_t.
   Finally, P computes $Y_t = \mathsf{Base.Ev}(F_t, X_t, x)$. The underlying garbing scheme is correct by assumption. Therefore, $Y_t = d_t$.
3. P obtains PoR. This string is immediately available as r_t ($\mathsf{Stack.En}$ line 8).

P XORs together each of these elements ($\mathsf{Stack.Ev}$ line 14), obtaining the output Y which has the same value as d. That is, $\mathsf{Stack.Ev}(F, \mathsf{Stack.En}(e, x), x) = d$. Therefore, `Stack` is correct. □

4.2 Soundness

Definition 2 (Soundness). *A garbling scheme is* ***sound*** *if for all* $n = \mathsf{poly}(\kappa)$*, all functions* $f : \{0,1\}^n \to \{0,1\}$*, all inputs* $x \in \{0,1\}^n$ *such that* $\mathsf{ev}(f, x) = 0$*, and all probabilistic polynomial time adversaries* $\mathcal{A}$ *the following probability is negligible in* κ*:*

$$\mathit{Pr}(\mathcal{A}(F, \mathit{En}(e, x), x) = d : (F, e, d) \leftarrow \mathit{Gb}(1^\kappa, f))$$

Soundness is a more succinct version of authenticity [BHR12], restricted to the ZK setting. Informally, soundness ensures that a prover who does *not* have a valid witness cannot convince the verifier otherwise. More specifically, we require that no malicious evaluator can extract the garbling scheme's secret d unless she knows an input x such that $f(x) = 1$.

In our garbling scheme, d combines 1-labels of all clauses and the proof of retrieval PoR. We show that an adversarial P who is given $(F, \mathsf{Stack.En}(e, x), x)$, such that $\mathsf{Stack.ev}\left(f, x\right) = 0$, cannot obtain at least one of the components of d and hence cannot output d, except with negligible probability.

Theorem 2. *If the underlying garbling scheme* Base *is sound, then the garbling scheme* `Stack` *(Fig. 1) is sound (Definition 2).*

Proof. By soundness of the underlying garbling scheme. Recall that $d = (\bigoplus d_1..d_m) \oplus \textsc{PoR}$. That is, the output label is the XOR sum of the output labels for each clause and the proof of retrieval. Consider an arbitrary input $(t \mid\mid x_t) \leftarrow x$, such that $\mathsf{Stack.ev}\left(f, x\right) = 0$. We proceed by case analysis on t.

Suppose t is invalid (i.e., $t \notin [1..m]$) and thus $\mathsf{Stack.En}(x)$ outputs all seeds $s_1..s_m$. Then by the definition of $\mathsf{Stack.En}$, $\mathcal{A}$ will *not* receive $\textsc{PoR}$ and hence cannot construct d (except with negligible probability).

Suppose that $t \in [1..m]$, i.e. t is valid. Because $\mathsf{Stack.ev}\left(f, x\right) = 0$, it must be that $\mathsf{Base.ev}\left(f_t, x_t\right) = 0$. Now, $\mathcal{A}$'s input includes the proof of retrieval $\textsc{PoR}$, as well as the seeds for each clause except for clause t. Therefore, an adversary can easily obtain each output label except d_t. We must therefore demonstrate that our scheme prevents an adversary without a witness from successfully constructing d_t, and thereby prevent construction of d. d_t is independent of all values in the scheme except for the values related to the clause itself: s_t, f_t, F_t, X_t, and e_t. By assumption, Base is sound. Therefore, since x_t is not a witness for clause t, the adversary cannot obtain d_t (except with negligible probability), and therefore cannot construct d (except with negligible probability).

Therefore `Stack` is sound. □

4.3 Verifiability

Definition 3 (Verifiability). *A garbling scheme is* ***verifiable*** *if there exists an expected polynomial time algorithm* Ext *such that for all x where $f(x) = 1$, the following probability is negligible in κ:*

$$\mathsf{Pr}\left(\mathsf{Ext}(F, e) \neq \mathsf{Ev}(F, \mathsf{En}(e, x), x) : (F, e, \cdot) \leftarrow \mathcal{A}(1^\kappa, f), \mathsf{Ve}(f, F, e) = 1\right)$$

Informally, verifiability prevents even a malicious verifier from learning the prover's inputs. In the ZK protocol, the prover checks the construction of the garbling via Ve. Verifiability ensures that this check is reliable. That is, it guarantees that if $f(x) = 1$, then the output value $\mathsf{Ev}\left(F, \mathsf{En}\left(e, x\right), x\right)$ is unique and moreover can be efficiently extracted given the encoding. This implies that the verifier has access to the secret d ahead of time. Therefore, V learns nothing by receiving d from the prover, except for the fact that $f(x) = 1$. This holds also for maliciously generated circuits, as long as they pass the verification procedure.

Theorem 3. *If the underlying garbling scheme* Base *is correct, then the garbling scheme* `Stack` *is verifiable (Definition 3).*

Proof. By correctness of `Stack`. Let (F', e') be a garbling of f constructed by $\mathcal{A}$. Let x satisfy $f(x) = 1$. Let Y be the value obtained by evaluating this garbling:

```
Proc $\mathit{Stack.Ext}(F, e)$:
    $(f_1..f_m \mathbin{||} \cdot) \leftarrow F$
    $(\textsc{PoR} \mathbin{||} s_1..s_m \mathbin{||} \cdot) \leftarrow e$
    $(\cdot, \cdot, d) \leftarrow \mathsf{Stack.Gb}(1^\kappa, f_1..f_m, \textsc{PoR} \mathbin{||} s_1..s_m)$
    return $d$
```

Fig. 2. The Stack.Ext algorithm that demonstrates verifiability of `Stack`.

$$Y = \mathsf{Ev}\left(F', \mathsf{En}\left(e', x\right), x\right)$$

Let R be the randomness included in e' (i.e. $R = \textsc{PoR} \mathbin{||} s_1..s_m$). Let (F, e, d) be the result of calling Stack.Gb on this randomness:

$$(F, e, d) = \mathsf{Stack.Gb}\left(1^\kappa, f, R\right)$$

We first claim that Y must be equal to d.

Suppose not, i.e. suppose $Y \neq d$. By correctness (Theorem 1), Ev always returns d; therefore it must be the case that (F', e') is different from (F, e), i.e. either $F' \neq F$ or $e' \neq e$. But if so, Stack.Ve would have returned 0 (Stack.Ve line 4). Verifiability assumes that Stack.Ve returns 1, so we have a contradiction. Therefore $Y = d$.

Now, we must prove that there exists a poly-time extraction algorithm Stack.Ext, which probabilistically extracts the output label from (F', e'). This construction and proof is immediate: Stack.Ext delegates to Stack.Gb. Namely (see Fig. 2 for full description of Stack.Ext), on input (F, e), Stack.Ext parses $(R, \cdot) \leftarrow e'$, runs $(\cdot, \cdot, d) \leftarrow \mathsf{Stack.Gb}(1^\kappa, f, R)$ and outputs d. We have already shown that d constructed this way satisfies $Y = d$.

Therefore `Stack` is verifiable. □

5 Instantiating Our Scheme

We built our implementation on the publicly available EMP-Toolkit [WMK16]. We use privacy-free half gates as the underlying garbling scheme [ZRE15]. That is, XOR gates are free (requiring no cryptographic material or operations) and all AND gates are implemented using fixed-key AES [BHKR13]. Each AND gate costs 1 ciphertext in cryptographic material, 2 AES encryptions to garble, and 1 AES encryption to evaluate. We use security parameter $\kappa = 128$.

We instantiate all [JKO13] ingredients, including committing OT. We use the maliciously-secure OT extension of [ALSZ15] in our implementation both because it is efficient and because an implementation with support for committing OT is available in EMP.

6 Performance Evaluation

Recent advances in non-interactive ZK proofs (NIZK) are astounding. The blockchain use case motivates intense focus on small proof size (as short as

work		**Experiment 1. Fig. 4**		**Experiment 2. [XZZ^{+}19]**	
		time (s)	**comm. (MB)**	**time (s)**	**comm. (MB)**
`Stack` [this work]	LAN	0.395	13.426	4.205	182.2
	sh. LAN	2.473		32.04	
	WAN	3.525		24.52	
[JKO13]	LAN	0.782	31.180	4.205	182.2
	sh. LAN	5.567		32.04	
	WAN	6.208		24.52	
[KKW18]		140	20	840	120
Ligero [AHIV17]		60	0.3	404	1.5
Aurora [BCR^{+}19]		1,000	0.15	3,214	0.174
Bulletproofs [BBB^{+}18]		1,800	0.002	13,900	0.006
STARK [BBHR19]		40	0.5	300	0.6
Libra [XZZ^{+}19]		15	0.03	202	0.051

Fig. 3. Experimental performance of our approach compared to state-of-the-art ZKP systems. **1.** We compare circuit C from Fig. 4 which makes calls to AES, SHA-1 and SHA-256 and has 7,088,734 gates (1,887,628 AND). **2.** We compare based on an experiment from [XZZ+19] where the prover builds a depth 8 Merkle tree from the leaves. The circuit invokes SHA-256 511 times. Resulting timings include prover computation, verifier computation, and all communication. For our and the [JKO13] GC-based approaches we separate timing results for LAN, Shared LAN, and WAN networks. Results for works other than ours and [JKO13] are either approximate interpolations from related works [KKW18, BBHR19] or taken directly from the reporting of [XZZ+19].

several hundred bytes!) and fast verifier computation time. Prover computation time is usually superlinear ($O(|\mathcal{C}| \log |\mathcal{C}|)$ or higher in most schemes, with Libra and Bulletproofs offering linear time) with relatively large constants. As proof circuits grow larger, the high constants and superlinear computational scaling becomes burdensome and GC-based proof systems become more efficient thanks to linear computation scaling with small constants.

We focus our performance comparison on JKO and the fastest NIZK systems, such as [KKW18, BBHR19], Bulletproofs [BBB+18], Ligero [AHIV17], Aurora [BCR+19], and Libra [XZZ+19]. Figure 3 shows that GC-based approaches (`Stack` and JKO) scale better than current NIZKs at the cost of interactivity, and Fig. 5 shows how `Stack` improves on JKO w.r.t. the branching factor.

A reader familiar with recent GC research and related work discussed in Sect. 1.6 may already have a very good sense for the performance of our scheme `Stack`, both in computation and communication. Indeed, `Stack` simply calls privacy-free half gates and XORs the results. Compared to Free IF [Kol18] (a GC protocol using topology-decoupling, not a ZK scheme), our communication is 2× smaller, since we use 1-garbled-row privacy free garbling.

Our and the Baseline Systems. We implemented and ran our scheme `Stack` and [JKO13] instantiated with privacy-free half gates [ZRE15], as the state-of-the-art baseline. Most of the code (except for handling stacking) is shared between the two systems. By comparing the performance of these two protocols, we isolate the effect of stacking garbled material. In addition, we include detailed comparison to performance numbers reported by other state-of-the-art systems [BBB+18, KKW18, AHIV17, BCR+19, BBHR19, XZZ+19] in Sect. 6.2.

Boolean vs Arithmetic/R1CS representations are difficult to compare. Arithmetic operations are costly in Boolean world; program control flow and other operations (e.g., bit operations in hash functions and ciphers) often cannot be done in arithmetic, and a costly bit decomposition is required. Because of this, we focus on the benchmark that emerged as universal in recent literature: SHA-256 evaluations. We use standard SHA-256 Boolean circuits available as part of EMP, and other works use R1CS representations optimized for their work.

System and Experiment Setup. We implemented our and JKO protocols based on EMP [WMK16]. We ran both P and V *single-threaded* on the same machine, a ThinkPad$^{\text{TM}}$ Carbon X1 laptop with an Intel® Core$^{\text{TM}}$ i7-6600U CPU @ 2.60 GHz and 16 GB of RAM. We record the total communication and the total wall clock time. Each experimental result was averaged over 5 runs. We use the Linux command `tc` to simulate three network settings (shared LAN models the setting where LAN is shared with other traffic):

Network Setting	bandwidth (mbps)	latency (ms)
LAN	1000	2
Shared LAN	50	2
WAN	100	100

RAM and CPU Consumption. GC-based ZK proofs can be performed with very low RAM and CPU. This is because GC generation and evaluation is a highly serializable and streamlined process: Gen only needs to keep in RAM the amount of material proportional to the largest cross-section of the GC. Wire labels and garbled gates can be discarded once they no longer appear in future gates. Further, each AND gate garbling requires only 2 AES calls.

In contrast, recent NIZK systems are resource-hungry. They execute their experiments on high-end machines with very high RAM. For example, STARK was run on a powerful server with 32 3.2 GHz AMD cores and 512 GB RAM. In Experiment 2, Libra uses 24.7 GB of RAM while running on 64 GB machine [Zha19].

We execute all our experiments on a standard laptop with 16 GB RAM (of which 146 MB is used in Experiment 1, as reported by Linux `time` command). We *do not* adjust our numbers to account for the hardware differences.

6.1 Experiment 1: Merkle Tree Proof (JKO Comparison Focus)

We first evaluate our approach against prior work using a Merkle tree membership benchmark, discussed in Sect. 1.1. This experiment is designed to show how our scheme compares to JKO. We include comparison to state-of-the-art NIZK as an additional point of reference.

Circuit	**# AND**	**# XOR**	**# INV**
Clause C1: proof w.r.t. tree 1	812936	519699	986677
Clause C2: proof w.r.t. tree 2	546089	2243643	55237
Clause C3: proof w.r.t. tree 3	528601	944039	451828
$\mathbf{C = (C1 \lor C2 \lor C3)}$	1887628	3707381	1493725

Fig. 4. Clause and circuit sizes in our experiment. Clauses are defined in Sect. 6.1.

For the sake of concreteness, we constructed a scenario whereby P's record is certified by inclusion in a Merkle tree whose root is published by an authority. There are several such roots published, and P wishes to hide which root certifies her. P's record, in addition to arbitrary data fields, contains a 128-bit secret key, which P may use as a witness to prove statements about its record.In our experiment, P wishes to prove membership of her record in one of three Merkle trees, as well as properties of her record. We will explain the exact details of this benchmark in the full version of this paper.

The resulting circuit C (cf. Fig. 4) consists of three conditional branches, each clause corresponding to a proof for a specific Merkle tree. The clauses execute various combinations of calls to SHA256, SHA-1 and AES. Total circuit size (i.e. what JKO and other ZK systems would evaluate) is over 7 million gates.

Figure 3 tabulates results and includes the estimated performance of the NIZK systems [BBB+18,KKW18,AHIV17,BCR+19,BBHR19,XZZ+19]. The larger proof statement sizes we consider were not reported in prior works (e.g., [KKW18,BCR+19]); we estimate their performance by considering their asymptotic complexity and extrapolating their reported numbers. The tabulation includes 4 metrics. This experiment explores JKO comparison, and below we discuss metrics w.r.t. JKO. (We discuss at length other NIZKs in Sect. 6.2.)

- **Total communication** (in MB). Our reported communication includes performing commitments, OTs, and sending the circuit garbled material.
  Discussion. Stacking yields a 2.3× improvement over JKO. This is optimal for stacked garbling: total circuit size is 2.3× larger than the largest clause.
- **Total LAN wall clock time** used to complete each protocol in a simulated LAN setting. The simulated LAN has 1 gbps bandwidth and 2 ms latency.
  Discussion. Our approach yields a 2.0× speedup over JKO, due to reduced communication. Our total speedup does not quite match the 2.3× proof size improvement because our computation cost is same as JKO. As 1 gbps is extremely fast, computation takes a noticeable portion of the overall time.
- **Total shared LAN wall clock time** in a setting where LAN is shared with other traffic and approximately 50 Mbps of bandwidth is available.
  Discussion. Our approach yields a 2.25× speedup, close to the optimal 2.3×. In shared LAN the cost of computation becomes less important.
- **Total WAN wall clock time** with 100 mbps bandwidth and 100 ms latency.
  Discussion. Our approach yields a 1.76× speedup. As network latency increases, the number of rounds becomes important. Both [JKO13] and our approach have the same number of rounds, and hence our performance improvement is less pronounced than in the shared LAN setting.

6.2 Experiment 2: Merkle Tree Building (NIZK Comparison Focus)

As discussed above, Boolean/arithmetic/R1CS representations each have their advantages, and their comparison is highly nuanced. SHA-256 evaluation has become an informal standard by which recent NIZKs compare their performance. We use a standard Boolean circuit for SHA-256 that is included with EMP.

Libra [XZZ+19] includes a benchmark where P computes the root of a depth-8 Merkle tree (256 leaves; total 511 SHA-256 evaluations) as part of a proof. When compiled as a Boolean circuit, this benchmark includes ≈60 million gates. Figure 3 includes results for this benchmark; our focus is on the relative efficiency of our approach against Libra and other state-of-the-art NIZKs. Performance numbers for NIZKs were obtained from [XZZ+19], except in the case of [KKW18] and [BBHR19] which were not tabulated by [XZZ+19]. The numbers for these two works were extrapolated based on their reported performance.

Discussion. This experiment does not present an opportunity to take advantage of stacking since there is no conditional branching. Therefore, our approach reduces to [JKO13] equipped with privacy-free half gates. Still, this helps to demonstrate the high concrete efficiency of the GC-based ZKP approach. We (and [JKO13]) are several orders of magnitude faster (over LAN; one or more orders over WAN) in this second benchmark than each reported NIZKs except Libra. We outperform Libra by 6× over WAN and nearly 50× over LAN.

We now present more detailed comparison of Fig. 3 results with the individual NIZK schemes, each of which offers different advantages and trade offs.

- Ligero, Aurora and STARK are NIZK proof systems in the 'interactive oracle proof' paradigm (IOP). Among these three superlinear-runtime works, STARK is most competitive in total runtime due to better constants. Our work outperforms STARK by 10–100×, depending on the network. Our advantage would be higher for cases with branching (cf. Sects. 6.1 and 6.3).
- [KKW18] is linear both in computation and proof size with moderate constants. It may be preferable for smaller-size statements ([KKW18] suggest their scheme can be used as a signature scheme based on AES or LowMC cipher), or for proofs of very large statements due to linear scaling of the prover work. Our work outperforms [KKW18] in the proof time metric because [KKW18] has constants much higher than us: [KKW18] simulates 40-100-player MPC and also repeats the proof multiple times. We are two orders of magnitude faster than [KKW18]. Further, our approach yields smaller proof size in Experiment 1 due to our ability to stack the three clauses.
- Bulletproofs [BBB+18] features linear proof time and staggeringly small proofs, logarithmic in the size of the witness! It has high constants due the use of public key operations. We are 1,000 s of times faster than Bulletproofs.
- Libra [XZZ+19] not only constructs small proofs (with size second only to Bulletproofs amongst the considered works), but also features linear prover time with low-moderate constants. Notably (and unlike all other considered works), Libra requires one time trusted setup, which somewhat limits its applicability. We outperform Libra by 6× over WAN and nearly 50× over LAN. Our advantage will increase as the branching increases.

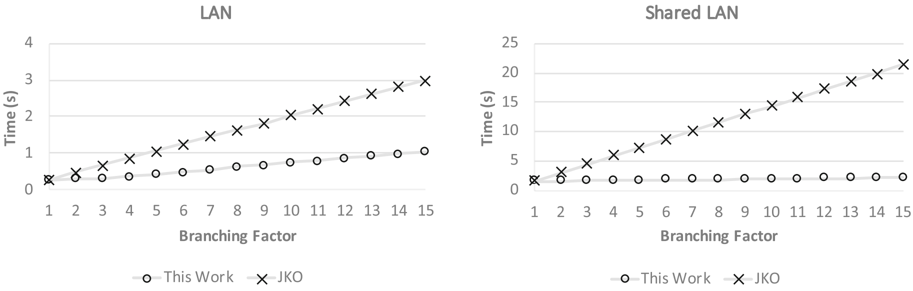

Fig. 5. Plotted results of Experiment 2, evaluating 1-out-of-n randomly generated clauses each of size 500K AND/2M total gates. Each data point plots the total wall clock time needed to perform a proof.

6.3 Experiment 3: Scaling to Many Clauses

We explore how our approach scales in overall proof time as the number of proof disjuncts increases. This metric helps quantify our advantage over [JKO13]. In this experiment, we measure performance of proof statements with different numbers of disjoint clauses and plot total proof times in Fig. 5. To ensure there are no shortcuts in proofs (e.g. exploiting common subcircuits across the branches), we generate all clauses randomly (details will be included in the full version of this paper). Each circuit has 500,000 AND gates and 2 million total gates. We focus on total proof time, and compare our performance to [JKO13].

Discussion. This experiment shows the benefit of reduced communication and its relative cost to computation. In a *single-thread execution* on a LAN, our approach can complete the 1-out-of-15 clause proof (8M AND gates and 30M total gates) in 1 s. This is less than 15× communication improvement over [JKO13] due to relatively high computation cost. As we scale up computation relative to communication (by multi-threading, or, as in our experiment, by consuming only 50 Mbps bandwidth on a shared LAN), our performance relative to [JKO13] increases. In single-threaded execution on shared LAN we are 10× faster than [JKO13] with 15× smaller communication.

7 Proving Existence of Bugs in Program Code

We present a compelling application where our approach is particularly effective: P can demonstrate in ZK the existence of a bug in V's program code. In particular, V can arrange a corpus of C code into various snippets, annotated with assertions. Some assertions, such as array bounds checks and division by zero checks can be automatically inserted. In general, assertions can include arbitrary Boolean statements about the program state. Once the program is annotated, P can demonstrate that she knows an input that causes a program assertion in a snippet to fail. We stress that the instrumentation alone, which can be automated, does not help V to find the bug. P's secret is the snippet ID and input which exercises the error condition caught by an assertion.

```
 1: static const char* SMALL_BOARD = "small_board_v11";
 2: int* alloc_resources(const char* board_type) {
 3:    int block_size;
 4:    // The next line has a bug!!
 5:    if (!strncmp(board_type, SMALL_BOARD, sizeof(SMALL_BOARD))) {
 6:       block_size = 10;
 7:    } else { block_size = 100; }
 8:    return malloc(block_size * sizeof(int));  }
 9: int incr_clock(const char* board_type, int* resources) {
10:    int clock_loc;
11:    if (!strncmp(board_type, SMALL_BOARD, strlen(SMALL_BOARD))) {
12:       clock_loc = 0;
13:    } else { clock_loc = 64 }
14:    (*(resources + clock_loc))++;
15:    return resources[clock_loc];  }
16: void snippet(const char* board_type) {
17:    int* res = alloc_resources(board_type);
18:    incr_clock(board_type, res);  }
```

Fig. 6. An example C snippet that the prover can demonstrate has a bug. Lines 5 and 11 contain inconsistent string comparisons that can cause undefined behavior.

As a simple example, consider the following piece of C code:

```
char example(const char* s) { return s[1]; }
```

Once the program has been instrumented to detect invalid memory dereferences, the prover can submit the input `""` (the empty string) as proof that this program has a bug: The input is empty, but the program attempts to access index 1.

Ours is the best-in-class ZK approach to this application for two reasons:

1. Common programs contain seemingly innocuous constructs, such as pointer dereferences and array accesses, that compile to very large circuits and hence result in very large proof statements. As we have demonstrated, the JKO paradigm, and hence our proof system, is particularly well suited for proving large statements as quickly as possible.
2. Many organizations have truly enormous repositories of code. This is problematic even for fast interactive techniques like JKO because larger code bases require more communication.
   In contrast, our approach remains realistic as the repository grows larger: Communication is constant in the number of snippets (it is proportional to the maximum snippet length). We believe that this advantage opens the possibility of implementing this application in industrial settings.

We include a proof of concept of this use case. Further expanding this is an exciting direction for future work, both in the area of cryptography and of software engineering/compiler design.

experiment	LAN time (s)	WAN time (s)	comm. (MB)	compilation (s)
4 snippets	0.107	2.327	1.542	0.054
1,000 snippets	4.953	6.716	1.600	10.468

Fig. 7. Results for running `Stack` for the bug proving application with 4 and 1,000 snippets. We record LAN and WAN time to complete the proof, total communication, and the time to compile all snippets to Boolean circuits.

At the same time, we can already handle relatively complex code. One of the snippets we implemented (Fig. 6) contains a mistake inspired by a real-world bug in the in MITRE Common Weakness Enumeration (CWE) CWE-467 [cwe19]. This bug is potentially dangerous: For example, MITRE illustrates how it can lead to overly permissive password checking code. We implemented this C code snippet and three others that range between 30 and 50 lines of code.

Consider Fig. 6 Lines 5 and 11. These two lines both perform string comparisons using **`strncmp`**. However, Line 5 incorrectly compares the first n characters where n is the result of the **`sizeof`** call. This call returns the size of a pointer (8 on 64 bit systems) rather than the length of the string. The comparison should have used **`strlen`** in place of **`sizeof`**. An observant prover can notice that a malicious input like `"small_boERROR"` will cause inconsistent behavior that leads to a dereference of unallocated memory.

We instrumented this snippet and three others. Together, these four snippets exercise everyday programming tasks such as user input validation, string parsing, nontrivial memory allocation, and programming against a specification. We will include the source code for all four snippets in the full version of this paper. When compiled to Boolean circuits, these four snippets range between 70,000 and 90,000 AND gates. The number of AND gates is largely determined by the operations performed; e.g. dereferencing memory (array lookup) is expensive while adding integers is cheap. We use the snippets to exercise `Stack` in two experiments:

1 First, we had P demonstrate that she knows a bug in at least 1 out of the 4 snippets. In particular, her input is the string `"small_boERROR"` and triggers an assertion in the code shown in Fig. 6.
2 Second, we simulated a larger code base with 1,000 snippets of 30-50 LOC. Ideally, this code base would contain 1,000 or more *unique* snippets, but since in this work we hand-code instrumentations, this would be an unrealistic effort. We approximate real performance by including multiple copies of each of our four snippets (250 copies each) in the proof disjunction and carefully ensuring that we don't take replication-related shortcuts. P proves the existence of the bug in the first copy of the snippet from Fig. 6.

In both experiments we recorded (1) the total LAN proof time, (2) the total WAN proof time, (3) the total message transmission, and (4) the total time to compile each snippet to a Boolean circuit using the EMP toolkit [WMK16]. The results reflect our expectations and are tabulated in Fig. 7. Note, the $1,000$

snippet experiment is less than 250× slower than the 4 snippet experiment due to constant costs such as setting up a channel and evaluating OTs.

Communication stays nearly constant between the two experiments despite a large increase in the size of the proof challenge. This is a direct result of our contribution of clause stacking. The small change in communication is a result of additional OTs needed for P to select 1 target out of 1,000. Because of the relatively small proof size, both experiments run fast, even on our modest hardware: The 4-snippet proof takes a tenth of a second and the 1,000 snippet proof takes fewer than 5 s. We also ran the same two experiments against [JKO13]. In the 4 snippet experiment, JKO took 0.2211 s on LAN and 3.056 s on WAN, consuming 5.253 MB of communication. The 1,000 snippet experiment crashed our modest hardware as JKO tried to allocate an enormous piece of memory to hold the garblings of the large circuit. Therefore, we tried again with only 500 snippets. Here, JKO took 13.868 s on LAN and 86.356 s on WAN, using 645.9 MB of communication. Again, our approach significantly outperforms [JKO13] due to clause stacking. Performance may already be realistic for some use cases and will likely improve through future work.

Compiling C programs into Boolean circuits is currently the slowest part of our proof. Compilation speed has largely been ignored in prior work; it is unsurprising that the EMP-toolkit is not heavily optimized for it. We believe future work will significantly improve compilation.

7.1 Snippet Instrumentation

We instrument the snippets by extending EMP [WMK16] with pointers (and arrays to facilitate pointers) and implementations of C standard library functions. These features are critical to handling realistic program code and Fig. 6 prominently uses them. We briefly discuss how these features are implemented.

First, we examine pointers and arrays. Our implementation of pointers is greatly simplified, and we leave more general and efficient handling of pointers for future work. In our implementation, a pointer consists of a triple of:

1. A cleartext pointer to an array. This array is allocated to a fixed publicly known size by calls to our instrumentation of **`malloc`**.
2. An encrypted index into the array. Pointer operations (e.g., pointer offset by an integer) operate over this index. Calls to **`malloc`** set this index to 0.
3. An encrypted maximum index. **`malloc`** determines this maximum value based on the size argument.

Pointer dereferences contain an instrumented assertion that checks that the private index is ≥ 0 and is less than the maximum index. It is this assertion that allows the prover to demonstrate Fig. 6 has a bug: The dereference on Line 14 triggers this assertion on particular inputs. After this assertion is checked, the pointer dereference is implemented as a linear scan over the array. For each index of the array, we perform an equality check against the encrypted index. We multiply the output of each equality check by the array entry at that index. Therefore, the result of each multiplication is 0, except for at the target index, where the result is the dereferenced value. We add all multiplication results together using XOR, which returns the dereferenced value.

This pointer handling is limited. For example, we cannot handle a program that conditionally assigns a pointer to one of two different memory locations constructed by different calls to **`malloc`**: Each pointer can only hold one cleartext array pointer. Additionally, it is likely possible to concretely improve over linearly scanning the entire cleartext array.

Second, we discuss C standard library functions. In fact, with the availability of pointers this instrumentation is mostly uninteresting. The implementations are relatively straightforward pieces of C code that we instrument in a manner similar to the snippets. For example, our instrumentation of **`strlen`** takes an instrumented pointer as an argument. It walks the cleartext array of the pointer and increments an encrypted counter until the null character is reached.

Notably, we allow functions to contain loops, but place hard-coded upper bounds on the number of allowed iterations for any loop.

Acknowledgment. This work was supported in part by NSF award #1909769 and by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), via 2019-1902070008. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein. This work was also supported in part by Sandia National Laboratories, a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

[AHIV17] Ames, S., Hazay, C., Ishai, Y., Venkitasubramaniam, M.: Ligero: lightweight sublinear arguments without a trusted setup. In: Bhavani, M.T., Evans, D., Malkin, T., Xu, D. (ed.) ACM CCS 2017, pp. 2087–2104. ACM Press, October/November 2017

[ALSZ15] Asharov, G., Lindell, Y., Schneider, T., Zohner, M.: More efficient oblivious transfer extensions with security for malicious adversaries. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9056, pp. 673–701. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46800-5_26

[BBB+18] Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: short proofs for confidential transactions and more. In: 2018 IEEE Symposium on Security and Privacy, pp. 315–334. IEEE Computer Society Press, May 2018

[BBHR18] Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046 (2018). https://eprint.iacr.org/2018/046

[BBHR19] Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable zero knowledge with no trusted setup. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11694, pp. 701–732. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26954-8_23

[BCG+13] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: verifying program executions succinctly and in zero knowledge. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8043, pp. 90–108. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40084-1_6

[BCR+19] Ben-Sasson, E., Chiesa, A., Riabzev, M., Spooner, N., Virza, M., Ward, N.P.: Aurora: transparent succinct arguments for R1CS. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11476, pp. 103–128. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17653-2_4

[BG93] Bellare, M., Goldreich, O.: On defining proofs of knowledge. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 390–420. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-48071-4_28

[BHKR13] Bellare, M., Hoang, V.T., Keelveedhi, S., Rogaway, P.: Efficient garbling from a fixed-key blockcipher. In: 2013 IEEE Symposium on Security and Privacy, pp. 478–492. IEEE Computer Society Press, May 2013

[BHR12] Bellare, M., Hoang, V.T., Rogaway, P.: Foundations of garbled circuits. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) ACM CCS 2012, pp. 784–796. ACM Press, October 2012

[CDG+17] Chase, M., et al.: Post-quantum zero-knowledge and signatures from symmetric-key primitives. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017, pp. 1825–1842. ACM Press, October/November 2017

[CDS94] Cramer, R., Damgαrd, I., Schoenmakers, B.: Proofs of partial knowledge and simplified design of witness hiding protocols. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 174–187. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-48658-5_19

[CFH+15] Costello, C., et al.: Geppetto: versatile verifiable computation. In: 2015 IEEE Symposium on Security and Privacy, pp. 253–270. IEEE Computer Society Press, May 2015

[CGKS95] Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. In: 36th FOCS, pp. 41–50. IEEE Computer Society Press, October 1995

[CPS+16] Ciampi, M., Persiano, G., Scafuro, A., Siniscalchi, L., Visconti, I.: Improved OR-composition of sigma-protocols. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016. LNCS, vol. 9563, pp. 112–141. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49099-0_5

[cwe19] Common weakness enumeration (2019). https://cwe.mitre.org/

[Dam10] Damgαrd, I.: On Σ-protocols (2010). http://www.cs.au.dk/~ivan/Sigma.pdf. Accessed 11 May 2019

[DP92] De Santis, A., Persiano, G.: Zero-knowledge proofs of knowledge without interaction (extended abstract). In: 33rd FOCS, pp. 427–436. IEEE Computer Society Press, October 1992

[FNO15] Frederiksen, T.K., Nielsen, J.B., Orlandi, C.: Privacy-free garbled circuits with applications to efficient zero-knowledge. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 191–219. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_7

[GGPR13] Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic Span programs and succinct NIZKs without PCPs. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 626–645. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_37

[GK14] Groth, J., Kohlweiss, M.: One-out-of-many proofs: or how to leak a secret and spend a coin. Cryptology ePrint Archive, Report 2014/764 (2014). http://eprint.iacr.org/2014/764

[GMO16] Giacomelli, I., Madsen, J., Orlandi, C.: ZKBoo: faster zero-knowledge for Boolean circuits. In: Holz, T., Savage, S. (eds.) USENIX Security 2016, pp. 1069–1083. USENIX Association, August 2016

[GMR85] Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof-systems (extended abstract). In: 17th ACM STOC, pp. 291–304. ACM Press, May 1985

[GMW91] Goldreich, O., Micali, S., Wigderson, A.: Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. J. ACM **38**(3), 690–728 (1991)

[Gro16] Groth, J.: On the size of pairing-based non-interactive arguments. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 305–326. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_11

[IKOS07] Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge from secure multiparty computation. In: Johnson, D.S., Feige, U. (eds.) 39th ACM STOC, pp. 21–30. ACM Press, June 2007

[JKO13] Jawurek, M., Kerschbaum, F., Orlandi, C.: Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In: Sadeghi, A.-R., Gligor, V.D., Yung, M., (eds.) ACM CCS 2013, pp. 955–966. ACM Press, November 2013

[Kil92] Kilian, J.: A note on efficient zero-knowledge proofs and arguments (extended abstract). In: 24th ACM STOC, pp. 723–732. ACM Press, May 1992

[KKW18] Katz, J., Kolesnikov, V., Wang, X.: Improved non-interactive zero knowledge with applications to post-quantum signatures. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018, pp. 525–537. ACM Press, October 2018

[Kol18] Kolesnikov, V.: Free `IF`: how to omit inactive branches and implement $\mathcal{S}$-universal garbled circuit (almost) for free. In: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018. LNCS, vol. 11274, pp. 34–58. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03332-3_2

[KS06] Kiraz, M.S., Schoenmakers, B.: A protocol issue for the malicious case of Yao's garbled circuit construction. In: Proceedings of 27th Symposium on Information Theory in the Benelux, pp. 283–290 (2006)

[KS08] Kolesnikov, V., Schneider, T.: Improved garbled circuit: free XOR gates and applications. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008. LNCS, vol. 5126, pp. 486–498. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70583-3_40

[LO13] Lu, S., Ostrovsky, R.: How to garble RAM programs? In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 719–734. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_42

[Mic94] Micali, S.: CS proofs (extended abstracts). In: 35th FOCS, pp. 436–453. IEEE Computer Society Press, November 1994

[PHGR13] Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: nearly practical verifiable computation. In: 2013 IEEE Symposium on Security and Privacy, pp. 238–252. IEEE Computer Society Press, May 2013

[RTZ16] Ranellucci, S., Tapp, A., Zakarias, R.: Efficient generic zero-knowledge proofs from commitments (extended abstract). In: Nascimento, A.C.A., Barreto, P. (eds.) ICITS 2016. LNCS, vol. 10015, pp. 190–212. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49175-2_10

[Sch90] Schnorr, C.-P.: Efficient identification and signatures for smart cards. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 239–252. Springer, New York (1990). https://doi.org/10.1007/0-387-34805-0_22

[WMK16] Wang, X., Malozemoff, A.J., Katz, J.: EMP-toolkit: efficient multiparty computation toolkit (2016). https://github.com/emp-toolkit

[WTs+18] Wahby, R.S., Tzialla, I., Shelat, A., Thaler, J., Walfish, M.: Doubly-efficient zkSNARKs without trusted setup. In: 2018 IEEE Symposium on Security and Privacy, pp. 926–943. IEEE Computer Society Press, May 2018

[XZZ+19] Xie, T., Zhang, J., Zhang, Y., Papamanthou, C., Song, D.: Libra: succinct zero-knowledge proofs with optimal prover computation. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11694, pp. 733–764. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26954-8_24

[ZCD+17] Zaverucha, G., et al.: Picnic. Technical report, National Institute of Standards and Technology (2017). https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions

[Zha19] Zhang, Y.: Personal communication (2019)

[ZRE15] Zahur, S., Rosulek, M., Evans, D.: Two halves make a whole. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 220–250. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_8