A SUPPLEMENTAL EXAMPLES

A.1 *Prelude* and *Overture* Protocol Examples

Example A.1 (Additive Secret Sharing). Additive secret sharing is an MPC protocol where k parties split up their secrets into *shares*, with the property that all k shares are required to reconstruct the secret and any fewer reveals nothing. Additionally, this sharing enjoys additive homomorphism. In the binary field it is well-known that we can extend our use of `xor` and flips to generate an arbitrary number of shares with additive homomorphism.

For example, to generate shares for three-party additive sharing, clients 1,2, and 3 can break up and distribute their shares to each other as follows:

```
v[2,"s1"] := flip[1,"1"] xor flip[1,"s1"] xor s[1,"1"];
v[3,"s1"] := flip[1,"1"];
v[1,"s2"] := flip[2,"1"] xor flip[2,"s2"] xor s[2,"1"];
v[3,"s2"] := flip[2,"1"];
v[1,"s3"] := flip[3,"1"] xor flip[3,"s3"] xor s[3,"1"];
v[2,"s3"] := flip[3,"1"]
```

Then, assuming that party 0 is a "public" client, each party sums its own shares, and then the sum of the sum of shares is revealed in the output view `v[0,"output"]`. By additive homomorphism of `xor`, this output is the sum of the 3 secrets.

```
v[0,"ss1"] := v[1,"s2"] xor v[1,"s3"] xor flip[1,"s1"];
v[0,"ss2"] := v[2,"s1"] xor v[2,"s3"] xor flip[2,"s2"];
v[0,"ss3"] := v[3,"s1"] xor v[3,"s2"] xor flip[3,"s3"];
v[0,"output"] := v[0,"ss1"] xor v[0,"ss2"] xor v[0,"ss3"]
```

We note that, as for MPC protocols generally, the notion of "secrecy" here is subtle and not the same as for encryption as in Example 2.2, particularly when the threat model allows protocol participants to be corrupt. For example, suppose that $1 \in C$, and `s[1,"1"]` is 1, and after running the protocol suppose that `v[0,"output"]` is 0. Then the adversary knows that 2 and 3's secrets are either 1 and 0, or 0 and 1, with 50/50 probability, and definitely neither 0 and 0, nor 1 and 1.

Buliding on Example A.1, we can define a function `share3` that abstracts the process of splitting a given client's secret into 3 separate shares.

Example A.2. The function `share3` splits a client's secret into 3 shares returned as a record with fields `s1-3`:

```
share3(client, secretid) {
  let s1 = flip[client, "share1"] in
  let s2 = flip[client, "share2"] in
  let s3 = (s1 xor s2) xor s[client, "s:" || secretid] in
  {s1 = s1;s2 = s2;s3 = s3}
}
```

Here is a *Prelude* program that uses this function definition:

```
let shares = share3(1, "mysecret") in
v[2,s1] := shares.s2;
v[3,s1] := shares.s3
```

which generates the following *Overture* program, as we formalize in Example A.3 below:

```
v[2,s1] := flip[1, "share2"];
v[3,s1] := flip[1, "share1"] xor flip[1, "share2"] xor s[1, "s:mysecret"]
```

Example A.3. Let $C, e_{A.2}$ be the *Prelude* program and let $\pi_{A.2}$ be the *Overture* program defined in Example A.2. We refer to the latter as "accumulated" by evaluation of the former in the sense that $\langle \varnothing, e_{A.2} \rangle \rightarrow^* \langle \pi_{A.2}, () \rangle$.

Example A.4. Given share3 as defined in Example A.2 and annotation:

$$\mathcal{A}(\texttt{share3}) = \texttt{cid(client)} * \texttt{string(sid)}$$

the type of share3 is $\mathcal{A}(\texttt{share3}) \longrightarrow \tau, \varnothing$ where τ is:

```
{ s1 : bool[client]; s2 : bool[client]; s3 : bool[client] }
```

A.2 Program Distribution Examples

In the following, we will refer to program examples defined in Section 2.3 as π_x, where x is the Example number.

Example 2.2 revisited. In this example, from the adversarial view, s[1,"0"] is 1 or 0 with equal probability given any value of v[0,"0"]. More precisely, for any β_1, β_2 we have:

$$(\mathrm{PD}(\pi_{2.2}))(\{\texttt{s[1,"0"]} \mapsto \beta_1\} | \{\texttt{v[0,"0"]} \mapsto \beta_2\}) = .5$$

Note that this corresponds to *perfect secrecy* as described in [6].

Example A.1 revisited. This standard example of an MPC secure protocol illustrates how a functionality- in this case addition (in the binary field)- can be implemented securely. As mentioned previously, it is *not* the case that *no* information about input secrets can be revealed to the adversary in the MPC threat model, since the result of the functionality can itself reveal information. For example, if the result of addition is 0, the adversary knows that two of the clients' secrets must be 0 and the third's must be 1, though not whose is whose- so we can observe:

$$(\mathrm{PD}(\pi_{A.1}))(\{\texttt{s[1,"1"]} \mapsto 0, \texttt{s[2,"1"]} \mapsto 1, \texttt{s[3,"1"]} \mapsto 0\} | \{\texttt{v[0,"output"]} \mapsto 0\}) = .33$$

Further, since the standard MPC threat model allows participating clients to be corrupted we can formally consider the impacts on information released to the adversary by conditioning the program distribution on corrupt secrets. So if we assume client 2 with input secret 1 is corrupted, and we assume the output is 0, then the adversary knows with certainty that clients 1 and 3 both have input secrets 0. We can express this as:

$$\mathrm{PD}(\pi_{A.1}))(\{\texttt{s[1,"1"]} \mapsto 0, \texttt{s[3,"1"]} \mapsto 0\} | \{\texttt{s[2,"1"]} \mapsto 1, \texttt{v[0,"output"]} \mapsto 0\}) = 1$$

A critical point to note is that this information revealed to the adversary does not violate MPC security- under the assumption of public release of the functionality result, and under the assumption of the possible corruption of clients, this is the cost of doing business. Rather, the concern is that the protocol does not release any additional information through information communicated to views of corrupted parties. Additive secret sharing as realized in $\pi_{A.1}$ is MPC secure in this sense- so for example, for any β_1 and β_2 we have:

$$\begin{array}{c} \mathrm{PD}(\pi_{A.1}))(\{\texttt{s[1,"1"]} \mapsto 0\} | \{\texttt{s[2,"1"]} \mapsto 0, \texttt{v[0,"output"]} \mapsto 0\}) = \\ \mathrm{PD}(\pi_{A.1}))(\{\texttt{s[1,"1"]} \mapsto 0\} | \{\texttt{v[2,"s1"]} \mapsto \beta_1, \texttt{v[2,"s3"]} \mapsto \beta_2, \\ \qquad \texttt{s[2,"1"]} \mapsto 0, \texttt{v[0,"output"]} \mapsto 0\}) = \\ .5 \end{array}$$

B YGC CODE IN DETAIL

In Figures 8 and 9 we define a codebase for Yao's garbled circuits (YGC). This definition follows the *point-and-permute* method described in [13] and elsewhere, to which the reader is referred for more in-depth discussion. In this implementation client 2 is the *garbler* and client 1 is the *evaluator*. The garbler builds the garbled tables and shares them with the evaluator, who then evaluates the gate in an oblivious fashion until the final public output is generated through decryption. Our

```
keygen(gid, b1, b2) { select4(b1,b2,H[gid || "1"],H[gid || "2"],H[gid || "3"],H[gid || "4"]) }

keysgen(gid, b1, b2)
{
  let k11 = keygen(gid,b1,b2) in
  let k10 = keygen(gid,b1,not b2) in
  let k01 = keygen(gid,not b1,b2) in
  let k00 = keygen(gid,not b1,not b2) in
  {k11 = k11; k10 = k10; k01 = k01; k00 = k00}
}

andtable(keys, bt, ap, bp)
{
  let r11 = (keys.k11 xor bt) in
  let r10 = (keys.k10 xor (not bt)) in
  let r01 = (keys.k01 xor (not bt)) in
  let r00 = (keys.k00 xor (not bt)) in
  permute4(ap,bp,r11,r10,r01,r00)
}

sharetable(gid, tid, table)
{
  v[1, "gate:" || gid || tid || "1"] := table.v1;
  v[1, "gate:" || gid || tid || "2"] := table.v2;
  v[1, "gate:" || gid || tid || "3"] := table.v3;
  v[1, "gate:" || gid || tid || "4"] := table.v4
}

owl(gid) {  { k = flip[2,"gate:" || gid || ".k"]; p = flip[2,"gate:" || gid || ".p"] }  }
```

Fig. 8. Yao's Garbled Circuits, selected auxiliary functions.

certification method leverages this definition of compositional units comprising complete related subprotocols– i.e., both the garbler's construction of the gates and the evaluator's evaluation of them. However, we note that our formal results are applicable to the YGC style where the garbler shares the entire circuit prior to evaluation. Our implementation is well-typed, with input type annotations for top-level functions listed in Figure 10.

The pointer bits in wire labels are used to select permuted rows in table garblings. The key bits are used to identify a unique key for table row in each garbled gate. Intuitively, if β_1 and β_2 are either key or pointer bits encoding 1 on two input wire labels to a binary gate, rows and keys in the gate are enumerated in the order:

$$\neg\beta_1\neg\beta_2,\ \neg\beta_1\beta_2,\ \beta_1\neg\beta_2,\ \beta_1\beta_2$$

C REWRITING *Overture* TO STRATIFIED DATALOG

Here we show how to rewrite any π to an equivalent Datalog program, which supports application of recent work in linear algebraic interpretation of Datalog and optimizations of model computation on high performance computers [38]. The method here also enumerates *runs*(π) memory-by-memory, rather than in a "batched" manner as in our first method, allowing parallelization of model computation. The rewriting we describe here is to Datalog with negation, with a negation-as-failure model, though we can also use techniques in [38] to eliminate negation from resulting programs. *Atoms* are *Overture* variables, *literals* are atoms or negated atoms, and clause bodies are

```
garbledecode(gid)
{
  let wl = owl(gid) in
  let r1 = wl.k xor true in
  let r0 = (not wl.k) xor false in
  v[1,"OUTtt1"] := select(wl.p,r1,r0);
  v[1,"OUTtt2"] := select(not wl.p,r1,r0)
}

evaldecode(wv) { wv.k xor select(wv.p,v[1,"OUTtt1"],v[1,"OUTtt2"]) }

evalgate(gid, wva, wvb)
{
  let k = keygen(gid,wva.k,wvb.k) in
  let g = "gate:" || gid in
  let ct = select4(wva.p,wvb.p,
              v[1,g || "tt1"],v[1,g || "tt2"],v[1,g || "tt3"],v[1,g || "tt4"]) in
  let cp = select4(wva.p,wvb.p,
              v[1,g || "pt1"],v[1,g || "pt2"],v[1,g || "pt3"],v[1,g || "pt4"]) in
  { k = k xor ct; p = k xor cp }
}

andgate(gid, ga, gb)
{
  let wla = owl(ga) in
  let wlb = owl(gb) in
  let wlc = owl(gid) in
  let keys = keysgen(gid,wla.k,wlb.k) in
  sharetable(gid,tt,andtable(keys,wlc.k,wla.p,wlb.p));
  sharetable(gid,pt,andtable(keys,wlc.p,wla.p,wlb.p))
}

encode(sa, sb)
{
  let owl1 = owl(sa) in
  let owl2 = owl(sb) in
  v[1,"gate:" || sa || "1.k"] := OT(s[1,sa],owl1.k,(not owl1.k));
  v[1,"gate:" || sa || "1.p"] := OT(s[1,sa],owl1.p,(not owl1.p));
  v[1,"gate:" || sb || "2.k"] := select(s[2,sb],owl2.k,(not owl2.k));
  v[1,"gate:" || sb || "2.p"] := select(s[2,sb],owl2.p,(not owl2.p));
  { wv1 = { k = v[1,"gate:" || sa || "1.k"]; p = v[1,"gate:" || sa || "1.p"] };
    wv2 = { k = v[1,"gate:" || sb || "2.k"]; p = v[1,"gate:" || sb || "2.p"] } }
}
```

Fig. 9. Yao's Garbled Circuits, garbled gates and evaluation code.

conjunctions of literals. A *Datalog program* is a list of clauses.

$$\begin{array}{rcll} \alpha & ::= & \mathtt{v}[\iota, w] \mid \mathtt{s}[\iota, w] \mid \mathtt{flip}[\iota, w] \mid \mathtt{H}[w] & \textit{(atoms)} \\ \textit{body} & ::= & \alpha \mid \neg\alpha \mid \alpha \wedge \textit{body} \mid \neg\alpha \wedge \textit{body} \mid \varnothing & \\ \textit{clause} & ::= & \alpha \leftarrow \textit{body} & \end{array}$$

The first step in converting a protocol ρ to a Datalog program is to apply *models* "locally" to each view definition in π, obtaining constraints on memories that satisfy each view in isolation.

```
garbledecode : string(gid)

evaldecode   : { k = bool[1]; p = bool[1] }

evalgate     : string(gid) *  { k = bool[1]; p = bool[1] } * { k = bool[1]; p = bool[1] }

andgate      : string(gid) * string(ga) * string(gb)

encode       : string(sa) * string(sb)
```

Fig. 10. Yao's Garbled Circuits, selected gate library type annotations.

Lemma C.1. *Let $vars(\varepsilon)$ be the variables in ε, and define:*

$$vtt(\mathsf{v}[\iota, w] := \varepsilon) \triangleq (\mathsf{v}[\iota, w], (models\ (mems(vars\ \varepsilon))\ \varepsilon))$$

Then $vtt(\mathsf{v}[\iota, w] := \varepsilon) = (\mathsf{v}[\iota, w], M)$ where $M = \{m \in mems(vars\ \varepsilon) \mid [\![m, \varepsilon]\!]_\iota = 1\}$ for some ι.

Given this definition, the mapping of vtt across a program π- i.e., $(map\ vtt\ \pi)$- essentially defines the logic program for "view atoms" modulo syntactic conversion. We can accomplish the latter as follows, where $datalog(\pi)$ defines the full conversion.

Definition C.2. We define the conversion from memories to literals and clause bodies as follows:

$$\lfloor \alpha \mapsto 1 \rfloor = \alpha \qquad \lfloor \alpha \mapsto 0 \rfloor = \neg\alpha \qquad \lfloor \{\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n\} \rfloor = \lfloor \alpha_1 \mapsto \beta_1 \rfloor \wedge \cdots \wedge \lfloor \alpha_n \mapsto \beta_n \rfloor$$

Given pairs (α, M) in the range of vtt, we define the conversion to clauses as follows:

$$clauses(\alpha, \{m_1, ..., m_n\}) = \alpha \leftarrow \lfloor m_1 \rfloor \vee \cdots \vee \alpha \leftarrow \lfloor m_n \rfloor$$

The *Overture*-to-Datalog conversion is then defined as:

$$datalog(\pi) \triangleq map\ clauses\ (map\ vtt\ \pi)$$

In addition to converting view definitions to logic clauses, we also need to convert secrets and random tapes. Since we assume given values for these in an arbitrary run of the program, we can capture these as a "fact base" in our program, where a fact is a clause of the form $\alpha \leftarrow \varnothing$ and means that α is true in any model of the program.

Definition C.3. Given m, let $\{\alpha_1, \ldots, \alpha_n\} = \{\alpha \in \mathrm{dom}(m) \mid m(\alpha) = 1\}$. Then define:

$$facts(m) = \alpha_1 \leftarrow \varnothing, \ldots, \alpha_n \leftarrow \varnothing$$

For any safe π with $iovars(\pi) = S \cup V$ and $flips(\pi) = F$, and $m \in mems(S \cup F)$, it is the case that $(facts(m), datalog(\pi))$ is a *normal*, *stratified* (in fact, non-recursive) Datalog program, and so has a unique Least Herbrand Model and is thus amenable to HPC optimization techniques [5]. To compute $runs(\pi)$, and thus $\mathrm{PD}(\pi)$, we compute the Least Herbrand Model m of $(facts(m'), datalog(\pi))$ for all $m' \in mems(S \cup F)$, observing that model computation for each m' can be done in parallel. The following establishes correctness of this approach.

Lemma C.4. *For all π with $iovars(\pi) = S \cup V$ and $flips(\pi) = F$, $datalog(\pi)$ is a* normal, stratified *program [5], and m is the unique Least Herbrand Model of $(facts(m_{S \cup F}), datalog(\pi))$ iff $m \in runs(\pi)$.*

A full empirical exploration of the application of HPC optimizations is beyond the scope of this paper but is a compelling topic for future work. The reader is referred to [30] for empirical results of HPC optimizations in other logic programming contexts.

D SUPPLEMENTARY PROOFS

Theorem 5.8. *Assume given pre-imageable $\mathcal{F}$ and a protocol π that correctly implements $\mathcal{F}$. If NIMO(π) then π is passive secure.*

Proof. Let H and C be an arbitrary partition of the federation and suppose π satisfies noninterference modulo output. Let $iovars(\pi) = (V, S)$ and $flips(\pi) = F$ with output view $out \in V$ and let m be an arbitrary element of $mems(S \cup F)$. Then the distribution of adversarial views in the real world is, by definition, $\mathrm{PD}(\pi)_{(V_C|m_S)}$.

The simulator is given both m_{S_C} and $\mathcal{F}(m_S)$. The simulation $\mathrm{Sim}_C(m_{S_C}, \mathcal{F}(m_S))$ can be defined as follows. First, some $m' \in K_i(\mathcal{F}, \mathcal{F}(m))$ is randomly chosen such that $m' =_C m_S$, as is a random tape $m'' \in mems(F)$[1]. Then, the run of $(m' \cup m'', \pi)$ is evaluated in simulation, yielding $(m^{\mathrm{Sim}}, \varnothing)$. The simulation returns $m^{\mathrm{Sim}}_{V_C}$ as a result.

Now, since the random tape is selected in simulation from the same distribution that we assume for the real world, after selection of m' the probability that any particular $m^{\mathrm{Sim}}_{V_C}$ is returned is by definition $\mathrm{PD}(\pi)(m^{\mathrm{Sim}}_{V_C}|m')$. Furthermore, since we assume that π correctly implements $\mathcal{F}$, this means $m_S =_C m' \wedge \mathrm{PD}(\pi)_{(\{out\}|m_S)} = \mathrm{PD}(\pi)_{(\{out\}|m')}$, so by the definition of noninterference modulo output, any choice of m' yields the same distribution $\mathrm{PD}(\pi)_{(V_C|m_S)}$. Therefore by definition $\mathrm{PD}(\mathrm{Sim}_C(m_{S_C}, \mathcal{F}(m))) = \mathrm{PD}(\pi)_{(V_C|m_S)}$. □

We observe the following properties of separation which are borrowed from prior work [6] and which will be used frequently in proofs.

Lemma D.1. *The following properties hold:*
(1) $P \vdash Y * Z$ iff $P \vdash Z * Y$
(2) $P \vdash x \sim y$ if $P \vdash y \sim x$
(3) $P \vdash x \sim y$ and $P \vdash y \sim z$ imply $P \vdash x \sim z$
(4) $P \vdash X * (Y \cup Z)$ implies ($P \vdash X * Y$ and $P \vdash X * Z$)
(5) ($P \vdash X * Y$ and $P \vdash (X \cup Y) * Z$) implies $P \vdash X * (Y \cup Z)$

The following property also follows by results in [6] and will be useful to make constructions that demonstrate variable separation.

Lemma D.2. $\mathrm{BD}(\pi) \vdash X * Y$ *iff for all $m^1, m^2 \in runs(\pi)$ there exists $m \in runs(\pi)$ with $m^1_X \cap m^2_Y \subseteq m$.*

Lemma D.3. *Given ε_1, ε_2, and i where $vars(\varepsilon_1) \cap vars(\varepsilon_2) = \varnothing$ and $[\![m_1, \varepsilon_1]\!]_i = \beta$ with $m_1(x) = [\![m_2, \varepsilon_2]\!]_i$ and* $\mathrm{dom}(m_1) = vars(\varepsilon_1)$ *and* $\mathrm{dom}(m_2) = vars(\varepsilon_2)$. *Then $[\![m_1 \cap m_2, \varepsilon_1[\varepsilon_2/x]]\!]_i = \beta$.*

Proof. By straightforward structural induction on ε. □

Lemma 6.12 (Substitution∗). *If* $\mathrm{BD}(\pi_2) \vdash \{f\} * X$ *and $\pi_1; \pi_2[\varepsilon/f]$ is safe with $vars(\pi_1, \varepsilon) \cap X = \varnothing$ then* $\mathrm{BD}(\pi_1; \pi_2[\varepsilon/f]) \vdash vars(\varepsilon) * X$.

Proof. Suppose on the contrary it was not the case that $\mathrm{BD}(\pi_1; \pi_2[\varepsilon/f]) \vdash vars(\varepsilon) * X$. Then by Lemma D.2 there exists $m_1, m_2 \in runs(\pi_1; \pi_2[\varepsilon/f])$ with no $m \in runs(\pi_1; \pi_2[\varepsilon/f])$ such that $m^1_{vars(\varepsilon)} \cap m^2_X \subseteq m$. But also by assumption and Lemma D.2 for any β there exists $m' \in runs(\pi_2)$ with $\{f \mapsto \beta\} \cap m^2_X \subseteq m'$. So in particular, we have $\{f \mapsto [\![m^1_{vars(\varepsilon)}, \varepsilon]\!]_i\} \cap m^2_X \subseteq m'$ for any i. This, the assumption $vars(\pi_1, \varepsilon) \cap vars(\pi_2) = \varnothing$, and application of Lemma D.3 leads to the consequence that there exists $m \in runs(\pi_1; \pi_2[\varepsilon/f])$ with $m \supseteq m^1_{vars(\varepsilon)} \cap m^2_X$ given the assumption $vars(\pi_1, \varepsilon) \cap vars(\pi_2) = \varnothing$, which is a contradiction. □

[1]The real/ideal model allows consultation of a "Random Oracle".

Lemma 7.5 (GMW Encode Invariant). *Given well-typed $\pi_1; E[encode(s_1, s_2)]$ for certified encode, let:*

$$\langle \pi_1, E[encode(s_1,s_2)]\rangle \longrightarrow^* \langle \pi_1;\pi_2, E[v]\rangle$$

If we have the following preconditions, where $iovars(\pi_1) = S' \cup V$:

i. $\mathrm{PD}(\pi_1) \vdash S_{\{2\}} * V_{\{1\}}$
ii. $\mathrm{PD}(\pi_1) \vdash S_{\{1\}} * V_{\{2\}}$

then we have as a postconditions, where $iovars(\pi_1;\pi_2) = S' \cup V'$:

i. $\mathrm{PD}(\pi_1;\pi_2) \vdash S'_{\{2\}} * V'_{\{1\}}$
ii. $\mathrm{PD}(\pi_1;\pi_2) \vdash S'_{\{1\}} * V'_{\{2\}}$

Proof. By assumptions of well-typedness we have for some `"s1"` and `"s2"`:

$$\begin{array}{rcl} v & = & \texttt{\{shares1 = }v_1\texttt{; shares2 = }v_2\texttt{\}} \\ v_1 & = & \texttt{\{ c1 = v[1,"s1out"]; c2 = v[2,"s1out"] \}} \\ v_2 & = & \texttt{\{ c1 = v[1,"s2out"]; c2 = v[2,"s2out"] \}} \end{array}$$

And by Definition 7.2 we have:

$$\mathrm{BD}(\pi) \vdash \texttt{\{s[1,"s1"]\}} * \texttt{\{v[2,"s1out"],v[2,"s2out"]\}}$$

$$\mathrm{BD}(\pi) \vdash \texttt{\{s[2,"s2"]\}} * \texttt{\{v[1,"s1out"],v[1,"s2out"]\}}$$

Since we assume that secrets are in uniform and independent marginal distributions a priori, and $vars(\pi_1) \cap vars(\pi_2) = \varnothing$ by condition (iii) of Definition 7.2 and assumptions of well-typedness, the result follows by preconditions and Lemma D.1. □

Lemma 7.12 (YGC Gate Invariant). *Given well-typed $\pi_1; E[gate(g, g_1, g_2, v_1, v_2)]$ and certified gate where:*

$$\langle \pi_1, E[gate(g,g_1,g_2,v_1,v_2)]\rangle \longrightarrow^* \langle \pi_1;\pi_2, E[v]\rangle$$

If the following preconditions hold where $iovars(\pi_1) = S \cup V$:

(1) $\{g_1\} \cap \{g_2\} \cap wired(\pi_1) = \varnothing$ *and* $g \notin \pi_1$
(2) $goc(\pi_1, v_1, g_1)$ *and* $goc(\pi_1, v_2, g_2)$
(3) $\mathrm{PD}(\pi_1) \vdash S_{\{1\}} * V_{\{2\}}$
(4) $\mathrm{PD}(\pi_1) \vdash S_{\{2\}} * V_{\{1\}}$

then we have as postconditions where $iovars(\pi_1) = S \cup V'$:

(1) $goc(\pi_1;\pi_2, v, g_2)$
(2) $\mathrm{PD}(\pi_1,\pi_2) \vdash S_{\{1\}} * V'_{\{2\}}$
(3) $\mathrm{PD}(\pi_1,\pi_2) \vdash S_{\{2\}} * V'_{\{1\}}$

Proof. Given preconditions we have $v_1 = \{\texttt{k} = \varepsilon_k^1; \texttt{p} = \varepsilon_p^1\}$ and $v_2 = \{\texttt{k} = \varepsilon_k^2; \texttt{p} = \varepsilon_p^2\}$ for some ε_k^1,ε_k^2,ε_p^1,ε_p^2 with correlations as per Definition 7.7. Let π be as defined in Definition 7.8. We observe:

$$\begin{array}{c} \pi_2 = \\ \pi[\varepsilon_k^1/\texttt{flip[2,"gate:a.k"]}][\varepsilon_p^1/\texttt{flip[2,"gate:a.p"]}] \\ [\varepsilon_k^2/\texttt{flip[2,"gate:b.k"]}][\varepsilon_p^1/\texttt{flip[2,"gate:b.p"]}] \end{array}$$

Given that g_1 and g_2 are distinct and not wired in π_1 we are assured that `owl`(g_1) and `owl`(g_1) are in independent uniform distributions, and given that $g \notin \pi_1$ we are assured that `owl`(g) contains entirely fresh flips. Thus by condition (i) of Definition 7.8 and Lemma 6.12 we have:

$$\pi_1;\pi_2 \vdash vars(\varepsilon_k^1, \varepsilon_k^2, \varepsilon_p^1, \varepsilon_p^2) * vdefs(\pi_2)$$

Thus by Lemmas 6.11 and D.1 we establish postconditions (ii) and (iii).

Also since v_1 and v_2 are correlated either positively or negatively with $\texttt{owl}(g_1)$ and $\texttt{owl}(g_2)$ respectively by precondition (ii), by Definition 7.7, precondition (i), and Lemma 6.13 we establish postcondition (i), since Definition 7.8 requires gate output correlation with $\texttt{owl}(g)$ given any input valence conditions. □

Lemma 7.13 (YGC Encode Invariant). *Given well-typed $\pi_1; E[encode(s_1, s_2)]$ and certified encode where If the following preconditions hold where $iovars(\pi_1) = S \cup V$:*

(1) $\mathrm{PD}(\pi_1) \vdash S_{\{1\}} * V_{\{2\}}$
(2) $\mathrm{PD}(\pi_1) \vdash S_{\{2\}} * V_{\{1\}}$

then we have as postconditions where $iovars(\pi_1) = S' \cup V'$:

(1) $goc(\pi_1; \pi_2, v_1, s_1)$ and $goc(\pi_1, v_2, s_2)$
(2) $\mathrm{PD}(\pi_1; \pi_2) \vdash S'_{\{1\}} * V'_{\{2\}}$
(3) $\mathrm{PD}(\pi_1; \pi_2) \vdash S'_{\{2\}} * V'_{\{1\}}$

Proof. By Definition 7.9 we have for some "s1" and "s2":

$$\mathrm{BD}(\pi_2) \vdash \{\texttt{s[1,"s1"]}\} * vdefs(\pi_2)_{\{2\}} \qquad \mathrm{BD}(\pi_2) \vdash \{\texttt{s[2,"s2"]}\} * vdefs(\pi_2)_{\{1\}}$$

Since we assume that secrets are in uniform and independent marginal distributions a priori, and $vars(\pi_1) \cap vars(\pi_2) = \varnothing$ by condition (iv) of Definition 7.9 and assumptions of well-typedness, conditions (ii-iii) follow by Lemmas 6.11 and D.1. Also by Definition 7.9 we have $goc(\pi_2, v_1, s_1)$ and $goc(\pi_2, v_1, s_1)$, so also $goc(\pi_1; \pi_2, v_1, s_1)$ and $goc(\pi_1; \pi_2, v_1, s_1)$ by condition (iv) of Definition 7.9 and Lemmas 6.11 and D.1. □

Theorem 7.14. *If $out := \mathsf{decode}(g, e)$ is a well-typed YGC circuit definition using certified components where no gate output is wired more than once, and $\langle \varnothing, out := \mathsf{decode}(g, e)\rangle \longrightarrow^* \langle \pi, \varnothing\rangle$, then NIMO$(\pi)$.*

Proof. By assumptions of well-typedness we have for some ε_k, ε_p, and ε:

$$\langle \varnothing, out := \mathsf{decode}(g, e)\rangle \longrightarrow^* \langle \pi_1, out := \mathsf{decode}(g, \{\mathsf{k} = \varepsilon_k; \mathsf{p} = \varepsilon_p\})\rangle \longrightarrow^* \langle \pi_1; \pi_2, out := \varepsilon\rangle$$

Let $iovars(\pi_1) = S \cup V$. By Lemmas 7.13 and 7.12 we have $\mathrm{PD}(\pi_1) \vdash S_{\{1\}} * V_{\{2\}}$ and $\mathrm{PD}(\pi_1) \vdash S_{\{2\}} * V_{\{1\}}$. Let π be as defined in Definition 7.10. We observe:

$$\pi_2 =$$
$$\pi[\varepsilon_k/\{\texttt{flip[2,"gate:c.k"]}][\varepsilon_p/\texttt{flip[2,"gate:c.p"]}]$$

so also by Definition 7.10 and Lemma 6.12 we have:

$$\mathrm{BD}(\pi_1; \pi_2) \vdash vars(\varepsilon_k, \varepsilon_p) * vdefs(\pi_2)$$

so, letting $iovars(\pi_1; \pi_2) = S \cup V'$ by Lemmas 6.11 D.1 we have $\mathrm{PD}(\pi_1; \pi_2) \vdash S_{\{1\}} * V'_{\{2\}}$ and $\mathrm{PD}(\pi_1; \pi_2) \vdash S_{\{2\}} * V'_{\{1\}}$. Thus, letting $\pi' \triangleq (\pi_1; \pi_2; out := \varepsilon)$, for all β:

$$\mathrm{PD}(\pi')_{(S_{\{1\}} \cup V_{\{2\}} | \{out \mapsto \beta\})} \vdash S_{\{1\}} * V_{\{2\}} \quad \text{and} \quad \mathrm{PD}(\pi')_{(S_{\{2\}} \cup V_{\{1\}} | \{out \mapsto \beta\})} \vdash S_{\{2\}} * V_{\{1\}}$$

thus for all $m \in mems(V_2 \cup \{out\})$:

$$\mathrm{PD}(\pi')_{(S_{\{1\}} | m)} = \mathrm{PD}(\pi')_{(S_{\{1\}} | m_{\{out\}})}$$

and for all $m \in mems(V_1 \cup \{out\})$:

$$\mathrm{PD}(\pi')_{(S_{\{2\}} | m)} = \mathrm{PD}(\pi')_{(S_{\{2\}} | m_{\{out\}})}$$

establishing the result by Lemma 5.9 and Definition 5.6. □