Automatic Proofs of Privacy of Secure Multi-Party Computation Protocols Against Active Adversaries

Martin Pettai\textsuperscript{1,2,3}, Peeter Laud\textsuperscript{2}
University of Tartu\textsuperscript{1}, Cybernetica AS\textsuperscript{2}, STACC\textsuperscript{3}
martin.pettai@ut.ee, peeter.laud@cyber.ee

Abstract We describe an automatic analysis to check secure multiparty computation protocols against privacy leaks. The analysis is sound — a protocol that is deemed private does not leak anything about its private inputs, even if active attacks are performed against it. Privacy against active adversaries is an essential ingredient in constructions aiming to provide security (privacy + correctness) in adversarial models of intermediate (between passive and active) strength. Using our analysis we are able to show that the protocols used by the Sharemind secure multi-party computation platform are actively private.

1 Introduction

In a secure multiparty computation (SMC) problem, each party $P_i$ from the set of $n$ parties provides input $x_i$ and expects to learn $y_i$, where $(y_1,\ldots,y_n) = f(x_1,\ldots,x_n)$ for some publicly known function $f$. Moreover, each party $P_i$ is expected to learn only $y_i$; it must learn nothing more about the inputs and outputs of other parties (except of what can be deduced from $x_i$ and $y_i$).

SMC is the universal cryptographic functionality. Constructions transforming any $f$ to a SMC protocol have been known for a long time [20] and are considered to be too inefficient for practical use. Over the last years, many frameworks for SMC have been proposed [29,17,21,10,12,28,23,31], facilitating the specification of secure protocols and bringing the practical use of SMC closer to reality.

Security of a SMC protocol is defined as the indistinguishability (using appropriate simulators) of the execution of the protocol from the use of an ideal functionality that computes $f$. Security thus implies that (i) the protocol preserves privacy by not letting the adversarial parties to learn anything they could not learn through the interaction with the ideal functionality, and (ii) the protocol delivers the correct answer to all non-adversarial parties. Recent results have given value to the studies of privacy independently of security. Indeed, privacy is composable [8], allowing complex private protocols to be constructed from simpler ones. The private protocols can then be transformed to secure ones by additional checks near the end of their execution. Also, private protocols are necessary to achieve consistent computations [27] that provide a cheaper alternative to actively secure protocols. Both cases need privacy against active adversaries.
Our privacy checker is targeted towards SMC protocols that aim to provide information-theoretic privacy against adversarial parties. Typically, such protocols [18,11] use secret sharing to represent the intermediate values during the computation. The checker is integrated with our toolchain for compiling and maintaining the SMC protocols for SHAREMIND [25]. Using it, the protocols are first specified in a high-level declarative language similar to the pseudo-code appearing in publications on SMC protocols [16,11], often including simpler protocols as subroutines. The specification is compiled to an intermediate representation (IR), analyzed and optimized. From the IR, code in C++ is generated and compiled together with the rest of the SHAREMIND system. Our privacy checker works on the IR, which is highly suitable for such analyses.

The SHAREMIND platform allows privacy-preserving applications to be specified as a composition of SMC protocols [9]. For the results of our analysis to be applicable to such applications, we need composability. We show that the property checked by our analyser is composable. Additionally, the existing composability results for privacy [8] easily carry over to the active security model. Either of them is usable to deduce the privacy preservation of complex applications.

The benefits of our privacy checker are the most apparent in developing, extending and maintaining large sets of SMC protocols (currently, SHAREMIND employs more than 100 different protocols for various arithmetic, relational, and database operations with shared values), that work with additively shared values. In this setting it gives us guarantees that the specified protocols are private and the compilation (up to the intermediate representation) and the optimizations do not destroy this property. It is infeasible to obtain such guarantees in a way that does not involve significant automation — the protocol set is large, the optimizations (including automatic parallelization of subprotocols) applied to them often complex and their security properties subtle. The results of our privacy checker offer much more confidence than manually generated and verified proofs of privacy (whether against active or even just passive attacks).

We have applied our privacy checker to the SHAREMIND protocol set and verified that they indeed provide active privacy. This opens up the possibilities to use SHAREMIND in settings where security against active attacks is necessary.

We will begin in Sec. 2 by defining the security and privacy of SMC protocols. In Sec. 3, we will describe the IR of protocols in SHAREMIND (for the generation of IR, see [25]). We will describe our algorithm for verifying input privacy in Sec. 4 and prove its soundness in Sec. 5. In Sec. 6, we will describe how we implemented the algorithm and how it worked in practice. We will review the related work in Sec. 7 and discuss our results in Sec. 8.

2 Security and Privacy of SMC Protocols

We define the privacy (and security) of protocols in the usual manner [14,19], through the ideal-real paradigm. An SMC protocol $\pi$ for $n$ parties is black-box private (or secure) if there is a simulator $\text{Sim}$, such that for all environments $Z$
and adversaries $\mathcal{A}$, the views of $\mathcal{Z}$ in configurations $\mathcal{Z} \parallel \pi \| \mathcal{A}$ and $\mathcal{Z} \parallel \mathcal{F} \| (\text{Sim} \| \mathcal{A})$ are indistinguishable. Here $\mathcal{F}$ is either $F^I_{\text{priv}}$ or $F^I_{\text{sec}}$, the ideal SMC functionality for $n$ parties computing $f$ for the purposes of defining privacy or security. The functionality $F^I_{\text{priv}}$ is simple: at the beginning it accepts the the adversary’s requests to corrupt a number (up to a certain bound) of parties (i.e. we are only dealing with non-adaptive corruptions in this paper). It will then obtain the inputs of the $n$ parties from $\mathcal{Z}$ and send the corrupted parties’ inputs to the adversary. It will produce no further output.

The definition of $F^I_{\text{sec}}$ is more involved — it also produces outputs to $\mathcal{Z}$ if the adversary allows. Also, the inputs and the outputs of the corrupted parties can be further modified by the adversary. As we do not deal with security of SMC protocols in this paper, we will refer to [19] for further discussions.

We see that for showing the privacy of $\pi$, we must present a simulator that is able to construct the (distribution of the) messages exchanged between corrupt and non-corrupt parties, using just the inputs of corrupt parties. The simulation must work for all (joint) probability distributions of parties’ inputs, generated by any possible $\mathcal{Z}$. As next, we will give a sufficient condition for the simulator to exist. Our privacy analysis checks for this condition. Let $P$ denote the set $\{1, \ldots, n\}$.

Lemma 1. For a protocol $\pi$ and sets of participants $A, B$, where $A \cap B = \emptyset$ and $A \cup B = P$, let $D_{A,B}^\pi(x; y)$ denote the probability distribution of messages sent by parties in $B$ to parties in $A$ in the execution of $\pi$ with the adversary $A$, if $x$ is the tuple of inputs to parties in $A$ and $y$ the tuple of inputs to parties in $B$. If for all adversaries $A$, all sets of parties $A$ that can be corrupted, all tuples of inputs $x, y_1, y_2$ we have $D_{A \setminus A, P \setminus A}^\pi(x, y_1) = D_{A \setminus A, P \setminus A}^\pi(x, y_2)$, then $\pi$ is black-box private.

Proof. The simulator $\text{Sim}$ works as follows. In the beginning, it receives from $A$ the request to corrupt parties in the set $A \subset P$. It forwards this request to the ideal functionality $\mathcal{F}_{\text{priv}}$, and receives back their inputs $x$. It will now pick an arbitrary $y$ as the inputs of the parties in $P \setminus A$, and run the protocol $\pi$ on the inputs $(x, y)$, together with the adversary $A$. The adversary, even in cooperation with the environment $\mathcal{Z}$, cannot distinguish this run from a real run of $\pi$, where the inputs to parties in $P \setminus A$ may differ, because the messages that $\text{Sim}$ sends to the adversary come from the same distribution, and these messages are the only inputs $A$ and $\mathcal{Z}$ receive from the protocol / simulator. 

3 The Protocols

The intermediate representation of our protocols, also used by our privacy checker, is an arithmetic circuit with the nodes having an extra attribute. Fig. 1 shows the (unoptimized) protocol for multiplying two values in the main protocol set of SHAREMIND, based on additive sharing among three parties and tolerating one corruption [11]. The protocol computes $w = uv$, where each value $x$ is represented as $x = (x_1 + x_2 + x_3) \mod N$ for a fixed modulus $N$, with $i$-th party
holding the value $x_i$. In Fig. 1, nodes labeled with $u$ and $v$ denote the input nodes for parties; nodes labeled with $w$ denote the outputs. Nodes labeled $\$\$ denote the generation of a random element of $\mathbb{Z}_N$.

**Figure 1.** SHAREMIND’s multiplication protocol

The extra attribute of each node is the identity of the party executing it. In Fig. 1, this is denoted by the shape of the node. This attribute immediately determines the messages sent from one party to the other. The communication is depicted in Fig. 1 by drawing the edges corresponding to message sends with solid lines, while local dependencies are drawn with dotted lines.

Formally, an arithmetic circuit $G$ consists of

- A set of nodes $V_G$.
- A mapping $\text{pr}_G : V_G \to V_G^*$, giving the predecessors of each node. The predecessors are ordered. The predecessor relation must define a directed acyclic graph (dag). Let $u \rightarrow_G v$ denote $u \in \text{pr}_G(v)$.

A labeling $\lambda_G : V_G \rightarrow \text{Op}$, giving the operation in each node. The number of operands of $\lambda_G(v)$ must equal $|\text{pr}_G(v)|$.

A labeling $U_G : V_G \rightarrow [n]$, giving the party executing this node.

The set of supported operations $\text{Op}$ contains some special elements. A node $v$ labeled with input is an input vertex. The value in this node is given by party $U_G(v)$. If this party is honest, then the value in this node should remain secret to the adversary. A node labeled with random denotes random number generation. The value in this node is a uniformly distributed element of $\mathbb{Z}_N$, independent of all other random values and inputs.

All other operations are required to be deterministic. A protocol run of $G$ is a mapping $r_G : V_G \rightarrow \mathbb{Z}_N$, where $r_G(v)$ is arbitrary if $\lambda_G(v)$ is input or random, and $r_G(v) = \lambda_G(v)(r_G(v_1), \ldots, r_G(v_k))$ for all other vertices $v$, where $\text{pr}_G(v) = v_1 \cdots v_k$. Let $\mathcal{R}_G$ denote the set of all protocol runs of $G$.

In Sharemind’s protocols, certain operations, for example addition, subtraction, exclusive or, negation, etc. are reversible, according to the following definition.

**Definition 1.** A $k$-ary operation $\otimes \in \text{Op}$ is reversible if $\{z \mid y \in \mathbb{Z}_N, z = \otimes(x_1, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_k)\} = \mathbb{Z}_N$ for all $i$ and $x_1, \ldots, x_k$. Let there be an operation $\text{rev}_\otimes \in \text{Op}$, such that $y = \text{rev}_\otimes(x_1, \ldots, x_{i-1}, z, x_{i+1}, \ldots, x_k)$.

Let $\text{OpR} \subseteq \text{Op}$ be the set of all reversible operations.

Our analyser tries to prove privacy against a coalition of parties $A$ where $A$ is a non-empty proper subset of $P$. Any information about the inputs of the parties outside $A$ must not be leaked to vertices that belong to parties in $A$. To show privacy in the sense of Sec. 2, the analyser must be run for each coalition $A$ that the adversary is allowed to corrupt.

4 Algorithm for Privacy Checking

We consider the case where the adversary $\mathcal{A}$ is active. In this case, it is possible that $\mathcal{A}$ does not follow the protocol correctly. It must still receive and send the same number (and type) of messages as in the original protocol. Thus the subgraph of the circuit induced by the vertices of the parties in $A$ may be replaced with a black box that has the appropriate number of incoming and outgoing edges.

As we are interested in information-theoretic security, we may assume the adversary to be a deterministic algorithm. Indeed, for each possible random tape $s$, the adversary $\mathcal{A}(s)$, using the random coins determined by $s$, has a certain advantage in distinguishing the protocol $\pi$ from the composition $\mathcal{F} || \text{Sim}$. We may pick an $s$ that maximizes this advantage and consider it to be hard-wired into $\mathcal{A}$.

The input of the adversary is the list of values coming from the circuit into the black box together with their arrival times (i.e. the adversary can measure the time at which messages arrive). We consider times to be non-negative integers,
i.e. the time is discrete and the protocol is started at time 0. The output of the adversary is the list of values sent from the black box to the circuit together with their dispatch times.

We can model the black box as a subgraph. It has one vertex, called the adversarial sink, where all edges coming into the black box end. For each message sent out from the adversary, it has one vertex, called an adversarial source, from which exactly one edge goes out of the black box. The value of an adversarial source vertex is the corresponding value sent from the adversary. The operation of the vertex is considered to be the special nullary operation $\text{advsrc}$. The value of the adversarial sink may be thought as a tuple of all values sent to the adversary and its operation would be the special operation $\text{advsink}$ with an appropriate arity. The values of the adversarial source vertices are uniquely determined by the value of the adversarial sink vertex but we do not know anything about this dependency because the active adversary may choose any values for the adversarial source vertices.

The resulting graph (with the subgraph controlled by the adversary replaced, as described above) is still a circuit, and we call circuits of this kind, active-adversarial dags (aadags).

**Definition 2.** We say that an arithmetic circuit $G$ is an aadag iff all of the following hold:

1. $G$ has exactly one vertex with the operation $\text{advsink}$ (we assume $\text{advsink} \in \text{Op}$) and this vertex has no outgoing edges. We call such a vertex the adversarial sink and denote it $\text{sink}_G$.
2. Every vertex of $G$ that has the operation $\text{advsrc}$ (we assume $\text{advsrc} \in \text{Op}$) has exactly one outgoing edge and no incoming edges. We call such vertices the adversarial sources.
3. There are no edges in $G$ whose both endpoints are in the set consisting of the adversarial sink and the adversarial sources. The adversarial sink and the adversarial sources are called the adversarial vertices and the rest of the vertices are called the non-adversarial vertices.

Given an arithmetic circuit $G$ and a set of parties $A$, our privacy checker transforms the circuit to the corresponding aadag, where the adversarially controlled subgraph is replaced with the nodes labeled $\text{advsrc}$ and $\text{advsink}$. It will then execute the algorithm given in Fig. 2 and 3. Let $\text{thr}_1(X_1, \ldots, X_k)$ denote the set of elements that are present in exactly one of the sets $X_1, \ldots, X_k$. Let $\text{thr}_{\geq 2}(X_1, \ldots, X_k)$ denote the set of elements that are present in at least two sets.

## 5 Soundness

### 5.1 Preliminaries

In the following proofs, we assume that the input is an aadag but the algorithm gives the same result on the original arithmetic circuit. If the input is an aadag,
These are part of the input to the algorithm (the circuit itself is given implicitly):

- **numRandoms** is the number of random vertices outside $A$ in the circuit
- **numVertices** is the number of all vertices in the circuit
- **randomIndex** is a one-to-one mapping from the set of random vertices outside $A$ to the set $\{0, \ldots, \text{numRandoms} - 1\}$
- **randomIndex(v)** = $i$ iff $v$ is a random vertex that is mapped to the index $i$; the vertex $v$ is called the $i$th random vertex

These are filled by the for each loop:

- **isSensitive** is an array of numVertices booleans
- **isSensitive[v]** = true iff vertex $v$ contains information that must not leak to $A$
- **leakRandoms** is an array of numVertices subsets of $\{0, \ldots, \text{numRandoms} - 1\}$
- $i \in \text{leakRandoms}[v]$ iff the $i$th random value may be leaked from vertex $v$
- **usableRandoms** is an array of numVertices subsets of $\{0, \ldots, \text{numRandoms} - 1\}$
- $i \in \text{usableRandoms}[v]$ iff the $i$th random value may be used to encrypt the information in vertex $v$ (if $A$ does not get any information about the $i$th random value from elsewhere)

**Figure 2.** Global variables used by the algorithm

the set $A$ in the algorithm is the set consisting of the adversarial sink and the adversarial sources. If the input is the original circuit, then $A$ is the set of vertices belonging to the adversary.

Because the adversary is deterministic, its output (including the timings) is uniquely determined by its input. Parts of the input can be sent to the adversary at different times and parts of the output can be sent by the adversary already before the whole input has arrived. All input and output messages sent at or before time $T - 1$ will uniquely determine all messages sent at time $T$.

Each operation outside the adversary takes a positive integer amount of time that does not depend on its operands. The adversary cannot send an output message $m$ earlier than 1 unit of time after all the input messages that $m$ depends on have arrived.

In Sec. 5.2, we will see how to make aadag transformations that do not change the protocol. In Sec. 5.3, we will prove some loop invariants of the algorithm. In Sec. 5.4, we will look at the timing of vertices. In Sec. 5.5, we will define some properties that imply privacy and prove the main theorem that the property verified by the algorithm implies privacy. In Sec. 5.6, we will prove that the property verified by our algorithm is composable.

### 5.2 Simulating Aadags

Protocol runs of an aadag is defined similarly to protocol runs of a circuit but with special handling of adversarial sources and the sink.

**Definition 3.** A protocol run of an aadag $G$ is a mapping $r_G : V_G \setminus \{\text{sink}_G\} \rightarrow \mathbb{Z}_N$, where $r_G(v)$ is arbitrary if $\lambda_G(v) \in \{\text{input}, \text{random}, \text{advsrc}\}$, and $r_G(v) =$
for each vertex $v$ (in topological order of vertices) do
  if $v \in A$ then
    isSensitive[$v$] ← false
    leakRandoms[$v$] ← ∅
    usableRandoms[$v$] ← ∅
  else if $\lambda_G(v) = \text{input}$ then
    isSensitive[$v$] ← true
    leakRandoms[$v$] ← ∅
    usableRandoms[$v$] ← ∅
  else if $\lambda_G(v) = \text{random}$ then
    isSensitive[$v$] ← false
    leakRandoms[$v$] ← ∅
    usableRandoms[$v$] ← $\{i\}$, where $i = \text{randomIndex}(v)$
  else if $\lambda_G(v) \in \text{OpR}$ then
    Let $v_1 \cdots v_k = \text{pr}_G(v)$
    isSensitive[$v$] ← $\bigvee_{i=1}^k$ isSensitive[$v_i$]
    leakRandoms[$v$] ← $\text{thr}_{\geq 2}(\text{usableRandoms}[v_1], \ldots, \text{usableRandoms}[v_k]) \cup \bigcup_{i=1}^k \text{leakRandoms}[v_i]$
    usableRandoms[$v$] ← $\text{thr}_1(\text{usableRandoms}[v_1], \ldots, \text{usableRandoms}[v_k]) \setminus \text{leakRandoms}[v]$
  else
    Let $v_1 \cdots v_k = \text{pr}_G(v)$
    isSensitive[$v$] ← $\bigvee_{i=1}^k$ isSensitive[$v_i$]
    leakRandoms[$v$] ← $\bigcup_{i=1}^k \text{leakRandoms}[v_i] \cup \bigcup_{i=1}^k \text{usableRandoms}[v_i]$
    usableRandoms[$v$] ← ∅
  end if
end for

Let $S$ be the set of non-random vertices outside $A$ from which there is an edge into $A$

while there exists a vertex $v \in S$ and an index $i$ such that
  $i \in \text{usableRandoms}[v] \land$
  for each vertex $w \in S$ different from $v$:
    $i \notin \text{leakRandoms}[w] \land i \notin \text{usableRandoms}[w]$
do
  Modify the aadag into a simulating aadag where the operation of $v$ is random
  (as described in Lemmas 2 and 3 and their proofs)
Remove $v$ from $S$
if for each vertex $v \in S$: isSensitive[$v$] = false then
  exit (the protocol is private)
else
  exit (cannot prove that the protocol is private)
end if

Figure 3. The algorithm itself
\(\lambda_G(v)(r_G(v_1), \ldots, r_G(v_k))\) for all other vertices \(v\), where \(pr_G(v) = v_1 \cdots v_k\). Let \(R_G\) denote the set of all protocol runs of \(G\).

We now define a certain preorder that allows us to make graph transformations that do not change the protocol described by the graph.

**Definition 4.** We say that an aadag \(G_1\) can simulate an aadag \(G_2\) iff all of the following hold:

1. \(V_{G_1} = V_{G_2} = V\);
2. \(\lambda_{G_1}^{-1}(\text{input}) = \lambda_{G_2}^{-1}(\text{input})\);
3. \(\lambda_{G_1}^{-1}(\text{advsrc}) = \lambda_{G_2}^{-1}(\text{advsrc})\);
4. There is a vertex \(v_a \in V\), such that \(\lambda_{G_1}(v_a) = \lambda_{G_2}(v_a) = \text{advsink}\);
5. \(|\lambda_{G_1}^{-1}(\text{random})| = |\lambda_{G_2}^{-1}(\text{random})|\);
6. The edges to the adversarial sink are the same in both graphs;
7. \(R_{G_1} = R_{G_2}\);
8. If \(v\) is a vertex from which there is an edge to the adversarial sink then every path ending in \(v\) in \(G_1\) is also a path in \(G_2\) and for every vertex on this path that is non-random in \(G_1\), the operation, the incoming edges, and the ordering of predecessors are the same as in \(G_2\).

It is easy to see that this relation is reflexive and transitive, i.e. it is a preorder.

The next lemma (and its proof) describes the basic modification made by the algorithm that moves the generation of a random value one step nearer to the adversary but does not change the protocol.

**Lemma 2.** Suppose all of the following hold in an aadag \(G\):

- \(z \rightarrow_G y\) and that edge is unique;
- \(\lambda_G(z) = \text{random}\);
- \(\lambda_G(y) \in \text{OpR}\);
- All paths from \(z\) to the adversarial sink in \(G\) go through the vertex \(y\)

Then there exists an aadag \(G'\) that can simulate \(G\) such that

- \(\lambda_{G'}(y) = \text{random}\)
- the only edges that may differ between \(G\) and \(G'\) are those that end in \(z\) or \(y\)
- the only vertices whose operation or ordering of predecessors may differ between \(G\) and \(G'\) are \(z\) and \(y\)

**Proof.** We construct \(G'\) from \(G\) using the following transformation that may change the operation and predecessors of \(z\) and \(y\) but does not change anything else:

- Let \(v_1 \cdots v_k = pr_G(y)\), where \(v_m = z\)
- Let \(f = \lambda_G(y)\). Then \(f \in \text{OpR}\) and \(\text{rev}_f^{\text{pr}} \in \text{Op}\)
- Let \(v_i' = v_i\) for all \(i \neq m\) and let \(v_m' = y\)
- Set \(\lambda_{G'}(z) = \text{rev}_f^{\text{pr}}\) and \(pr_{G'}(z) = v'_1 \cdots v'_k\)
- Set \(\lambda_{G'}(y) = \text{random}\) and \(pr_{G'}(y) = \varepsilon\) (empty)
It is easy to see that $G'$ is a valid aadag (the changed vertices still have the same number of incoming edges as the arity of their operation). It is also easy to see that $G'$ satisfies the three itemized statements of the current lemma.

We now show that $G'$ can simulate $G$. The propositions 1, 2, 3, 4, 6 of Def. 4 are obviously satisfied. Proposition 5 is also satisfied by the isomorphism that relates $z$ in $G$ to $y$ in $G'$ and all other random vertices (which are different from $z$ and $y$ and thus their operation is not changed) to themselves.

Now we look at proposition 7 of Def. 4. Consider any protocol run $r_G$ in $G$. It fixes the values of all vertices in $G$ except the adversarial sink. Because the only vertices whose operation or predecessors may have changed from $G$ to $G'$ are $z$ and $y$, all other vertices of $G'$ (whose operation is not input, random, advsrc, or advsink) have a value equal to their operation applied to the values of their predecessors. $\lambda_{G'}(y) = \text{random}$. Thus we only need to check that the value of $z$ in $G'$ is equal to its operation applied to the values of its predecessors, i.e. that $r_G(z) = \lambda_{G'}(z)(r_G(v'_1), \ldots, r_G(v'_k))$.

Because $r_G$ is a run of $G$, we have

$$r_G(y) = \lambda_G(y)(r_G(v_1), \ldots, r_G(v_k)) = f(r_G(v_1), \ldots, r_G(v_k))$$

Because $f$ is reversible, we have (from Def. 1)

$$r_G(v_m) = \text{rev}^m(r_G(v'_1), \ldots, r_G(v'_k))$$

Thus the values of all vertices (except the adversarial sink) are correctly calculated according to their operations in $G'$ and values of their predecessors in $G'$ and $r_G$ is also a protocol run in $G'$.

Also every protocol run in $G'$ is a protocol run in $G$ because if we apply the transformation described above to $G'$ (exchanging the roles of $y$ and $z$), we get $G''$ which is the same as $G$.

Now consider proposition 8 of Def. 4. Let $v$ be a vertex from which there is an edge to the adversarial sink in $G'$ (and $G$). Let $w$ be a vertex in $G'$ from which there is a path to $v$. Let this path be $p$. If $p$ does not contain either of the vertices $y$ and $z$ then proposition 8 of Def. 4 follows from the second and third itemized statements of the current lemma. Now consider the other case. Let $u$ be the last vertex on $p$ that is in the set $\{y, z\}$. Then the path from $u$ to $v$ also exists in $G$ because no edges on this path are changed between $G$ and $G'$. If $u$ is $z$ then the path from $u$ through $v$ to the adversarial sink in $G$ must contain $y$. Contradiction. Thus $u$ is $y$. Because $y$ is nullary in $G'$, it must be the first vertex of the path $p$, i.e. $u$, $v$, and $y$ are the same vertex. Thus the path $p$ also exists in $G$. For every vertex on $p$ different from $y$, i.e. for every vertex on $p$ that is non-random in $G'$, the operation, the incoming edges, and the ordering of predecessors are the same as in $G$. This proves proposition 8 of Def. 4.

The next lemma shows how we make some number of modifications described in the previous lemma to move the generation of a random value to a vertex whose value is sent to the adversary without any further calculations. Thus the message sent to the adversary is now completely random.
Lemma 3. Suppose all of the following hold in an aadag $G$:

- $v_0, v_1, \ldots, v_n (n \geq 1)$ is the only path in $G$ from the $i$th random vertex $v_0$ to the vertex $v = v_n$
- The operation of $v_0$ is random
- The operations of $v_1, \ldots, v_n$ are reversible and not $\text{advsink}$
- All paths from the $i$th random vertex to the adversarial sink in $G$ go through the vertex $v$

Then there exists an aadag $G'$ that can simulate $G$ such that

- the operation of $v$ is random in $G'$
- the only edges that may differ between $G$ and $G'$ are those that end in one of the vertices $v_0, v_1, \ldots, v_n$
- the only vertices whose operation or ordering of predecessors may differ between $G$ and $G'$ are $v_0, v_1, \ldots, v_n$

Proof. Use induction over $n$. The base case $n = 1$ holds by Lemma 2 with $z = v_0$ and $y = v$. Now consider the case $n \geq 2$. We use the induction hypothesis for the path $v_0, v_1, \ldots, v_{n-1}$. The first three premises are obviously satisfied. The fourth one is also satisfied because all paths from $v_0$ to the adversarial sink in $G$ go through $v$ and the only path from $v_0$ to $v$ also goes through $v_{n-1}$. The induction hypothesis gives us an aadag $G_1$ that can simulate $G$ such that

- the operation of $v_{n-1}$ is random in $G_1$
- the only edges that may differ between $G$ and $G_1$ are those that end in one of the vertices $v_0, \ldots, v_{n-1}$
- the only vertices whose operation or ordering of predecessors may differ between $G$ and $G_1$ are $v_0, \ldots, v_{n-1}$

Now we use Lemma 2 with $v_{n-1}$ as $z$, $v$ as $y$, and $G_1$ as $G$. The first premise is satisfied because $G_1$ can simulate $G$ and the edge from $v_{n-1}$ to $v$, which exists in $G$, cannot have changed in $G_1$. The second and third premise are obviously satisfied.

Now consider the fourth premise. Suppose there exists a path from $v_{n-1}$ to the adversarial sink in $G_1$ that does not go through $v$. Let $v_k$ be the last vertex on this path that is in the set $\{v_0, \ldots, v_{n-1}\}$. This vertex exists because $v_{n-1}$ is in that set. Then the path from $v_k$ to the adversarial sink does not contain any of the vertices $v_0, \ldots, v_{n-1}$, except as the first vertex. Thus the edges in this path have not changed between $G$ and $G_1$, thus the path also exists in $G$. We prepend to this path the path $v_0, \ldots, v_k$ (a prefix of the path $v_0, v_1, \ldots, v_n$). Now we have a path from $v_0$ to the adversarial sink in $G$ that does not go through $v$. Contradiction. Thus the fourth premise is also satisfied.

Lemma 2 now gives us an aadag $G'$ that can simulate $G_1$ such that

- the operation of $v$ is random in $G'$
- the only edges that may differ between $G_1$ and $G'$ are those that end in $v_{n-1}$ or $v$
the only vertices whose operation or ordering of predecessors may differ between $G_1$ and $G'$ are $v_{n-1}$ and $v$

Now $G'$ can simulate $G$ because of transitivity and the statements of the current lemma for $G'$ follow from the statements we got from Lemma 2 and from the induction hypothesis (which was described above).  

\[
\square
\]

5.3 Algorithm Invariants

In the next two lemmas, we prove some loop invariants of the algorithm.

Lemma 4. After the execution of the \textbf{for each} loop in Fig. 3, the following statements hold for all $v$ different from the adversarial sink and all $i$:

\begin{itemize}
    \item $\text{isSensitive} [v] = \text{true}$ if and only if there is a path from a non-adversarial input vertex to the vertex $v$
    \item If $i \notin \text{leakRandoms}[v] \land i \notin \text{usableRandoms}[v]$ then there is no path from the $i$th random vertex to the vertex $v$
    \item If $i \in \text{usableRandoms}[v]$ then there exists exactly one path from the $i$th random vertex to $v$ and the operation of every vertex on this path is reversible and different from $\text{advsink}$, except the first vertex, whose operation is \text{random}
\end{itemize}

Proof. Each vertex of the circuit is processed exactly once in the \textbf{for each} loop and this processing assigns values to the array elements $\text{isSensitive}[v], \text{leakRandoms}[v]$, and $\text{usableRandoms}[v]$, but it does not modify any other array elements. This processing uses the array elements of the predecessors of $v$, which have already been calculated because vertices are processed in topological order. We use induction on the index of $v$ in the topological order of vertices different from the adversarial sink. We are allowed to exclude the sink because the sink is not a predecessor of any vertex.

Consider the first statement. From the algorithm code, one of the following statements holds:

\begin{itemize}
    \item $v \in A$, $\text{isSensitive}[v] = \text{false}$, and because $v$ is not the adversarial sink, it must be an adversarial source and thus there is no edge that ends in $v$
    \item $v \notin A$, $\text{isSensitive}[v] = \text{true}$, and the operation of $v$ is \text{input}
    \item $v \notin A$, $\text{isSensitive}[v] = \text{false}$, and the operation of $v$ is \text{random}
    \item $\text{isSensitive}[v] = \text{false}$, and for each predecessor $w$ of $v$, $\text{isSensitive}[w] = \text{false}$ and thus from the induction hypothesis, there is no path from the non-adversarial input vertices to $w$
    \item $\text{isSensitive}[v] = \text{true}$, and there exists a predecessor $w$ of $v$ such that $\text{isSensitive}[w] = \text{true}$ and thus from the induction hypothesis, there is a path from a non-adversarial input vertex to $w$
\end{itemize}

In all five cases, there is a path from a non-adversarial input vertex to $v$ if and only if $\text{isSensitive}[v] = \text{true}$.

Now consider the second statement. Suppose $i \notin \text{leakRandoms}[v] \land i \notin \text{usableRandoms}[v]$. Then from the algorithm code, one of the following statements holds:
– $v \in A$ and thus $v$ is an adversarial source
– $v \notin A$ and the operation of $v$ is input
– $v \notin A$ and the operation of $v$ is random and $\text{randomIndex}(v) \neq i$
– $v$ is not the $i$th random vertex and for each predecessor $w$ of $v$, $i \notin \text{leakRandoms}[w] \land$
  $i \notin \text{usableRandoms}[w]$, and thus from the induction hypothesis, there is no path from the $i$th random vertex to $w$

In all three cases, there is no path from the $i$th random vertex to $v$.

Now consider the third statement. Suppose $i \in \text{usableRandoms}[v]$. Then from the algorithm code, one of the following statements holds:

– $v \notin A$ and the operation of $v$ is random and $\text{randomIndex}(v) = i$
– The operation of $v$ is reversible and different from $\text{advsink}$ and for exactly one predecessor $w$ of $v$, $i \in \text{usableRandoms}[w]$. In this case, by the induction hypothesis, there exists exactly one path from the $i$th random vertex $r$ to $w$, and this path has the required property. We extend this path with the edge from $w$ to $v$. Thus we have one path from $r$ to $v$ and it has the required property. Suppose there exists another path from $r$ to $v$. Let the vertex before $v$ in this path be $w'$. If $w' = w$ then we would have two different paths from $r$ to $w$, a contradiction. Thus $w' \neq w$ and there exists a path from $r$ to $w'$. By the induction hypothesis for the second statement, $i \notin \text{leakRandoms}[w'] \land i \notin \text{usableRandoms}[w']$ does not hold. Also $i \notin \text{usableRandoms}[w']$ because $w$ is the only predecessor of $v$ for which $i \in \text{usableRandoms}[w]$ holds. Thus $i \in \text{leakRandoms}[w']$. By the algorithm code (we know that we are in the fourth branch of the if statement) this implies $i \in \text{leakRandoms}[v]$ and $i \notin \text{usableRandoms}[v]$. Contradiction.

In both cases, there exists exactly one path from the $i$th random vertex to $v$ and the operation of every vertex on this path is reversible, except the first vertex, whose operation is random.

\[\square\]

**Lemma 5.** At the beginning and end of each iteration of the while loop in Fig. 3, the following statements hold:

– The statements of Lemma 4 hold for all $v \in S$ and for all $i$
– The set $S$ is the set of non-random vertices from which there is an edge into the adversarial sink
– The aadag at that point can simulate the aadag originally given to the algorithm

**Proof.** The set $S$ is initially finite and each iteration of the while loop removes one vertex from $S$, thus the while loop terminates. We use induction on the iteration number of the while loop. At the beginning of the first iteration, the first statement holds by Lemma 4 (and because $S$ does not contain the adversarial sink) and the second statement is true because of the initial value of $S$ defined just before the while loop. The third statement holds because the two dags mentioned are the same at this point. At the beginning of any other iteration,
the statements follow from the induction hypothesis for the end of the previous iteration.

Now suppose the statements hold at the beginning of an iteration. Consider the changes made to the aadag during the iteration. We change the current aadag $G$ into a new aadag $G'$ that can simulate $G$ using the transformations described in Lemma 3 with the $i$th random vertex as $v_0$ and the vertex $v$ as $v$. Because $i \in \text{usableRandoms}[v]$, the induction hypothesis gives that there exists exactly one path in $G$ from the $i$th random vertex to $v$ and the operation of every vertex on this path is reversible and different from $\text{advsink}$, except the first vertex, whose operation is random. Because $i \not\in \text{leakRandoms}[w] \land i \not\in \text{usableRandoms}[w]$ for all $w \in S$ different from $v$, the induction hypothesis gives that there is no path from the $i$th random vertex to any vertex in $S$ except $v$. Because $v \in S$, $v$ is not random and thus there is also no path from the $i$th random vertex to any vertex not in $S$ from which there is an edge to the adversarial sink. Thus all paths from the $i$th random vertex to the adversarial sink go through $v$. Thus all premises of Lemma 3 are satisfied and we can apply this lemma.

Lemma 3 and the transitivity of the \emph{can simulate} relation gives that the third statement of the current lemma holds at the end of the iteration.

The only changes made to the aadag during the iteration are in the vertices of the path $v_0, v_1, \ldots, v_n$ in Lemma 3. These vertices may have their operations or incoming edges changed but no other vertices or edges are modified. There is no path from any of these vertices $v_i$ to any of the vertices in $S$ except $v$ (because otherwise there would also be path from $v_0$ to that vertex).

The only vertex with an outgoing edge into the adversarial sink which has its operation or incoming edges changed, is $v$. It is changed from non-random to random, thus it should be removed from $S$, and no other vertex should have its membership in $S$ changed. This is exactly what is done in the algorithm, thus the second statement of the current lemma holds at the end of the iteration.

Now we consider the first statement of the current lemma. The statements of Lemma 4 depend only on the values of arrays $\text{isSensitive}$, $\text{leakRandoms}$, and $\text{usableRandoms}$, and on paths that end in a vertex in $S$ and on the operations of the vertices on those paths. The arrays are not changed during the iteration and the paths and the operations of the vertices on the paths are also not changed if we consider $S \setminus \{v\}$ instead of $S$ because $v$ is removed from $S$ during the iteration. Thus the first statement of the current lemma also holds at the end of the iteration.

\[\square\]

### 5.4 Timing

The next two definitions and a lemma describe the timing of the vertices of the aadag.

**Definition 5.** A protocol run timing function is a function $F$ that maps every tuple consisting of

- the identifier of a vertex $v$
the operation of \( v \)
- the list of identifiers of the predecessors of \( v \) (including their ordering as predecessors)
- the list of timings of the predecessors of \( v \)

to the timing of \( v \) and satisfies the following:
- the timing of \( v \) is a positive integer greater than the timing of every predecessor of \( v \)
- if the operation of \( v \) is random then the timing of \( v \) is 1
- \( F \) is monotonic in the timing of each predecessor of \( v \)

**Definition 6.** The timing of a vertex \( v \) in aadag \( G \) according to the protocol run timing function \( F \) is the integer obtained by applying the function \( F \) to the timings of the predecessors of \( v \) in \( G \) according to \( F \) and some other information that does not depend on \( F \) (as described in Def. 5). This definition is recursive (the timings can be calculated in the topological order of vertices in \( G \)).

**Lemma 6.** If an aadag \( G_1 \) can simulate an aadag \( G_2 \), \( v \) is a vertex from which there is a path to the adversarial sink in \( G_1 \), and \( F \) is a protocol run timing function then the timing of \( v \) in \( G_1 \) according to \( F \) is less than or equal to the timing of \( v \) in \( G_2 \) according to \( F \).

*Proof.* We use induction over the vertices of \( G_1 \) from which there is a path to the adversarial sink, in the topological order of vertices of \( G_1 \).

Let \( v \) be a vertex of \( G_1 \) from which there is a path to the adversarial sink. If \( v \) is random in \( G_1 \) then its timing in \( G_1 \) is 1, which is less than or equal to the timing of \( v \) in \( G_2 \), which is a positive integer.

Now we consider the case where \( v \) is not random in \( G_1 \). Then the predecessors of \( v \) in \( G_1 \) are also vertices from which there is a path to the adversarial sink and by the induction hypothesis, the timings of the predecessors of \( v \) in \( G_1 \) are less than or equal to the timings of the same vertices in \( G_2 \). Because \( G_1 \) can simulate \( G_2 \), we can use proposition 8 of Def. 4 and the operation of \( v \), the identifiers of the predecessors of \( v \), and their ordering as predecessors are the same in \( G_1 \) and \( G_2 \). Also the identifier of \( v \) is the same in \( G_1 \) and \( G_2 \). Because \( F \) is monotonic in the timing of each predecessor of \( v \), the timing of \( v \) in \( G_1 \) is less than or equal to the timing of \( v \) in \( G_2 \).

\( \square \)

5.5 Properties Implying Privacy

The next lemma relates the non-existence of paths between certain vertices of the aadag and statistical independence of the values of those vertices.

**Lemma 7.** Let \( X_1, \ldots, X_m \) be the values of nullary vertices \( v_1, \ldots, v_m \) (with \( X_i \) the value of \( v_i \)) in the aadag and \( Y_1, \ldots, Y_n \) be the values of any vertices \( w_1, \ldots, w_n \) (with \( Y_j \) the value of \( w_j \)) in the aadag. If for each \( i \in \{1, \ldots, m\} \) and each \( j \in \{1, \ldots, n\} \) there is no path in the aadag from \( v_i \) to \( w_j \) then the tuple \( (Y_1, \ldots, Y_n) \) is independent of the tuple \( (X_1, \ldots, X_m) \).
Proof. If we fix the values of all nullary vertices (adversarial sources, input vertices, and random vertices) in the aadag then the values of all vertices are automatically determined because all non-nullary operations in the aadag are deterministic. The probability distribution of \(Y_1, \ldots, Y_n\) is determined by the probability distribution of the nullary vertices. Because there is no path in the aadag from any of the vertices corresponding to \(X_1, \ldots, X_m\) to any of the vertices corresponding to \(Y_1, \ldots, Y_n\), the values \(Y_1, \ldots, Y_n\) do not change if we change \(X_1, \ldots, X_m\) but keep all other values of nullary vertices unchanged. Thus the conditional probability distribution of \((Y_1, \ldots, Y_n)\) given \((X_1, \ldots, X_m)\) is the same as the unconditional probability distribution of \((Y_1, \ldots, Y_n)\), i.e. \((Y_1, \ldots, Y_n)\) is independent of \((X_1, \ldots, X_m)\).

Now we will define two properties of aadags. The first is the one that we want (input privacy) and the second is the property that is actually verified by the algorithm. We will also prove that the first property implies black-box privacy. Then we will prove the theorem that the second property implies the first one and thus we can use the algorithm to verify input privacy.

**Definition 7.** A protocol given as an aadag is private against active adversaries iff for every active adversary, where

\(-\) for every positive integer \(T\), the adversary’s output at time \(T\) (the list of adversarial sources whose timing is \(T\) and their values) is uniquely determined by the adversary’s input by time \(T - 1\) (the values and timings of vertices whose timing is \(T - 1\) or less from which there is an edge into the adversarial sink),
\(-\) and the adversary does not output anything at time \(0\) or earlier,

the tuple of values of vertices from which there is an edge into the adversarial sink is independent of the tuple of values of non-adversarial input vertices.

**Lemma 8.** If a protocol given as an aadag is private against active adversaries (according to Def. 7) then the protocol is also black-box private as defined in Sec. 2.

Proof. Consider any protocol \(\pi\) and the corresponding aadag that is private against active adversaries. Consider any adversary \(A\) and any set of vertices \(A\) that the adversary may corrupt. The messages sent from the parties in \(P \setminus A\) (where \(P\) is the set of all parties) to the parties in \(A\) are exactly the values of vertices from which there is an edge into the adversarial sink.

Let the probability distribution of these messages be \(D_{A, P \setminus A}^{\pi}(x, y)\), where \(x\) is the tuple of inputs to parties in \(A\) and \(y\) is the tuple of inputs to parties in \(P \setminus A\), i.e. the values of non-adversarial input vertices. By Def. 7, \(D_{A, P \setminus A}^{\pi}(x, y)\) does not depend on \(y\), thus \(D_{A, P \setminus A}^{\pi}(x, y_1) = D_{A, P \setminus A}^{\pi}(x, y_2)\) for all \(x, y_1,\) and \(y_2\), and by Lemma 1, the protocol is black-box private. \(\Box\)

**Definition 8.** We say that the algorithm recognizes an aadag \(G\) as private iff the algorithm run with \(G\) as input returns the protocol is private.
Theorem 1. If the algorithm in Figures 2 and 3 recognizes an aadag $G$ as private then the protocol corresponding to $G$ is private against active adversaries (according to Def. 7).

Proof. By Lemma 5 (the third statement at the end of the final iteration), the aadag $G_2$ before the final if can simulate the original aadag $G_1$.

Now we turn our attention to the final if. The first statement of Lemma 4 and the condition of the if give that there is no path from the non-adversarial input vertices to the vertices in $S$ (which, by the second statement of Lemma 5 at the end of the final iteration, is the set of non-random vertices from which there is an edge into the adversarial sink). There is also no path from the non-adversarial input vertices to the random vertices from which there is an edge into the adversarial sink because a random vertex has no incoming edges and a random vertex also cannot be an input vertex.

By Lemma 7, the tuple of values of vertices from which there is an edge into the adversarial sink is independent of the tuple of values of non-adversarial input vertices. Thus the final aadag $G_2$ is private.

Suppose the initial aadag $G_1$ is not private, i.e. there exists an adversary $A$ that can gain information about the secret inputs. Fix a protocol run timing function $F$. By Lemma 6, the timings of the vertices from which there is a edge into the adversarial sink in $G_2$ are less than or equal to the timings of the same vertices in $G_1$. We construct an adversary $A'$ that uses $A$ as an oracle, and for every message that $A'$ receives from a vertex $v$ in $G_2$, it waits 0 or more units of time and forwards the message to $A$ at time equal to the timing of $v$ in $G_1$. Every message that $A$ sends out, is immediately (in 0 units of time) forwarded by $A'$ to an adversarial source in $G_2$.

Consider any possible protocol run in $G_1$. Because $G_2$ can simulate $G_1$, this run is also possible in $G_2$. To get this run, we set the secret input to the same values as in the original run, and set the random values to the values of the corresponding vertices in the original run (which might not have been random there). We use $A'$ as the adversary (with $A$ as its oracle). After all these inputs are fixed, the adversary and the non-adversarial aadag are deterministic.

We use induction over time moments (non-negative integers) to prove that at each time moment $T$, the values of vertices whose timing is $T$ or less are the same in the original run (produced with $G_1$ and $A$) and the simulating run (produced with $G_2$ and $A'$). At $T = 0$, this holds because all timings are positive and thus no vertices have yet been given values. Suppose that the induction hypothesis holds for at time $T - 1$. Then at time $T$, the values of the vertices with timing $T$ are determined. These values are the same in the original and the simulating run because of the induction hypothesis, the construction of $A'$, and the restriction on adversaries given in Def. 7. Thus the induction statement holds also at time $T$ and, by induction, also for all $T$.

If the runs terminate (if one does then so does the other), i.e. all vertices receive a value after some finite amount of time, then they produce the same protocol runs. Thus the adversary gains the same information about the secret
inputs as in the initial aadag. Thus the final aadag $G_2$ is not private. Contradiction.

If the runs do not terminate then (because the aadags are finite) there exists a time moment $T$ after which no more vertices receive a value. Then the produced protocol runs on the subgraphs of $G_1$ and $G_2$ (induced by the vertices whose values are determined) are the same for $G_1$ and $G_2$. Thus the adversary still gains the same information about the secret inputs as in the initial aadag. Thus the final aadag $G_2$ is not private. Contradiction.

\[\square\]

5.6 Composability

The property verified by our algorithm is also composable, as stated by the following theorem. Because the algorithm does not depend on which vertices are output vertices, we can use any vertices (except the adversarial sink) as output vertices.

**Theorem 2.** If the algorithm in Figures 2 and 3 recognizes the aadags $G_1$ and $G_2$ as private then the aadag $G$ obtained from $G_1$ and $G_2$ by uniting the output vertices of $G_1$ with the input vertices of $G_2$ and the adversarial sink of $G_1$ with that of $G_2$, is also recognized by the algorithm as private.

**Proof.** Suppose that $G_1$ and $G_2$ are recognized by the algorithm as private. If $v$ and the $i$th random vertex are in the same subgraph $G_j$ then the truth value of the statement $i \in \text{usableRandoms}[v]$ is the same in the algorithm run for $G_j$ and that for $G$ (this can be proved by induction over $v$ in $G$, in the topological order). The same can be proved for $i \in \text{leakRandoms}[v]$. Thus, if the condition of the while loop in the algorithm is true for some $v$ and $i$ in the run for $G_1$ or $G_2$ then the condition is also true for the same $v$ and $i$ in the run for $G$.

Thus the algorithm does the same modifications in the run for $G$ as in the runs for $G_1$ and $G_2$ and each modification is local to either $G_1$ or $G_2$. The run for $G$ may make some additional modifications (and thus remove additional elements from $S$) that cross the border between $G_1$ and $G_2$ if there is a path from a random vertex in $G_1$ through an output vertex of $G_1$ into $G_2$. Thus the value of $S$ that is used in the final if in the run for $G$ is a subset of the union of the final values of $S$ in the runs for $G_1$ and $G_2$. Let the three values of $S$ be $S_0, S_1, S_2$, respectively. Then $S_0 \subseteq S_1 \cup S_2$. Because $G_1$ and $G_2$ are recognized as private by the algorithm, $\text{isSensitive}[v] = \text{false}$ for every $v \in S_1$ in the run for $G_1$ and every $v \in S_2$ in the run for $G_2$.

Suppose that there exists $v \in S_0$ (and thus either $v \in S_1$ or $v \in S_2$) such that $\text{isSensitive}[v] = \text{true}$ in the run for $G$. By Lemma 4, there is a path from an input vertex $u$ of $G$ (which is also an input vertex of $G_1$) to $v$ in $G$. If $v \in S_1$ then the whole path is in $G_1$ and thus $\text{isSensitive}[v] = \text{true}$ also in the run for $G_1$. Contradiction. If $v \in S_2$ then the path from $u$ to $v$ goes through a vertex $w$ that is an input vertex of $G_2$. Thus there is a path from an input vertex of $G_2$ to $v$ in $G_2$ and $\text{isSensitive}[v] = \text{true}$ also in the run for $G_2$. Contradiction.
Thus for all \( v \in S \), \( \text{isSensitive}[v] = \text{false} \) for every \( v \in S_0 \) in the run for \( G \). Thus the condition of the final if is satisfied in the run for \( G \) and the algorithm recognizes \( G \) as private.

\( \square \)

The proof is easily generalized to the case where we have a modular composition of any number of protocols where the input of any sub-protocol may come from the outputs of several other sub-protocols or from global inputs. The composition must still not contain cycles.

### 6 Implementation

We have implemented the algorithm and it can prove privacy of several useful protocols on additively shared secrets, namely all protocols in [11], except for the division protocols that have not yet been implemented in the high-level language for protocols (but it would probably work on those as well). The protocols are used as building blocks in the SHAREMIND framework [6,7,11].

We have tested the algorithm for input sizes of 8, 16, 32, or 64 bits. The algorithm must be run separately for each input size because the circuit generated for a protocol may be different for different input sizes (the protocol may use recursion over the bits, which must be unfolded in the circuit). We have tested the algorithm only in the case of three parties, of which one can be corrupted. The algorithm takes the subset of the corrupted parties as an argument, thus it must be executed separately for each subset of corrupted parties for which privacy is required, unless the protocol is symmetrical in the parties.

In the implementation we use bit vectors (of length \( \text{numRandoms} \)) to encode the sets \( \text{leakRandoms}[v] \) and \( \text{usableRandoms}[v] \). Set operations then become bitwise boolean operations. The implementation also skips the modifications of the aadag because these do not change the output of the algorithm and were needed only for proving the soundness of the algorithm.

The time complexity of the algorithm (as stated in Fig. 3 but without modifications of the aadag) is \( O(N \cdot R) \), where \( N = \text{numVertices} \) and \( R = \text{numRandoms} \). To find the values \( v \) and \( i \) satisfying the condition of the while in \( O(N + R) \) time instead of \( O(N \cdot R) \), we use helper arrays that contain for each index \( i \) the number of vertices \( v \in S \) for which \( i \in \text{leakRandoms}[v] \) and the number of vertices \( v \in S \) for which \( i \in \text{usableRandoms}[v] \).

The largest circuit we tested (corresponding to a private shift-right protocol for 64 bits) had \( N = 44039 \) and \( R = 3428 \). Three runs of the algorithm (one against each of three parties as an adversary) on this circuit took a total of 2.5 seconds on a 2.50 GHz laptop.

### 7 Related Work

Our privacy checker is a solid example of the sequence-of-games approach to cryptographic proofs [3], the examples of which are many [24,5,1,2]. We make a sequence of small modifications to the non-adversarial part of the protocol
circuit. The modifications do not change the view of the adversary but may change how this view is computed. After all these modifications, the view of the adversary is computed without any reference to the inputs of honest parties, and input privacy is now obvious. Our analyser does not actually modify the protocol during its run; in this sense, it is similar to [26].

Our analysis tracks, which values are masked by which random values using reversible operations. There are some similarities to [22]; they can even claim completeness for a certain verification procedure, but applied to a much simpler and more regular language. In [30], automatic introduction of masks is considered, but the setting is again quite different from us.

Security and composition of multi-party protocols is investigated in [13]. They consider passive, active, non-adaptive, and adaptive adversaries, in the secure-channels and computational settings. For active adversaries, they require both secrecy (input privacy) and correctness. They use ideal adversaries using trusted parties and define an ideal evaluation process for evaluating a function. If this ideal process is emulated by a protocol then the protocol is secure. They do not prove security of concrete protocols, instead they give definitions of security and prove that if some protocols are secure according to these definitions then their modular composition is also secure. In [15], it is shown how to actually construct universally composable secure multiparty computation protocols for all functions $f$.

Compared to [13], we consider only active non-adaptive adversaries in the secure-channels setting, and we are only interested in privacy, not correctness. The ideal process in our case is the transformed aadag (where inputs of the good parties are not used to compute the adversary’s view, and thus privacy is guaranteed) that can simulate the original aadag. Also, we give an algorithm that tries to prove privacy for concrete protocols. We have implemented this algorithm and it succeeds in proving privacy of several useful protocols. It may be possible to add correctness (except for one bit of leakage) to our approach by using consistent computations [27].

Our algorithm is targeted towards multi-party protocols operating on secret shared values. Secret sharing in the secure channels setting (and information-theoretic security) is considered in [4], which also proves some completeness theorems in this setting. Composability for input private (against passive adversaries) protocols on shares is considered in [8]; their treatment is also applicable to active adversaries. We prove composability not for the actual security property (Def. 7) but rather for the property (Def. 8) verified by the algorithm (which implies the security property).

8 Conclusions

We have designed and implemented an algorithm that succeeds in proving the input privacy of most basic building-block three-party protocols on additively secret-shared integers against an active adversary that corrupts one of the parties. The algorithm takes as input the low-level specification of the protocol,
thus it can detect any potential privacy leaks introduced by a translation from a high-level specification. We have proved that the property verified by our algorithm implies input privacy and is composable. Thus protocols composed from the building blocks would also have input privacy.

Our analysis also validates the design decisions of SHAREMIND and its three-party protocol set from the provable security side. It shows that these protocols are secure in more demanding models of security, and paves the way towards fully active security without sacrificing the high performance of the current protocol set.

References


