2015 IEEE 28th Computer Security Foundations Symposium

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

Martin Pettai
University of Tartu / Cybernetica AS / STACC
martin.pettai@cyber.ee

Peeter Laud
Cybernetica AS
peeter.laud@cyber.ee

***Abstract*—We describe an automatic analysis to check secure multi-party 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.**

## I. INTRODUCTION

In a secure multiparty computation (SMC) problem, there are $n$ parties $P_1, \ldots, P_n$, with each party $P_i$ providing an input $x_i$ and expecting to learn the output $y_i$, 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 can be combined with cloud services to outsource computations on private data. In this situation, we may use secret sharing to split the private data between several cloud service providers so that the data cannot be reconstructed without a certain number of them colluding. In the cloud-based setting, the parties are naturally assigned the roles of "input parties", "computation parties" (the cloud servers) and "output parties" [1], some parties may have several roles. Only input parties $P_i$ have non-trivial inputs $x_i$, which they share among the computation parties. Only the output parties $P_j$ have non-trivial outputs $y_j$ that they reconstruct from the shares they receive from the computation parties.

SMC is the universal cryptographic functionality. Constructions transforming any $f$ to a SMC protocol have been known for a long time [2] and are considered to be too inefficient for practical use. Over the last years, many frameworks for SMC have been proposed [3], [4], [5], [6], [7], [8], [9], [10], facilitating the specification of secure protocols and making the use of SMC more practical.

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 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 [11], 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 [12], [13]. Also, private protocols are necessary to achieve *consistent computations* [14] 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 [15], [16] use secret sharing to represent the intermediate values during the computation. These protocols usually have a quite large communication volume (the number of bits exchanged between different parties) of intermediate messages compared to the number of bits published as the final output. This is further amplified when smaller protocols are composed into larger protocols and the intermediate outputs do not need to be published (they remain as secret shares, which are used directly as inputs of the next protocols). This increases the volume of intermediate messages but not the size of the final output.

Thus an adversarial party who makes active attacks by changing the protocol, can potentially leak much more private data through the intermediate messages sent to him than through the final outputs. This gives us the motivation why it is useful to check that the intermediate messages do not leak anything, without worrying about the correctness of final outputs. If we have eliminated the high-bandwidth flows of intermediate messages then only the low-bandwidth flows of final output remain possible. If these still leak too much then we can use other means to reduce these flows without worrying about the intermediate messages.

The SHAREMIND platform supports the specification of privacy-preserving applications through composition of SMC protocols, featuring a rich programming language SECREC [17] for this purpose. 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 [11] easily carry over to the active security model. Either of them is usable to deduce the privacy preservation of complex applications.

The checker is integrated with our toolchain for compiling and maintaining the SMC protocols for SHAREMIND [18]. Using it, the protocols are first specified in a high-level declarative language similar to the pseudo-code appearing in publications on SMC protocols [19], [16], often including simpler

© 2015, Martin Pettai. Under license to IEEE.

DOI 10.1109/CSF.2015.13

protocols as subroutines. This domain-specific language (the protocol DSL) is different from SecreC, which cannot handle individual shares of secret-shared values. 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. Then the protocols can be called from SecreC programs. Our privacy checker works on the IR, which is highly suitable for such analyses.

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; all these tailor-made protocols contribute to the efficiency of the framework), that work with additively shared values. In this setting it gives us guarantees that the used 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. II by defining the security and privacy of SMC protocols. In Sec. III, we will describe the IR of analyzed protocols in Sharemind (for the generation of IR, see [18]). We will describe our algorithm for verifying input privacy in Sec. IV and prove its soundness in Sec. V. In Sec. VI, we will show an example to illustrate composability and the kind of mistakes caught by our algorithm. In Sec. VII, we will discuss what our results imply about the privacy of more complex SMC programs. In Sec. VIII, we will describe how we implemented the algorithm and how it worked in practice. We will review the related work in Sec. IX and discuss our results in Sec. X.

## II. Security and Privacy of SMC Protocols

We define the privacy (and security) of protocols in the usual manner [20], [21], through the ideal-real paradigm.

*Definition 1:* An SMC protocol $\pi$ for $n$ parties is *black-box private* (or secure) if there is a simulator $\mathsf{Sim}$, such that for all environments $\mathcal{Z}$ and adversaries $\mathcal{A}$, the views of $\mathcal{Z}$ in configurations $\mathcal{Z}\|\pi\|\mathcal{A}$ and $\mathcal{Z}\|\mathcal{F}\|(\mathsf{Sim}\|\mathcal{A})$ are indistinguishable. Here $\mathcal{F}$ is either $\mathcal{F}^f_{\mathsf{priv}}$ or $\mathcal{F}^f_{\mathsf{sec}}$, the *ideal SMC functionality* for $n$ parties computing $f$ for the purposes of defining privacy or security. The symbol $\|$ denotes parallel composition (and connecting the corresponding input and output ports for sending messages).

Because we are interested in active adversaries, the adversary $\mathcal{A}$ may have both input and output ports connecting it to $\pi$ in $\mathcal{Z}\|\pi\|\mathcal{A}$. Here $\pi$ would execute the protocol only for the non-corrupted parties, any messages that should be sent to and received from the corrupted parties according to the protocol are instead exchanged with the adversary.

The functionality $\mathcal{F}^f_{\mathsf{priv}}$ is simple: at the beginning it accepts 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 $\mathcal{F}^f_{\mathsf{sec}}$ 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 [21] 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 corrupted and non-corrupted parties, using just the inputs of corrupted 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 $\mathfrak{D}^{\pi,\mathcal{A}}_{A,B}(\vec{x};\vec{y})$ denote the probability distribution of messages sent by parties in $B$ to parties in $A$ in the execution of $\pi$ with the adversary $\mathcal{A}$, if $\vec{x}$ is the tuple of inputs to parties in $A$ and $\vec{y}$ the tuple of inputs to parties in $B$. If for all adversaries $\mathcal{A}$, all sets of parties $A$ that can be corrupted, all tuples of inputs $\vec{x}$, $\vec{y}_1$, $\vec{y}_2$ we have $\mathfrak{D}^{\pi,\mathcal{A}}_{A,P\backslash A}(\vec{x},\vec{y}_1) = \mathfrak{D}^{\pi,\mathcal{A}}_{A,P\backslash A}(\vec{x},\vec{y}_2)$, then $\pi$ is black-box private.

*Proof:* The simulator $\mathsf{Sim}$ works as follows. In the beginning, it receives from $\mathcal{A}$ the request to corrupt parties in the set $A \subset P$. It forwards this request to the ideal functionality $\mathcal{F}_{\mathsf{priv}}$, and receives back their inputs $\vec{x}$. It will now pick an arbitrary $\vec{y}$ as the inputs of the parties in $P\backslash A$, and run the protocol $\pi$ on the inputs $(\vec{x},\vec{y})$, together with the adversary $\mathcal{A}$. 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 $P\backslash A$ may differ, because the messages that $\mathsf{Sim}$ sends to the adversary come from the same distribution, and these messages are the only inputs $\mathcal{A}$ and $\mathcal{Z}$ receive from the protocol / simulator. ■

## III. 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 passive corruption. The protocol computes $w = uv$, where each value $x$ is represented as $x = (x_1 + x_2 + x_3) \bmod N$ 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; each node labeled with +, -, or * computes, respectively, the sum, difference, or product of the values of its predecessors; nodes labeled with $w$ denote the outputs. Nodes labeled \$ denote the generation of a random element of $\mathbb{Z}_N$.

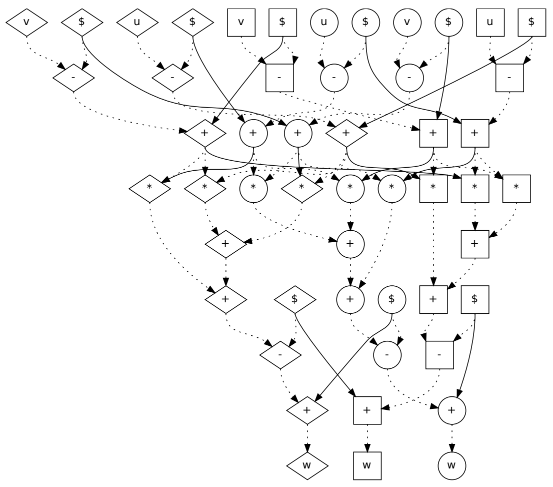

Figure 1. SHAREMIND's multiplication protocol

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 another. The communication is depicted in Fig. 1 by drawing the edges corresponding to message sends with solid lines, while local dependencies are shown with dotted lines.

We stress that the arithmetic circuits that we consider in this paper correspond to the building-block protocols operating on shares (such as the protocol in Fig. 1), thus both the input and the output of the protocols consist of shares of some values, not the values themselves. The actual SMC programs that calculate something useful call these protocols as building blocks after splitting the actual inputs into shares, later recombining the shares of some values into the actual outputs, and maybe also declassifying some intermediate results. We consider these full SMC programs only in Sec. VII.

Formally, an arithmetic circuit $G$ consists of

- A set of nodes $V_G$.
- A mapping $\mathsf{pr}_G : V_G \to V_G^*$, giving the predecessors of each node. The predecessors are ordered. The predecessor relation must define a directed acyclic graph (dag). Let $u \to_G v$ denote $u \in \mathsf{pr}_G(v)$.
- A labeling $\lambda_G : V_G \to \mathbf{Op}$, giving the operation in each node. The number of operands of $\lambda_G(v)$ must equal $|\mathsf{pr}_G(v)|$. We denote the set of nodes having an operation $f$, by $\lambda_G^{-1}(f)$.
- A labeling $\mathbf{U}_G : V_G \to P$, giving the party executing this node.

The set of supported operations $\mathbf{Op}$ contains some special elements. A node $v$ labeled with $\mathsf{input}$ is an *input vertex*. The value in this node is given by party $\mathbf{U}_G(v)$. If this party is honest, then the value in this node should remain secret to the adversary. A node labeled with $\mathsf{random}$ 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. They are not limited to only arithmetic operations, we can also have bitwise operations ($\mathsf{and}$, $\mathsf{or}$, $\mathsf{xor}$, $\mathsf{not}$), bit shifts, or any other deterministic operation.

A *protocol run* of $G$ is a mapping $\mathsf{r}_G : V_G \to \mathbb{Z}_N$, where $\mathsf{r}_G(v)$ is arbitrary if $\lambda_G(v)$ is $\mathsf{input}$ or $\mathsf{random}$, and $\mathsf{r}_G(v) = \lambda_G(v)(\mathsf{r}_G(v_1), \ldots, \mathsf{r}_G(v_k))$ for all other vertices $v$, where $\mathsf{pr}_G(v) = v_1 \cdots v_k$. Let $\mathbf{R}_G$ denote the set of all protocol runs of $G$.

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

*Definition 2:* A $k$-ary operation $\otimes \in \mathbf{Op}$ is *reversible* iff for all $i$ and $x_1, \ldots, x_k$,
$\{\otimes(x_1, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_k) \,|\, y \in \mathbb{Z}_N\} = \mathbb{Z}_N$.

Let $\mathsf{rev}^i_\otimes$ denote the operation in $\mathbf{Op}$ such that
$y = \mathsf{rev}^i_\otimes(x_1, \ldots, x_{i-1}, z, x_{i+1}, \ldots, x_k)$ iff
$z = \otimes(x_1, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_k)$. Let $\mathbf{OpR} \subseteq \mathbf{Op}$ be the set of all reversible operations. We assume that if $\otimes \in \mathbf{OpR}$ then $\mathsf{rev}^i_\otimes \in \mathbf{Op}$. This can be achieved by taking the closure of $\mathbf{OpR}$ under $\mathsf{rev}$, which increases the size of $\mathbf{OpR}$ by at most a factor of $(k+1)!$ (in practice much less), where $k$ is the highest arity of an operation in $\mathbf{OpR}$.

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. II, the analyser must be run for each coalition $A$ that the adversary is allowed to corrupt.

## IV. 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 but it can choose which values it sends out instead of the values that it should send according to the 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 $\mathcal{F}\|\mathsf{Sim}$. We may pick an $s$ that maximizes this advantage and consider it 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. 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 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 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* (*aadag*s).

*Definition 3:* 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 advsink (we assume advsink $\in \mathbf{Op}$) and this vertex has no outgoing edges. We call such a vertex the *adversarial sink* and denote it $\mathsf{sink}_G$
2) Every vertex of $G$ that has the operation advsrc (we assume advsrc $\in \mathbf{Op}$) 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 advsrc and advsink. It will then execute the algorithm given in Figures 2, 3, and 4. Let $\mathsf{thr}_1(X_1, \ldots, X_k)$ denote the set of elements that are present in exactly one of the sets $X_1, \ldots, X_k$. Let $\mathsf{thr}_{\geq 2}(X_1, \ldots, X_k)$ denote the set of elements that are present in at least two sets. In Fig. 4, we use the **fail** statement to indicate the situations where some statements would not be meaningful. We will later see in the soundness proof that these **fail** statements are actually never reached.

The algorithm first (in the **for each** loop) computes some global arrays (described in Fig. 2). Then (in the **while** loop) it tries to find for each vertex $v$ whose value is sent to the adversary, a corresponding random vertex $r$ whose value is used in the computation of $v$ in a way that makes also $v$ uniformly random, e.g. $v = (a+r)+b$. Every random vertex $r$ can be used to ascertain the randomness of only one vertex $v$ and $r$ can not be used in the computation of any other message sent to the adversary whose randomness is not yet ascertained. Otherwise, the adversary might get two messages depending on $r$ (e.g. $v = (a+r)+b$ and $u = (r+c)-d$) and might be able to cancel out (part of) the randomness, e.g. $u-v = c-d-a-b$. In the final **if**, the algorithm checks that those messages sent to the adversary whose randomness could not be ascertained, do not contain any sensitive information.

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 $\{0, \ldots, \mathsf{numRandoms} - 1\}$
- $\mathsf{randomIndex}(v) = i$ 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
- $\mathsf{isSensitive}[v] = \mathsf{true}$ iff vertex $v$ contains information that must not leak to $A$
- leakRandoms is an array of numVertices subsets of $\{0, \ldots, \mathsf{numRandoms} - 1\}$
- $i \in \mathsf{leakRandoms}[v]$ iff the $i$th random value may be leaked from vertex $v$
- usableRandoms is an array of numVertices subsets of $\{0, \ldots, \mathsf{numRandoms} - 1\}$
- $i \in \mathsf{usableRandoms}[v]$ 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 privacy-checking algorithm

The algorithm also transforms the graph during its execution. The transformations do not affect the final output but they simplify the soundness proof because after the transformations the privacy of the protocol will be obvious. The subroutine transformPath in Fig. 4 essentially reverses the edges of the path from $r$ to $v$ where $r$ is the random vertex corresponding to the vertex $v$ whose value is sent to the adversary. Now $v$ will be the random vertex instead of $r$ and the randomness of the message sent to the adversary becomes obvious. For the reversal of edges to be possible, the operation of all nodes (except the first) on the path must be reversible. This ensures that the reversal does not change the value of any vertex, only the order of their computation.

## V. SOUNDNESS

### A. 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, 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

```
for each vertex $v$ (in topological order of vertices) do
    if $v \in A$ then
        $\mathsf{isSensitive}[v] \leftarrow \mathsf{false}$
        $\mathsf{leakRandoms}[v] \leftarrow \emptyset$
        $\mathsf{usableRandoms}[v] \leftarrow \emptyset$
    else if $\lambda_G(v) = \mathsf{input}$ then
        $\mathsf{isSensitive}[v] \leftarrow \mathsf{true}$
        $\mathsf{leakRandoms}[v] \leftarrow \emptyset$
        $\mathsf{usableRandoms}[v] \leftarrow \emptyset$
    else if $\lambda_G(v) = \mathsf{random}$ then
        $\mathsf{isSensitive}[v] \leftarrow \mathsf{false}$
        $\mathsf{leakRandoms}[v] \leftarrow \emptyset$
        $\mathsf{usableRandoms}[v] \leftarrow \{i\}$, where $i = \mathsf{randomIndex}(v)$
    else if $\lambda_G(v) \in \mathbf{OpR}$ then
        Let $v_1 \cdots v_k = \mathsf{pr}_G(v)$
        $\mathsf{isSensitive}[v] \leftarrow \bigvee_{i=1}^{k} \mathsf{isSensitive}[v_i]$
        $\mathsf{leakRandoms}[v] \leftarrow \mathsf{thr}_{\geq 2}(\mathsf{usableRandoms}[v_1], \ldots, \mathsf{usableRandoms}[v_k]) \cup \bigcup_{i=1}^{k} \mathsf{leakRandoms}[v_i]$
        $\mathsf{usableRandoms}[v] \leftarrow \mathsf{thr}_1(\mathsf{usableRandoms}[v_1], \ldots, \mathsf{usableRandoms}[v_k]) \setminus \mathsf{leakRandoms}[v]$
    else
        Let $v_1 \cdots v_k = \mathsf{pr}_G(v)$
        $\mathsf{isSensitive}[v] \leftarrow \bigvee_{i=1}^{k} \mathsf{isSensitive}[v_i]$
        $\mathsf{leakRandoms}[v] \leftarrow \bigcup_{i=1}^{k} \mathsf{leakRandoms}[v_i] \cup \bigcup_{i=1}^{k} \mathsf{usableRandoms}[v_i]$
        $\mathsf{usableRandoms}[v] \leftarrow \emptyset$

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
            $i \in \mathsf{usableRandoms}[v] \wedge$
            for each vertex $w \in S$ different from $v$:
                $i \notin \mathsf{leakRandoms}[w] \wedge i \notin \mathsf{usableRandoms}[w]$
      do
    Call $\mathsf{transformPath}(v, i)$ (in Fig. 4)
            (This modifies the aadag into a simulating aadag where the operation of $v$ is $\mathsf{random}$)
    Remove $v$ from $S$
if for each vertex $v \in S$: $\mathsf{isSensitive}[v] = \mathsf{false}$ then
    exit(the protocol is private)
else
    exit(cannot prove that the protocol is private)
```

Figure 3. The privacy-checking algorithm

```
transformOne$(z, y)$:
  • Let $v_1 \cdots v_k = \mathsf{pr}_G(y)$, where $v_m = z$; if such an
    index $m$ does not exist or is not unique then fail
  • Let $f = \lambda_G(y)$; if $f \notin \mathbf{OpR}$ then fail
  • Let $v'_i = v_i$ for all $i \neq m$ and let $v'_m = y$
  • Set $\lambda_G(z) \leftarrow \mathsf{rev}_f^m$ and $\mathsf{pr}_G(z) \leftarrow v'_1 \cdots v'_k$
  • Set $\lambda_G(y) \leftarrow \mathsf{random}$ and $\mathsf{pr}_G(y) \leftarrow \varepsilon$ (empty)
transformPath$(v, i)$:
  • Let $v_0, v_1, \ldots, v_n$ ($n \geq 1$) be the only path in $G$ from
    the $i$th random vertex $v_0$ to the vertex $v = v_n$; if such
    a path does not exist or is not unique then fail
  • for $i = 0, 1, ..., n-1$ do call $\mathsf{transformOne}(v_i, v_{i+1})$
```

Figure 4. Transformation subroutines in the privacy-checking algorithm

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. V-B, we will see how to make aadag transformations that do not change the protocol. In Sec. V-C, we will state some loop invariants of the algorithm. In Sec. V-D, we will look at the timing of vertices. In Sec. V-E, we will define some properties that imply privacy and prove the main theorem that the property verified by the algorithm implies privacy. In Sec. V-F, we will see that the property verified by our algorithm is composable.

## B. Simulating Aadags

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

*Definition 4:* A *protocol run* of an aadag $G$ is a mapping $\mathsf{r}_G : V_G \setminus \{\mathsf{sink}_G\} \to \mathbb{Z}_N$, where $\mathsf{r}_G(v)$ is arbitrary if $\lambda_G(v) \in \{\mathsf{input}, \mathsf{random}, \mathsf{advsrc}\}$, and $\mathsf{r}_G(v) = \lambda_G(v)(\mathsf{r}_G(v_1), \dots, \mathsf{r}_G(v_k))$ for all other vertices $v$, where $\mathsf{pr}_G(v) = v_1 \cdots v_k$. Let $\mathbf{R}_G$ 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 5:* 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) $\lambda_{G_1}^{-1}(\mathsf{input}) = \lambda_{G_2}^{-1}(\mathsf{input})$;
3) $\lambda_{G_1}^{-1}(\mathsf{advsrc}) = \lambda_{G_2}^{-1}(\mathsf{advsrc})$;
4) There is a vertex $v_\mathsf{a} \in V$, such that $\lambda_{G_1}(v_\mathsf{a}) = \lambda_{G_2}(v_\mathsf{a}) = \mathsf{advsink}$;
5) $|\lambda_{G_1}^{-1}(\mathsf{random})| = |\lambda_{G_2}^{-1}(\mathsf{random})|$;
6) The edges to the adversarial sink are the same in both graphs;
7) $\mathbf{R}_{G_1} = \mathbf{R}_{G_2}$;
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. We call this relation *can simulate* because it is related to the notion of simulation used in Sec. II. As we will see in the proof of Theorem 1, if an aadag $G_2$ can simulate an aadag $G_1$ then there exists a simulator $\mathsf{S}$ such that the view of the adversary $\mathcal{A}$ in configurations $G_1 \| \mathcal{A}$ and $G_2 \| (\mathsf{S} \| \mathcal{A})$ is indistinguishable. In the proof of Theorem 1, $\mathsf{S} \| \mathcal{A}$ is represented by the constructed adversary $\mathcal{A}'$. As we are going to show, our algorithm transforms the original aadag to a new aadag that can simulate it. The privacy of the new aadag is obvious, thus the new aadag can be considered as an ideal functionality for the original protocol.

The next lemma describes the properties of 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 $G$:

- $z \to_G y$ and there is no other path in $G$ from $z$ to $y$;
- $\lambda_G(z) = \mathsf{random}$;
- $\lambda_G(y) \in \mathbf{OpR}$;
- All paths from $z$ to the adversarial sink in $G$ go through the vertex $y$

Then executing $\mathsf{transformOne}(z, y)$ (from Fig. 4) on $G$ does not ***fail*** and produces an aadag $G'$ that can simulate $G$ such that

- $\lambda_{G'}(y) = \mathsf{random}$
- 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:* The premises of the current lemma ensure that **fail** is not reached when executing $\mathsf{transformOne}(z, y)$. The aadag $G'$ is constructed from $G$ using the following transformation (from Fig. 4) that may change the operation and predecessors of $z$ and $y$ but does not change anything else:

- Let $v_1 \cdots v_k = \mathsf{pr}_G(y)$, where $v_m = z$
- Let $f = \lambda_G(y)$. Then $f \in \mathbf{OpR}$ and $\mathsf{rev}_f^m \in \mathbf{Op}$
- Let $v'_i = v_i$ for all $i \neq m$ and let $v'_m = y$
- Set $\lambda_{G'}(z) = \mathsf{rev}_f^m$ and $\mathsf{pr}_{G'}(z) = v'_1 \cdots v'_k$
- Set $\lambda_{G'}(y) = \mathsf{random}$ and $\mathsf{pr}_{G'}(y) = \varepsilon$ (empty)

It is easy to see that $G'$ satisfies the three itemized statements of the current lemma.

Because $G$ is an aadag, it is also a dag. Now we show that $G'$ is a dag, i.e. the transformation does not introduce cycles. Suppose there exists a cycle in $G'$. We consider the shortest such cycle, thus we can assume that it does not contain any vertex or edge more than once. This cycle must contain an edge $e$ that does not occur in $G$, otherwise the whole cycle would occur in $G$, a contradiction. The edge $e$ must end in $z$ or $y$ by the second itemized statement of the current lemma. Because $y$ does not have any predecessors in $G'$, it cannot occur in the cycle. Thus the edge $e$ must go from $a$ to $z$ for some vertex $a$. Now we divide the cycle into two parts: the edge from $a$ to $z$ and the path from $z$ to $a$. The path does not contain any edges that end in $z$ or $y$, thus this path occurs also in $G$.

We consider two cases depending on whether $a = y$ or not. If $a \neq y$ then the edge from $a$ to $z$ also occurs in $G$ and combined with the path from $z$ to $a$, we have a cycle in $G$, a contradiction. If $a = y$ then the path from $z$ to $a$ in $G$ is a path from $z$ to $y$, which contradicts the first premise of the current lemma. Thus $G'$ is a dag.

Now 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).

We now show that $G'$ can simulate $G$. The propositions 1, 2, 3, 4, 6 of Def. 5 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 $y$, thus their operation is not changed) to themselves.

Now we look at proposition 7 of Def. 5. Consider any protocol run $\mathsf{r}_G$ 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 $\mathsf{input}$, $\mathsf{random}$, $\mathsf{advsrc}$, or $\mathsf{advsink}$) have a value equal to their operation applied to the values of their predecessors. $\lambda_{G'}(y) = \mathsf{random}$. 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 $\mathsf{r}_G(z) = \lambda_{G'}(z)(\mathsf{r}_G(v'_1), \ldots, \mathsf{r}_G(v'_k))$.

Because $\mathsf{r}_G$ is a run of $G$, we have

$$\mathsf{r}_G(y) = \lambda_G(y)(\mathsf{r}_G(v_1), \ldots, \mathsf{r}_G(v_k)) = f(\mathsf{r}_G(v_1), \ldots, \mathsf{r}_G(v_k))$$

Because $f$ is reversible, we have (from Def. 2)

$$\mathsf{r}_G(v_m) = \mathsf{rev}^m_f(\mathsf{r}_G(v'_1), \ldots, \mathsf{r}_G(v'_k))$$

$$\mathsf{r}_G(z) = \lambda_{G'}(z)(\mathsf{r}_G(v'_1), \ldots, \mathsf{r}_G(v'_k))$$

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 $\mathsf{r}_G$ 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 $G$ again.

Now consider proposition 8 of Def. 5. 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. 5 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. 5. ■

The next lemma describes the properties of the subroutine $\mathsf{transformPath}$ from Fig. 4, which makes 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. The proof applies Lemma 2 $n$ times.

*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 $\mathsf{random}$
- The operations of $v_1, \ldots, v_n$ are reversible and not $\mathsf{advsink}$
- All paths from the $i$th random vertex to the adversarial sink in $G$ go through the vertex $v$

Then executing $\mathsf{transformPath}(v, i)$ (from Fig. 4) on $G$ does not ***fail*** and produces an aadag $G'$ that can simulate $G$ such that

- the operation of $v$ is $\mathsf{random}$ 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:* The first premise of the current lemma ensures that the **fail** in $\mathsf{transformPath}(v, i)$ is not reached. Then we essentially use Lemma 2 $n$ times to move the generation of a random value from $v_0$ to $v_1$ to ..., and finally to $v_n$ in $G'$. Lemma 2 ensures that the calls to $\mathsf{transformOne}$ do not **fail**.

We 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 $v_0, \ldots, v_{n-1}$. 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 $\mathsf{random}$ in $G_1$, the only edges that may differ between $G$ and $G_1$ are those that end in one of the vertices $v_0, \ldots, v_{n-1}$, and 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 second and third premise are obviously satisfied.

Now consider the first premise. The edge from $v_{n-1}$ to $v$, which exists in $G$, cannot have changed in $G_1$. Suppose there exists another path from $v_{n-1}$ to $v$ in $G_1$. This path cannot contain the edge from $v_{n-1}$ to $v$ because $G_1$ is acyclic. 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 $v$ 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 $v$ in $G$ that does not go through the edge from $v_{n-1}$ to $v$. This path is different from $v_0, v_1, \ldots, v_n$. Contradiction. Thus the first premise of Lemma 2 is 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 $v$. Similarly to the first premise above, we can show that there exists a suffix of this path that is in $G$ and can be prepended with a prefix of the path $v_0, v_1, \ldots, v_n$ to get 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 $\mathsf{random}$ in $G'$, the only edges that may differ between $G_1$ and $G'$ are those that end in $v_{n-1}$ or $v$, and 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). ■

## *C. Algorithm Invariants*

In the next two lemmas, we state some loop invariants of the algorithm. These will be used in the proofs of our Theorems 1 and 2. The lemmas relate the values of certain variables in the algorithm to the properties of the aadag (especially the existence of certain paths). Lemma 5 additionally shows that the aadag transformations made by the algorithm preserve the *can simulate* relation, and that the algorithm never fails by a run-time error.

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

- $\mathsf{isSensitive}[v] = \mathsf{true}$ if and only if there is a path from a non-adversarial input vertex to the vertex $v$
- If $i \notin \mathsf{leakRandoms}[v] \wedge i \notin \mathsf{usableRandoms}[v]$ then there is no path from the $i$th random vertex to the vertex $v$
- If $i \in \mathsf{usableRandoms}[v]$ 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 $\mathsf{advsink}$, except the first vertex, whose operation is $\mathsf{random}$

*Proof:* Due to lack of space, we have to give just a summary instead of the full proof. It uses induction over the vertices of the dag, in topological order, and case analysis based on the code of the algoritm in Fig. 3. The proofs of first two statements are simple. For the third statement, there is one simple case. The more difficult case is

- The operation of $v$ is reversible and different from $\mathsf{advsink}$ and for exactly one predecessor $w$ of $v$, $i \in \mathsf{usableRandoms}[w]$.

By the induction hypothesis, we have the unique path from the $i$th random vertex $r$ to $w$, which we extend with the edge from $w$ to $v$. This new path has the required property. If there is another such path then let the vertex before $v$ in that path be $w'$. Then we consider the cases $w = w'$ and $w \neq w'$ and both lead to contradiction (here we use the induction hypothesis for the second and the third statement). ■

*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
- The algorithm has not reached ***fail***

*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. The fourth statement holds because there are no **fail** statements before the **while** loop. 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 Fig. 4. We use Lemma 3 with the $\mathtt{i}$th random vertex as $v_0$ and the vertex $v$ as $v$. Because $i \in \mathsf{usableRandoms}[v]$, the induction hypothesis gives that there exists exactly one path in $G$ from the $\mathtt{i}$th random vertex to $v$ and the operation of every vertex on this path is reversible and different from $\mathsf{advsink}$, except the first vertex, whose operation is $\mathsf{random}$. Because $i \notin \mathsf{leakRandoms}[w] \wedge i \notin \mathsf{usableRandoms}[w]$ for all $\mathtt{w} \in S$ different from $v$, the induction hypothesis gives that there is no path from the $\mathtt{i}$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 $\mathtt{i}$th random vertex to any vertex not in $S$ from which there is an edge to the adversarial sink. Thus all paths from the $\mathtt{i}$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 *can simulate* relation give that the third and the fourth statement of the current lemma hold 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 $\mathsf{isSensitive}$, $\mathsf{leakRandoms}$, and $\mathsf{usableRandoms}$, 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. ■

## D. Timing

The next two definitions and a lemma describe the timing of the vertices of the aadag. Timing is important for two reasons. First, it allows us to formalize that any message sent or received by the adversary does not depend on the messages sent or received in the future, thus avoiding circular dependencies. This allows us to use induction over time moments to show that the view of the adversary at each time moment (the messages received by that time) is the same in the original aadag and the aadag transformed by the algorithm. Second, timing allows us to prove privacy against a more powerful and realistic adversary who can also measure (and use in the attack) the arrival times of messages, not only their contents.

*Definition 6:* 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), and 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 its timing is 1
- $F$ is monotonic in the timing of each predecessor of $v$

*Definition 7:* 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. 6). 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. 5 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 $G_2$. ■

## *E. Properties Implying Privacy*

In this paper, we are considering four different properties related to privacy. Two of them we already saw in Sec. II. The one in Def. 1 requires that the environment, with the help of the adversary, cannot distinguish between the real protocol and the ideal functionality. The second property is the premise of Lemma 1. It requires that the adversary cannot distinguish between any two instances of the real protocol (the protocol with two different input tuples for the honest parties).

In this section, we will define two more properties. The third property will be defined in Def. 8 and it is equivalent to the second property but it will explicitly specify the class of adversaries (active and deterministic) and restrict the class of protocol implementations to aadags (arithmetic circuits).

The fourth property will be defined in Def. 9 and it is the actual property recognized by the algorithm. It requires that sensitive values (whose computation uses inputs of honest parties) sent to the adversary be masked by randomness, not in another (ad-hoc) way. It is stronger than the third property because we can construct a protocol that sends to the adversary a non-randomized value that does not depend on the inputs of honest parties but whose computation still uses them, e.g. a value $x - x$ where $x$ is an input of an honest party. Such protocols, however, most probably contain bugs and thus it is not a shortcoming that the algorithm rejects them. Furthermore, it is not possible to find a polynomial algorithm for checking the third property, unless P = NP, because we can reduce an NP-complete problem (e.g. 3-SAT) to the problem of checking the third property. The reduction transforms an instance of the NP-complete problem into a protocol that leaks a private input with a non-zero (albeit very small) probability if and only if the answer to the instance is yes.

The four properties are related by Lemma 1, Lemma 7, and Theorem 1, such that the fourth implies the third, which implies the second, which implies the first. Thus if the algorithm accepts a protocol then this protocol is black-box private.

Input privacy against passive adversaries requires that the view of the adversary be independent of non-adversarial inputs. We define input privacy against active adversaries. For this we must quantify the requirement over all adversaries because the view of the adversary now depends on the adversary whereas in the passive case it depends only on the protocol. We restrict the quantification to only deterministic adversaries, which is enough, as we saw in Sec. IV.

*Definition 8:* 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 outputs nothing 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 7:* If a protocol given as an aadag is private against active adversaries (according to Def. 8) then the protocol is also black-box private as defined in Sec. II.

*Proof:* Consider any protocol $\pi$ and the corresponding aadag that is private against active adversaries. Consider any adversary $\mathcal{A}$ and any set of vertices $A$ that the adversary may corrupt. The messages sent from the parties in $P \backslash A$ (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 $\mathfrak{D}^{\pi,\mathcal{A}}_{A,P\setminus A}(\vec{x},\vec{y})$, where $\vec{x}$ is the tuple of inputs to parties in $A$ and $\vec{y}$ is the tuple of inputs to parties in $P\setminus A$, i.e. the values of non-adversarial input vertices. By Def. 8, $\mathfrak{D}^{\pi,\mathcal{A}}_{A,P\setminus A}(\vec{x},\vec{y})$ does not depend on $\vec{y}$, thus $\mathfrak{D}^{\pi,\mathcal{A}}_{A,P\setminus A}(\vec{x},\vec{y}_1) = \mathfrak{D}^{\pi,\mathcal{A}}_{A,P\setminus A}(\vec{x},\vec{y}_2)$ for all $\vec{x}$, $\vec{y}_1$, and $\vec{y}_2$, and by Lemma 1, the protocol is black-box private. ■

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 8:* Let $X_1,\dots,X_m$ be the values of nullary vertices $v_1,\dots,v_m$ (with $X_i$ the value of $v_i$) in the aadag and $Y_1,\dots,Y_n$ be the values of any vertices $w_1,\dots,w_n$ (with $Y_j$ the value of $w_j$) in the aadag. If for each $i \in \{1,\dots,m\}$ and each $j \in \{1,\dots,n\}$: there is no path in the aadag from $v_i$ to $w_j$ then the tuple $(Y_1,\dots,Y_n)$ is independent of the tuple $(X_1,\dots,X_m)$.

*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 $Y_1,\dots,Y_n$ 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 $X_1,\dots,X_m$ to any of the vertices corresponding to $Y_1,\dots,Y_n$, the values $Y_1,\dots,Y_n$ do not change if we change $X_1,\dots,X_m$ but keep all other values of nullary vertices unchanged. Thus the conditional probability distribution of $(Y_1,\dots,Y_n)$ given $(X_1,\dots,X_m)$ is the same as the unconditional probability distribution of $(Y_1,\dots,Y_n)$, i.e. $(Y_1,\dots,Y_n)$ is independent of $(X_1,\dots,X_m)$. ■

*Definition 9:* 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, 3, and 4 recognizes an aadag $G$ as private then the protocol corresponding to $G$ is private against active adversaries (according to Def. 8). Also, the algorithm never reaches ***fail***.

*Proof:* By Lemma 5 (the third and the fourth statement at the end of the final iteration), the aadag $G_2$ before the final **if** can simulate the original aadag $G_1$ and the algorithm has not **fail**ed by the point after the **while** loop. Thus the algorithm never **fail**s because there are no **fail** statements after the **while** loop.

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 8, 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 $\mathcal{A}$ 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 $\mathcal{A}'$ that uses $\mathcal{A}$ as an oracle, and for every message that $\mathcal{A}'$ receives from a vertex $v$ in $G_2$, it waits 0 or more units of time and forwards the message to $\mathcal{A}$ at time equal to the timing of $v$ in $G_1$. Every message that $\mathcal{A}$ sends out, is immediately (in 0 units of time) forwarded by $\mathcal{A}'$ 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 $\mathcal{A}'$ as the adversary (with $\mathcal{A}$ 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 $\mathcal{A}$) and the simulating run (produced with $G_2$ and $\mathcal{A}'$). 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 $\mathcal{A}'$, and the restriction on adversaries given in Def. 8. 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 a 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. ■

## F. Composability

The property verified by our algorithm is composable, as stated by the following theorem. As 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, 3, and 4 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 (we can assume w.l.o.g. that for each random vertex, its index is the same in all considered aadags) are in the same subgraph $G_j$ then the truth value of the statement $i \in \mathsf{usableRandoms}[v]$ 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 $i \in \mathsf{leakRandoms}[v]$.

If $v$ is a non-random vertex in $G_1$ and the $i$th random vertex is in $G_2$ then it is easy to see that $i \notin \mathsf{usableRandoms}[v]$ and $i \notin \mathsf{leakRandoms}[v]$ because there cannot be a path from the $i$th random vertex to $v$.

Note that the condition of the **while** loop

```
    $i \in \mathsf{usableRandoms}[v] \wedge$
    for each vertex $w \in S$ different from $v$:
        $i \notin \mathsf{leakRandoms}[w] \wedge i \notin \mathsf{usableRandoms}[w]$
```

is monotonic in time, i.e. if it becomes true for some $v$ and $i$ at some point in the algorithm run then it cannot become false any more for these $v$ and $i$ because $S$ can only become smaller and other parameters in the condition are constant. Thus if the condition becomes true for some $v$ and $i$ then the vertex $v$ will eventually be removed from $S$.

Let $u_1, \ldots, u_k$ (in this order) be the vertices removed from $S$ in the run for $G_2$. We use induction over this list to prove that for every $k' \le k$, the vertices $u_1, \ldots, u_{k'}$ (not necessarily in this order) will eventually be removed from $S$ in the run for $G$. At the point in the run for $G_2$ where $u_{k'}$ is removed, the condition of the **while** loop holds for $u_{k'}$ and some $i$, where the $i$th random vertex is in $G_2$. Denote the set $S$ at this point by $S_2$. Then $S_2$ contains all vertices of $G_2$ whose value is sent to the adversary except $u_1, \ldots, u_{k'-1}$. By the induction hypothesis, there exists a point $p_1$ in the run for $G$ where $u_1, \ldots, u_{k'-1}$ have been removed from $S$. Denote the set $S$ at this point by $S_1$. Then for every $w \in S_1$, either $w \in S_2$ or $w$ is not a vertex of $G_2$. In both cases, $i \notin \mathsf{leakRandoms}[w] \wedge i \notin \mathsf{usableRandoms}[w]$ holds. Also, $i \in \mathsf{usableRandoms}[u_{k'}]$ holds. Thus the condition holds for $u_{k'}$ and $i$ at point $p_1$ and the vertex $u_{k'}$ will eventually be removed from $S$ in the run for $G$. This proves the induction statement.

Denote by $S_0$ the set $S$ after the **while** loop (i.e. after removing $u_1, \ldots, u_k$) in the run for $G_2$. Because $G_2$ is recognized as private by the algorithm, $\mathsf{isSensitive}[v] = \mathsf{false}$ for every $v \in S_0$ in the run for $G_2$. Thus, by Lemma 4, there is no path from any input vertex of $G_2$ to any vertex in $S_0$. Every path from a vertex of $G_1$ to a vertex of $G_2$ must go through an input vertex of $G_2$. Thus, if $v \in S_0$ and the $i$th random vertex is in $G_1$ then there is no path from the $i$th random vertex to $v$ and thus $i \notin \mathsf{usableRandoms}[v]$ and $i \notin \mathsf{leakRandoms}[v]$.

Now let $q_1, \ldots, q_m$ (in this order) be the vertices removed from $S$ in the run for $G_1$. We use induction over this list to prove that for every $m' \le m$, the vertices $q_1, \ldots, q_{m'}$ (not necessarily in this order) will eventually be removed from $S$ in the run for $G$. At the point in the run for $G_1$ where $q_{m'}$ is removed, the condition of the **while** loop holds for $q_{m'}$ and some $i$, where the $i$th random vertex is in $G_1$. Denote the set $S$ at this point by $S_3$. Then $S_3$ contains all vertices of $G_1$ whose value is sent to the adversary except $q_1, \ldots, q_{k'-1}$. By the induction hypothesis, there exists a point $p_4$ in the run for $G$ where $q_1, \ldots, q_{k'-1}$ and also $u_1, \ldots, u_k$ have been removed from $S$. Denote the set $S$ at this point by $S_4$. Then for every $w \in S_4$, either $w \in S_3$ or $w$ is not a vertex of $G_1$. In the second case, we have $w \in S_0$. In both cases, $i \notin \mathsf{leakRandoms}[w] \wedge i \notin \mathsf{usableRandoms}[w]$ holds. Also, $i \in \mathsf{usableRandoms}[q_{m'}]$ holds. Thus the condition holds for $q_{m'}$ and $i$ at point $p_4$ and the vertex $q_{m'}$ will eventually be removed from $S$ in the run for $G$. This proves the induction statement.

Thus all vertices that are removed from $S$ in the runs of $G_1$ and $G_2$, are also removed in the run for $G$. 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 $S_6, S_7, S_8$, respectively. Then $S_6 \subseteq S_7 \cup S_8$. Because $G_1$ and $G_2$ are recognized as private by the algorithm, $\mathsf{isSensitive}[v] = \mathsf{false}$ for every $v \in S_7$ in the run for $G_1$ and every $v \in S_8$ in the run for $G_2$.

Suppose that there exists $v \in S_6$ (and thus either $v \in S_7$ or $v \in S_8$) such that $\mathsf{isSensitive}[v] = \mathsf{true}$ 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 $v \in S_7$ then the whole path is in $G_1$ and thus $\mathsf{isSensitive}[v] = \mathsf{true}$ also in the run for $G_1$. Contradiction. If $v \in S_8$ 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 $\mathsf{isSensitive}[v] = \mathsf{true}$ also in the run for $G_2$. Contradiction.

Thus for all $v \in S$, $\mathsf{isSensitive}[v] = \mathsf{false}$ for every $v \in S_6$ 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. ■

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.

## VI. An Example

In Fig. 5, we give an example of a bigger protocol composed of two smaller protocols. Each of the subprotocols is represented in the figure as a large box containing an arithmetic circuit. The upper box represents a circuit that takes as input a 1-bit integer $u$ secret-shared modulo 2 ($u = (u_1 + u_2 + u_3) \bmod 2$) and outputs $u$ secret-shared modulo $2^n$ ($u = (w_1 + w_2 + w_3) \bmod 2^n$).

The lower box represents the multiplication of two integers secret-shared modulo $2^n$ ($(u_1 + u_2 + u_3) \cdot (v_1 + v_2 + v_3) \equiv w_1 + w_2 + w_3 \pmod{2^n}$). It differs from Fig. 1 by omitting the resharing (re-randomization) at the end. This is still private and recognized as such by our algorithm, as long as the output is not declassified (it may still be used as input of other protocols working on shares, which do this re-randomization at the beginning).

The combined protocol is a simplified version of oblivious choice. It takes as input a 1-bit integer $a$ (secret-shared modulo 2, as it would be when it is the boolean result of a comparison protocol) and an $n$-bit integer $b$ and returns $b$ if $a = 1$ and 0 if $a = 0$. The output is $a \cdot b$ but we first need to convert $a$ to an $n$-bit integer, thus the need for two subprotocols.

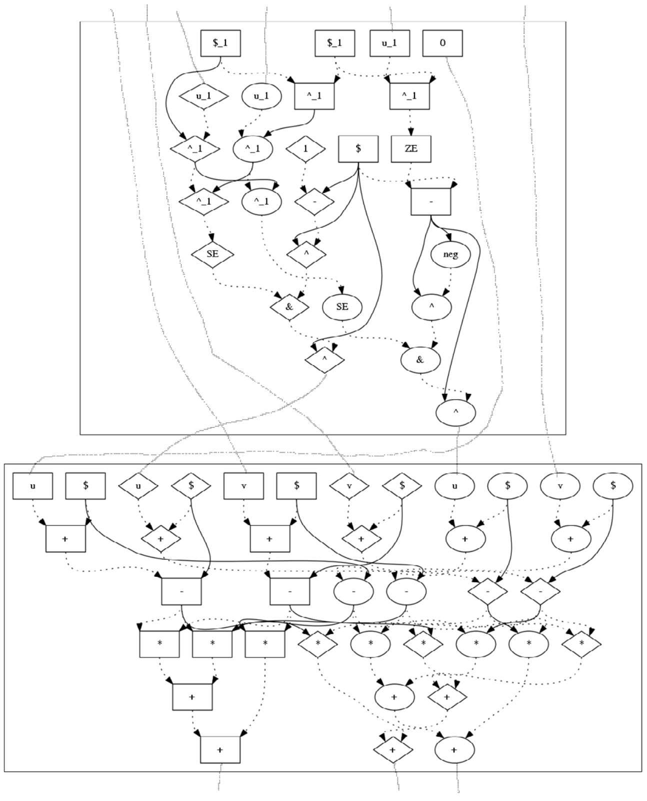

Figure 5. An example of composed arithmetic circuits

The labels of the nodes +, -, *, &, ^, neg, 0, 1, $ denote $n$-bit integer addition, subtraction, multiplication, bitwise AND, bitwise XOR, negation, constant 0, constant 1, random number generation, respectively. The labels with suffix _1 denote the same operations with 1-bit integers instead of n-bit. The labels SE and ZE denote sign extension and zero extension of 1-bit integers into n-bit integers.

At the top of the figure, there are six hand-drawn dotted lines (without arrows on either end) entering the upper box from above. These are the inputs of the composed protocol. Three of those lead to the inputs of the first box, labeled u_1. The other three go through the upper box to some of the inputs of the lower box, labeled $v$. Similar lines go from the outputs of the upper box to the inputs (labeled $u$) of the lower box. These show the pairs of vertices that are united in Theorem 2. And then there are three lines from the outputs of the lower box exiting the box through the bottom. These are the outputs of the composed protocol.

To check the privacy of the composed protocol, we can either check it directly, or check the privacy of each of the two subprotocols separately and then use Theorem 2 to get the privacy of the whole protocol. In the second case, we can check the privacy of the upper box either with or without the three values that are passed through the protocol unmodified, as these do not change the output of the algorithm. Checking components separately also has the advantage that we learn in which component a potential privacy leak occurs if the algorithm fails to prove the privacy.

Now we illustrate what kind of mistakes can be detected by the algorithm. These privacy leaks occur when the protocol does not use enough randomness to mask the values sent to other parties. In Fig. 5, the multiplication protocol uses 6 random nodes, 2 for each party. As the protocol is symmetrical, a programming mistake would probably leave out 3 random nodes (1 for each party) instead of only one and is thus not likely to go unnoticed by the programmer. Thus the multiplication protocol is not very error-prone.

On the other hand, the upper box (converting 1-bit integers to $n$-bit) is not symmetrical, and it contains three random nodes owned by the box party (the party whose nodes are box-shaped). For better illustration, we give also the protocol DSL source code of the graph in the upper box:

```
def shareconv (u : uint[1]) : uint[n+1] = {
  let {1}
    b = rng () //
    m = zextend (b ^ u) // m = zextend (u)
    m12 = rng ()
    m13 = m - m12
    b12 = rng ()
    b13 = b ^ b12; // ;
  let {2}
    s23 = (b12 from 1) ^ u;
  let {3}
    s32 = (b13 from 1) ^ u; // s32 = (b12 from 1) ^ u;
  let {2,3}
    s = (s23 from 2) ^ (s32 from 3);
  return
    party:
      1 -> 0
      2 -> if (s == 1) (1 - (m12 from 1)) else (m12 from 1)
      3 -> if (s == 1) (0 - (m13 from 1)) else (m13 from 1)
}
```

Here `let` $\{i\}$ denotes that the block is evaluated only by party $i$ ($i = 1, 2, 3$); ($v$ `from` $i$) denotes that the value of the variable $v$ is received from party $i$; `party: 1 -> ... 2 -> ... 3 -> ...` denotes that each party evaluates a separate expression. `rng ()` denotes random number generation. The variable `u` has a different value for each party (that party's share of the input). The other variables have only one value. Comments begin with `//` and continue until the end of line.

The three random variables `b`, `m12`, and `b12` are used to split the input (`u`) of party 1 into 4 random shares (`b12`, `b13`, `m12`, `m13`), such that all 4 are needed to reconstruct the input.

It seems counter-intuitive to need 4 shares when there are only 3 parties but actually they are all essential. If any of the random nodes is replaced with a constant or omitted (changing a subexpression of the form $a$ `^` $r$ or $a$ `-` $r$, where $r$ is random, to just $a$) then the protocol is no longer private. Thus programming mistakes are likely to occur here.

The code above passes the privacy check and the comments may be deleted. The comments show the changes that can be made to make the program insecure but still correct: the text before `//` should be replaced with the text after `//` (which may be empty). These changes occur when the programmer forgets to include the random node `b`. The privacy leak here is non-trivial because all sensitive values sent from one party to another are masked by randomness and only by combining two received values (e.g. `(b12 from 1)` and `(s23 from 2)`) can party 3 cancel out the randomness and leak a sensitive value. The protocol is still correct (the outputs reconstructed from shares are the same as in the original protocol) but not private. Our algorithm will detect the privacy leaks introduced by such mistakes.

It would also be possible to get some information about the location of the potential privacy leaks. We can modify the algorithm in Fig. 3 to output at the end the set $\{v \in S\colon \mathsf{isSensitive}[v] = \mathsf{true}\}$, which is non-empty if the privacy check fails. It contains the nodes whose values are sensitive and are sent to the adversary but for which the algorithm could not determine that they are masked by enough independent randomness. We can also modify the compiler to connect the nodes in the arithmetic circuit to variables in the DSL code. Then, in the DSL example where we omitted the variable `b`, the algorithm would flag the variable `s32` when checking against party 2, `s23` against party 3, and the protocol would be private against party 1 (because the omitted random variable `b` belongs to that party).

## VII. Privacy and Security of Full SMC Protocols

The notion of privacy as defined in this paper, checked by the proposed algorithm, and preserved by composition, applies to protocols that receive their inputs in secret-shared form and produce their outputs in the same form. Clearly, it also applies to the initial sharing step that the input parties perform to their inputs. It does *not* apply to the reconstruction step that is necessary for the output parties to learn their respective outputs. As this reconstruction step is part of any useful SMC application, we have to consider how to deploy a set of actively private protocols working on shared data, in order to reap the most benefits from the privacy property they satisfy.

In principle, there are two different approaches to deal with the privacy leak that may come with the reconstruction. The first approach is to ignore it. If the output of the computation is $\ell$ bits long, and this output is made available to the adversary, then the adversary is able to learn $\ell$ bits about the inputs. This is true even for the passive adversary, but an active adversary may be able to influence, which bits it learns. Whether this approach is acceptable or not, depends on the application. It may be acceptable if the size of the input to SMC is large, the size of the output is small (e.g. it is the result of running a Boolean query) and the leakage of a small number of bits about the input is deemed acceptable. In this case, actively private protocols ensure a much smaller leakage to an active adversary than protocols that are only passively secure.

The second approach to deal with the privacy leak is to verify the behaviour of the computing parties after the computation but before the reconstruction of outputs. In such model of *consistent computations* [14], the detection of misbehaviours [12], [13] means that the outputs are not reconstructed, thus also not learned by the adversary. Depending on the methods of verification, the adversary is going to learn either nothing about the inputs, or at most one bit (whether outputs were consistent with the inputs or not). Such verification, performed after the computation, can be much cheaper than actively secure protocols which ensure the correct behaviour of the parties throughout the computation.

Let us now consider the case where we need to execute several SMC protocols (or one protocol several times), e.g. make multiple queries on a secret-shared database. The composability described in Sec. V-F applies to the part of the protocol before output reconstruction. Thus, if we do not need the output of the previous protocol to decide which protocol to execute next (the outputs of previous protocols may still be used to determine the parameters of the next protocol) then we can combine all the protocols into a single protocol and reconstruct all outputs only after the whole combined protocol has been executed. In this case, we can still leak only one bit.

If we need to reconstruct the intermediate outputs, e.g. if there is a human who uses the results of the previous queries to decide which queries to make next, then we may leak more than one bit. One way to limit the leakage is to block further execution of queries after there has been a certain number of protocol failures (active attacks that change outputs). If we need to execute $n$ queries and allow at most $k$ failures then the number of possible outcomes for the adversary is not more than $(n+k)^k$ because it learns only the numbers of the queries that failed. Thus no more than $k \log_2(n+k)$ bits are leaked. If $k$ is constant then the adversary needs to actively attack an exponential number of queries to learn a certain number of bits, thus the attack would not be very practical.

## VIII. Implementation

We have implemented the algorithm and it can prove privacy of several useful protocols on additively shared secrets, including all protocols in [16]. The protocols are used as building blocks in the Sharemind framework [22], [23], [16].

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, but the algorithm (and its implementation) is actually general enough to be used for any number of parties, of which less than half are 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 $\mathsf{numRandoms}$) to encode the sets $\mathsf{leakRandoms}[v]$ and $\mathsf{usableRandoms}[v]$. 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 $N = \mathsf{numVertices}$ and $R = \mathsf{numRandoms}$. 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 $i \in \mathsf{leakRandoms}[v]$ and the number of vertices $v \in S$ for which $i \in \mathsf{usableRandoms}[v]$.

The largest circuit we tested (corresponding to a private integer division protocol for 64 bits) had $N = 102709$ and $R = 4952$. Three runs of the algorithm (one against each of three parties as an adversary) on this circuit took a total of 7.7 seconds on a 2.50 GHz laptop.

While in this paper we consider only integer protocols, the algorithm can also be extended to protocols on other data types that can be secret shared using reversible operations. Thus we have also used the implemented algorithm to prove the privacy of protocols on secret-shared floating-point numbers (which consist of a sign bit, significand, and exponent as separate secret-shared integers). The reversible operations supported by the implementation are addition, subtraction, negation, bitwise exclusive or, and bitwise not.

We have used the algorithm to prove the privacy of all primitive protocols on secret-shared integers and floating-point numbers that we tried, after they were correctly implemented. The protocols include arithmetic operations, comparison, bitwise operations, shifts, square root, and some others, including some faster protocols for the cases where only one of the arguments is secret-shared and the other is public.

If a protocol implementation contains a bug that introduces a privacy leak then the algorithm fails to prove the privacy. Thus the algorithm can be used by the programmer to detect such bugs, similarly to a static type checker.

Although there exist protocols that are private but are rejected by our algorithm, this incompleteness has never been an issue. We do not use, nor do we foresee using constructions that are private, but cannot be handled by our checker after straightforward protocol optimizations.

## IX. Related Work

Our privacy checker is a solid example of the sequence-of-games approach to cryptographic proofs [24], the examples of which are many [25], [26], [27], [28]. 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 it is computed. After all these modifications, the view 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 [29].

Our analysis tracks, which values are masked by which random values using reversible operations. There are some similarities to [30]; they can even claim completeness for a certain verification procedure, but applied to a much simpler and more regular language. In [31], 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 [32]. 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 the ideal-real paradigm to give definitions of security and prove the composability of their security notions. In [33], it is shown how to actually construct universally composable secure multiparty computation protocols for all functions $f$. Compared to [32], we consider only active non-adaptive adversaries in the secure-channels setting, and we are only interested in privacy, not correctness.

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 [34], which also proves some completeness theorems in this setting. Composability for input private (against passive adversaries) protocols on shares is considered in [11]; their treatment is also applicable to active adversaries. We prove composability not for the actual security property (Def. 8) but rather for the property (Def. 9) verified by the algorithm (which implies the security property).

It has also been proposed to use a type system for secure computation [35]. They consider only two-party protocols in the semi-honest model. They require all values that are sent to another party to be explicitly made random using a re-randomize operation. A value derived from the same randomness cannot be sent out more than once because it is zeroed after sending. In the three-party multiplication protocol in our Fig. 1, a party sends two values derived from the same randomness to both of the other parties (one to each party). Thus it would be difficult to extend the type system to the case of three (or more) parties.

On the other hand, it seems possible to adapt our algorithm to the two-party protocols considered in [35], which use both secret sharing and homomorphic encryption. The re-randomize operation for ciphertexts (multiplication by an encryption of a random value) can be considered as a reversible operation, with a slight complication that it is reversible only by another party (the one who has the private key). Thus, although the algorithm was designed for the Sharemind protocol set (three-party protocols using additive secret sharing), it may be useful also for some other SMC frameworks.

Compared to [35], our algorithm provides type inference (the programmer does not have to specify which values are used for re-randomization, the algorithm can automatically infer this), is fully static (no need to zero sent values at run time), and allows using the same randomness to hide information from two different non-colluding parties (which is needed for three-party secret-shared protocols). Our algorithm can be considered as a combined type-inference and type-checking algorithm for an implicit type system. This type system would be too complex and artificial to write out.

## X. Conclusions

We have designed and implemented an algorithm that succeeds in proving the input privacy of all tried 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 on shares composed from the building blocks would also have input privacy and composed SMC programs that also combine shares to reveal some values leak only a small number of bits compared to the communication volume.

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.

## XI. Acknowledgment

This research has been supported by Estonian Research Council through grant IUT27-1, and by ERDF through STACC and EXCS.

## References

[1] D. Bogdanov, L. Kamm, S. Laur, and P. Pruulmann-Vengerfeldt, "Secure multi-party data analysis: end user validation and practical experiments," Cryptology ePrint Archive, Report 2013/826, 2013, http://eprint.iacr.org/.

[2] O. Goldreich, S. Micali, and A. Wigderson, "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority," in *STOC*. ACM, 1987, pp. 218–229.

[3] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay—a secure two-party computation system," in *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2004, pp. 20–20.

[4] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen, "Asynchronous Multiparty Computation: Theory and Implementation," in *Public Key Cryptography*, ser. Lecture Notes in Computer Science, S. Jarecki and G. Tsudik, Eds., vol. 5443. Springer, 2009, pp. 160–179.

[5] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "TASTY: tool for automating secure two-party computations," in *CCS '10: Proceedings of the 17th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2010, pp. 451–462.

[6] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A framework for fast privacy-preserving computations," in *ESORICS*, ser. Lecture Notes in Computer Science, S. Jajodia and J. López, Eds., vol. 5283. Springer, 2008, pp. 192–206.

[7] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos, "SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics," in *USENIX Security Symposium*, Washington, DC, USA, 2010, pp. 223–239.

[8] L. Malka, "VMCrypt: modular software architecture for scalable secure computation," in *ACM Conference on Computer and Communications Security*, Y. Chen, G. Danezis, and V. Shmatikov, Eds. ACM, 2011, pp. 715–724.

[9] B. Kreuter, abhi shelat, and C. hao Shen, "Billion-Gate Secure Computation with Malicious Adversaries," Cryptology ePrint Archive, Report 2012/179, 2012.

[10] Y. Zhang, A. Steele, and M. Blanton, "PICCO: a general-purpose compiler for private distributed computation," in *ACM Conference on Computer and Communications Security*, A.-R. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM, 2013, pp. 813–826.

[11] D. Bogdanov, P. Laud, S. Laur, and P. Pullonen, "From Input Private to Universally Composable Secure Multiparty Computation Primitives," in *Proceedings of the 27th IEEE Computer Security Foundations Symposium, CSF 2014*, 2014.

[12] I. Damgård, M. Geisler, and J. B. Nielsen, "From passive to covert security at low cost," in *TCC*, ser. Lecture Notes in Computer Science, D. Micciancio, Ed., vol. 5978. Springer, 2010, pp. 128–145.

[13] P. Laud and A. Pankova, "Verifiable computation in multiparty protocols with honest majority," Cryptology ePrint Archive, Report 2014/060, 2014, http://eprint.iacr.org/.

[14] S. Laur and H. Lipmaa, "On the Feasibility of Consistent Computations," in *Public Key Cryptography*, ser. Lecture Notes in Computer Science, P. Q. Nguyen and D. Pointcheval, Eds., vol. 6056. Springer, 2010, pp. 88–106.

[15] R. Gennaro, M. O. Rabin, and T. Rabin, "Simplified VSS and Fast-Track Multiparty Computations with Applications to Threshold Cryptography," in *PODC*, 1998, pp. 101–111.

[16] D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson, "High-performance secure multi-party computation for data mining applications," *Int. J. Inf. Sec.*, vol. 11, no. 6, pp. 403–418, 2012.

[17] R. Jagomägis, "SecreC: a Privacy-Aware Programming Language with Applications in Data Mining," Master's thesis, Institute of Computer Science, University of Tartu, 2010.

[18] P. Laud, A. Pankova, M. Pettai, and J. Randmets, "Specifying Sharemind's arithmetic black box," in *PETShop'13, Proceedings of the 2013 ACM Workshop on Language Support for Privacy-Enhancing Technologies, Co-located with CCS 2013*, M. Franz, A. Holzer, R. Majumdar, B. Parno, and H. Veith, Eds. ACM, 2013, pp. 19–22.

[19] I. Damgård, M. Fitzi, E. Kiltz, J. Nielsen, and T. Toft, "Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation," in *Proceedings of The Third Theory of Cryptography Conference, TCC 2006*, ser. LNCS, vol. 3876. Springer, 2006.

[20] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *FOCS*. IEEE Computer Society, 2001, pp. 136–145.

[21] O. Goldreich, *Foundations of Cryptography: Volume 2, Basic Applications*. New York, NY, USA: Cambridge University Press, 2004.

[22] D. Bogdanov, "Sharemind: programmable secure computations with practical applications," Ph.D. dissertation, University of Tartu, February 2013.

[23] D. Bogdanov, R. Jagomägis, and S. Laur, "A Universal Toolkit for Cryptographically Secure Privacy-Preserving Data Mining," in *Intelligence and Security Informatics - Pacific Asia Workshop, PAISI'12, Kuala Lumpur, Malaysia, May 29, 2012. Proceedings*, ser. Lecture Notes in Computer Science, M. Chau, G. A. Wang, W. T. Yue, and H. Chen, Eds., vol. 7299. Springer, 2012, pp. 112–126.

[24] M. Bellare and P. Rogaway, "The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs," in *EUROCRYPT*, ser. Lecture Notes in Computer Science, S. Vaudenay, Ed., vol. 4004. Springer, 2006, pp. 409–426.

[25] P. Laud, "Symmetric Encryption in Automatic Analyses for Confidentiality against Active Adversaries," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2004, pp. 71–85.

[26] B. Blanchet, "A Computationally Sound Mechanized Prover for Security Protocols," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2006, pp. 140–154.

[27] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-Aided Security Proofs for the Working Cryptographer," in *CRYPTO*, ser. Lecture Notes in Computer Science, P. Rogaway, Ed., vol. 6841. Springer, 2011, pp. 71–90.

[28] G. Barthe, D. Pointcheval, and S. Z. Béguelin, "Verified security of redundancy-free encryption from Rabin and RSA," in *ACM Conference on Computer and Communications Security*, T. Yu, G. Danezis, and V. D. Gligor, Eds. ACM, 2012, pp. 724–735.

[29] P. Laud and V. Vene, "A Type System for Computationally Secure Information Flow," in *FCT*, ser. Lecture Notes in Computer Science, M. Liskiewicz and R. Reischuk, Eds., vol. 3623. Springer, 2005, pp. 365–377.

[30] C. S. Jutla and A. Roy, "Decision Procedures for Simulatability," in *ESORICS*, ser. Lecture Notes in Computer Science, S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459. Springer, 2012, pp. 573–590.

[31] A. Moss, E. Oswald, D. Page, and M. Tunstall, "Compiler Assisted Masking," in *CHES*, ser. Lecture Notes in Computer Science, E. Prouff and P. Schaumont, Eds., vol. 7428. Springer, 2012, pp. 58–75.

[32] R. Canetti, "Security and Composition of Multiparty Cryptographic Protocols," *J. Cryptology*, vol. 13, no. 1, pp. 143–202, 2000.

[33] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai, "Universally composable two-party and multi-party secure computation," in *STOC*, J. H. Reif, Ed. ACM, 2002, pp. 494–503.

[34] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract)," in *STOC*. ACM, 1988, pp. 1–10.

[35] F. Kerschbaum, "An Information-Flow Type-System for Mixed Protocol Secure Computation," in *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, K. Chen, Q. Xie, W. Qiu, N. Li, and W. Tzeng, Eds. ACM, 2013, pp. 393–404. [Online]. Available: http://doi.acm.org/10.1145/2484313.2484364