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

Martin Pettai
joint work with Peeter Laud
University of Tartu / Cybernetica AS / STACC

May 18, 2014

Introduction: SMC

- In a secure multiparty computation (SMC) problem:
  - There are n parties
  - Each party P_i provides an input x_i and expects to learn y_i
  - $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$ for some publicly known function f
  - 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).
- 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
  1. the protocol preserves privacy by not letting the adversarial parties to learn anything they could not learn through the interaction with the ideal functionality, and
  2. the protocol delivers the correct answer to all non-adversarial parties
- Recent results have given value to the studies of privacy independently of security

Introduction: Sharemind

- We have created an algorithm for checking privacy
- Our privacy checker is targeted towards SMC protocols that aim to provide information-theoretic privacy against adversarial parties
- Typically, such protocols use secret sharing to represent the intermediate values during the computation
- The checker is integrated with our toolchain for compiling and maintaining the SMC protocols for Sharemind
  - Using it, the protocols are first specified in a high-level declarative language, often including simpler protocols as subroutines
  - The specification is compiled to an intermediate representation (IR), analyzed and optimized
  - From the IR, code in C++ is generated and compiled together with the rest of the Sharemind system
  - Our privacy checker works on the IR, which is highly suitable for such analyses

Introduction: composability

- The Sharemind platform allows privacy-preserving applications to be specified as a composition of SMC protocols
- 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

Introduction: benefits

- The benefits of our privacy checker are the most apparent in developing, extending and maintaining large sets of SMC protocols on additively shared values
  - Currently, Sharemind employs more than 100 different protocols for various arithmetic, relational, and database operations with shared values
- In this setting it gives us guarantees that the specified protocols are private and the compilation (up to the intermediate representation) and the optimizations do not destroy this property
- It is infeasible to obtain such guarantees in a way that does not involve significant automation — the protocol set is large, the optimizations 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)

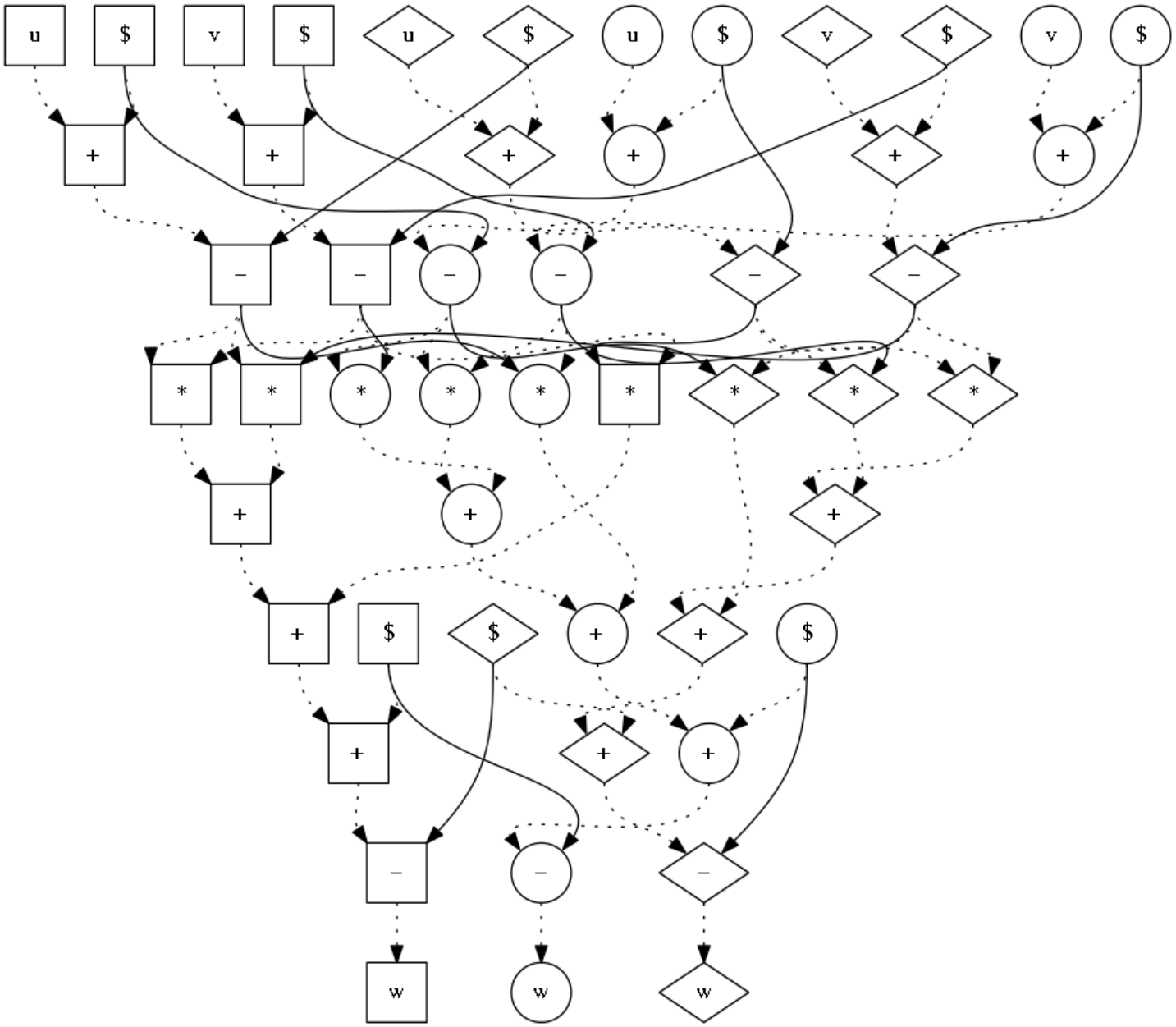
The intermediate representation (1)

- The intermediate representation of our protocols, also used by our privacy checker, is an arithmetic circuit
- The nodes have an extra attribute — the identity of the party executing it

The intermediate representation (2)

- In the next slide, there is an arithmetic circuit corresponding to a multiplication protocol
- It 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
- Nodes labeled with u and v denote the input nodes for parties
- Nodes labeled with w denote the outputs
- Nodes labeled \$ denote the generation of a random element of $\mathbb{Z}_N$
- The shape of the node denotes the party which calculates its value
- Communication is depicted in by drawing the edges corresponding to message sends with solid lines, while local dependencies are drawn with dotted lines

An example: multiplication protocol



Active adversary

- There is an adversary $\mathcal{A}$ that controls a subset of the parties executing the protocol
- $\mathcal{A}$ tries to learn something about the private inputs of other parties (the shares of u and v in the previous slide)
- We consider the case where $\mathcal{A}$ 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 of messages as in the original protocol
- Thus the subgraph of the circuit induced by the vertices of the parties controlled by $\mathcal{A}$ may be replaced with a black box that has the appropriate number of incoming and outgoing edges

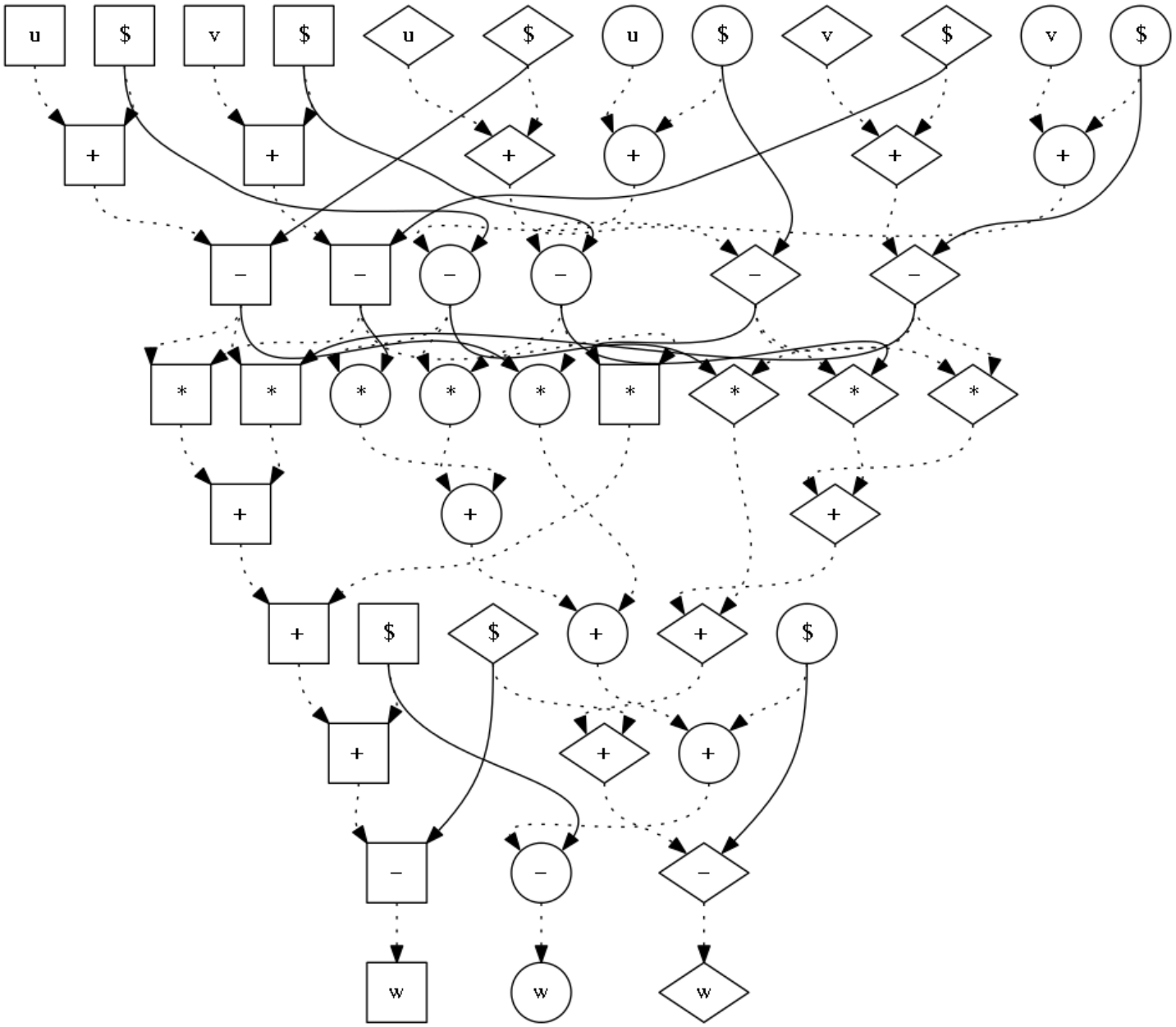
Aadags (1)

- 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 value of the adversarial sink may be thought as a tuple of all values sent to the adversary
- 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

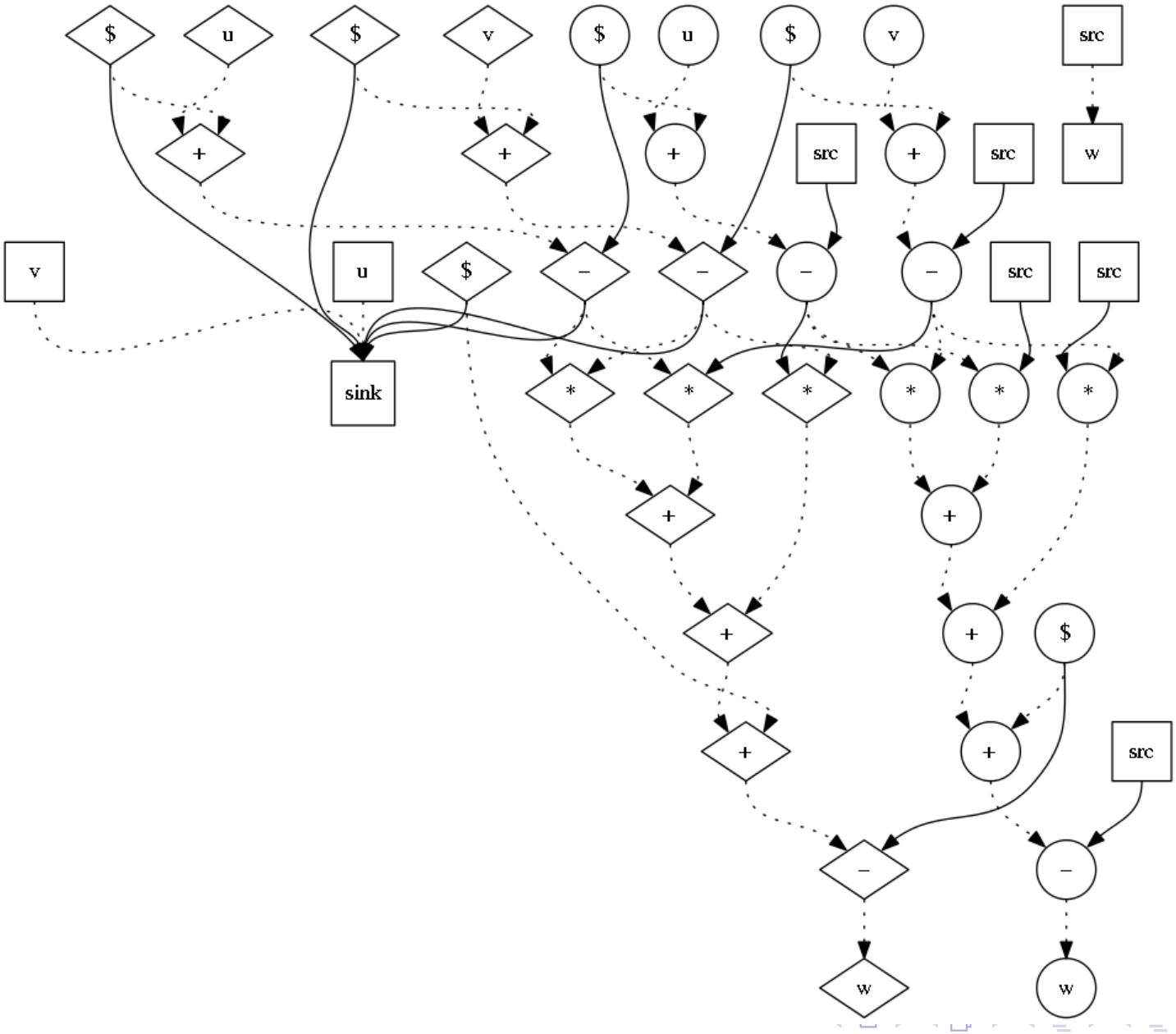
Aadags (2)

- 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)
- 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 sink and this vertex has no outgoing edges. We call such a vertex the *adversarial sink*
  2. Every vertex of G that has the operation src 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*

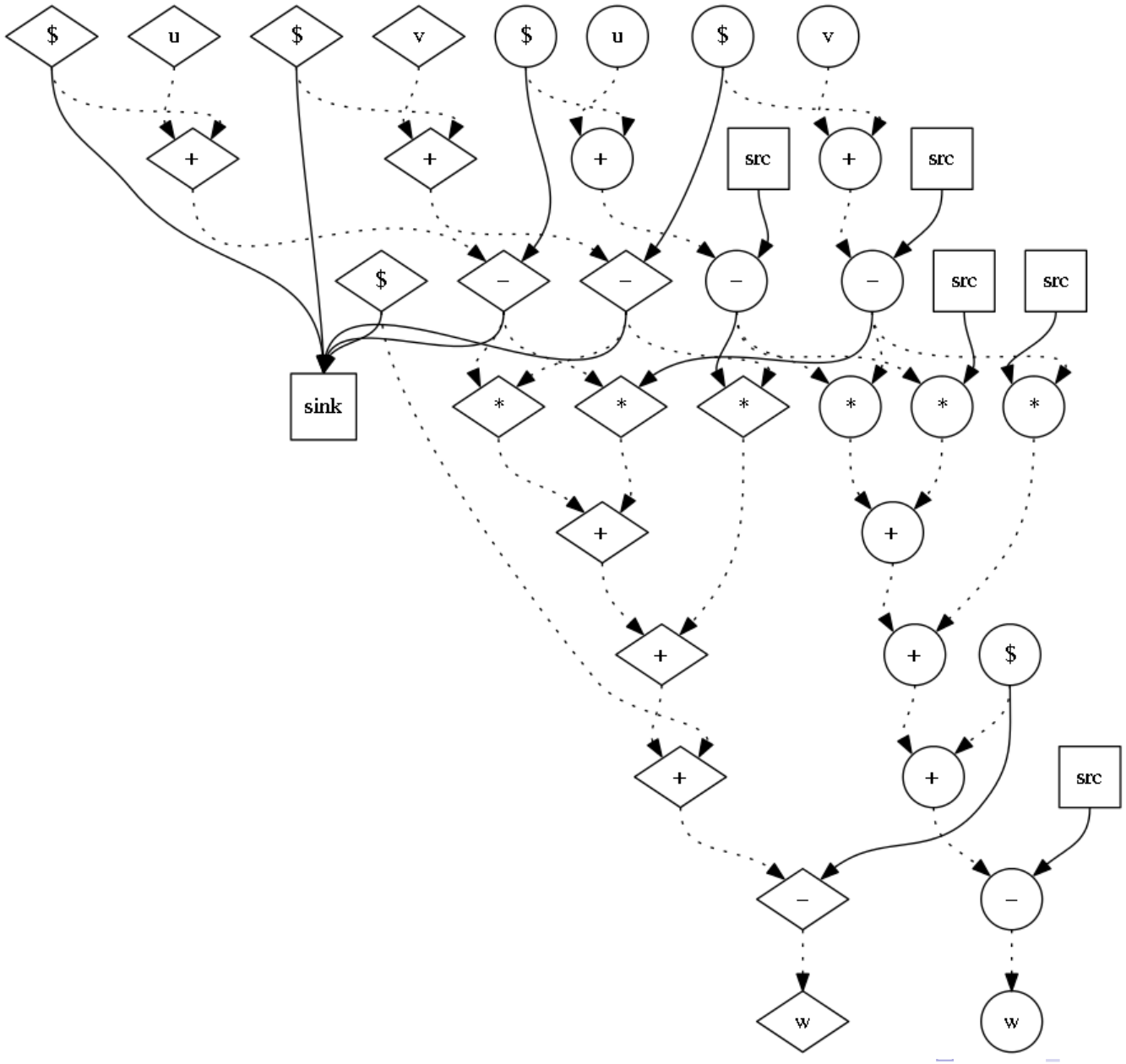
An example: multiplication protocol



An example: multiplication protocol as aadag (1)



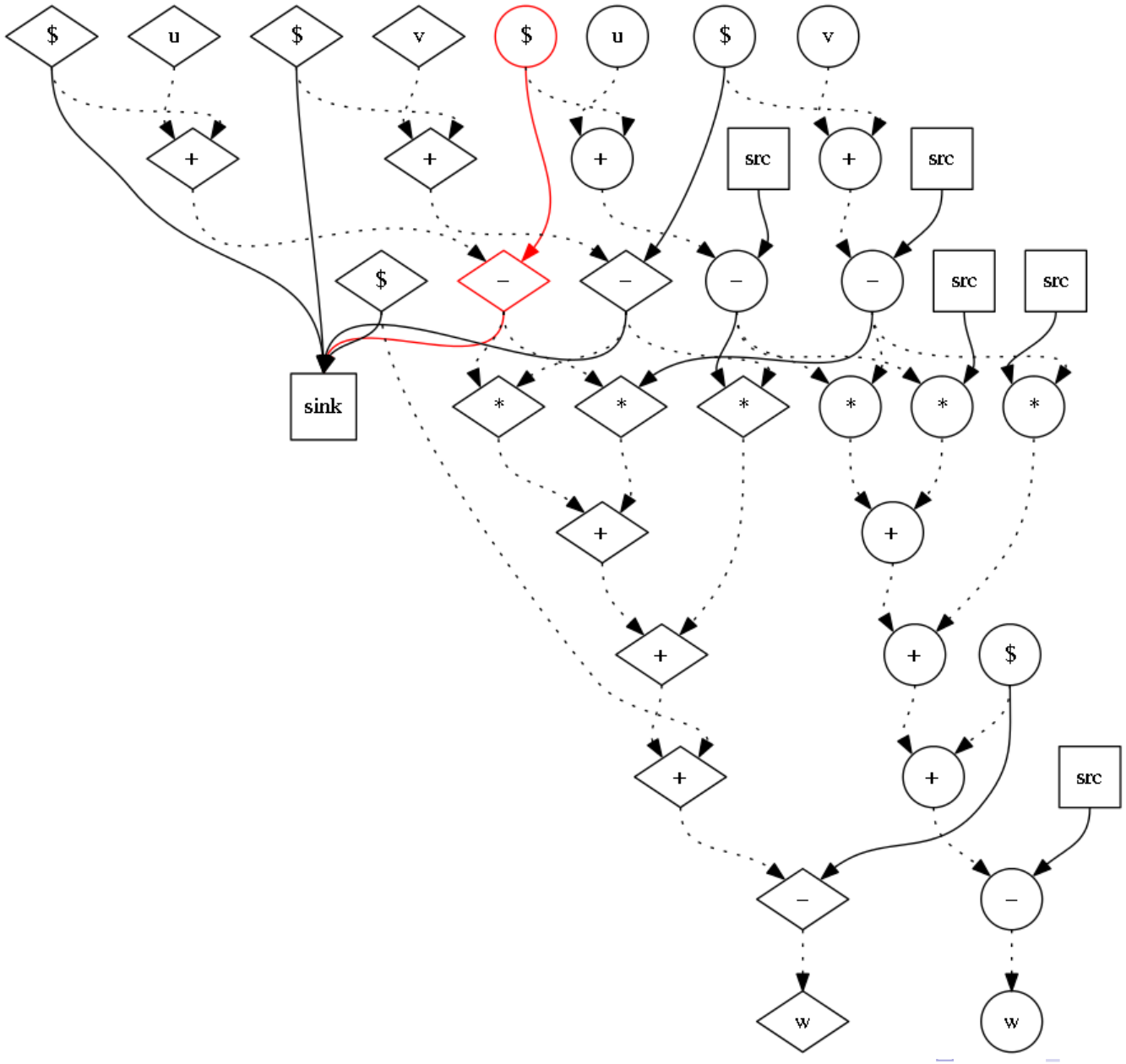
An example: multiplication protocol as aadag (2)



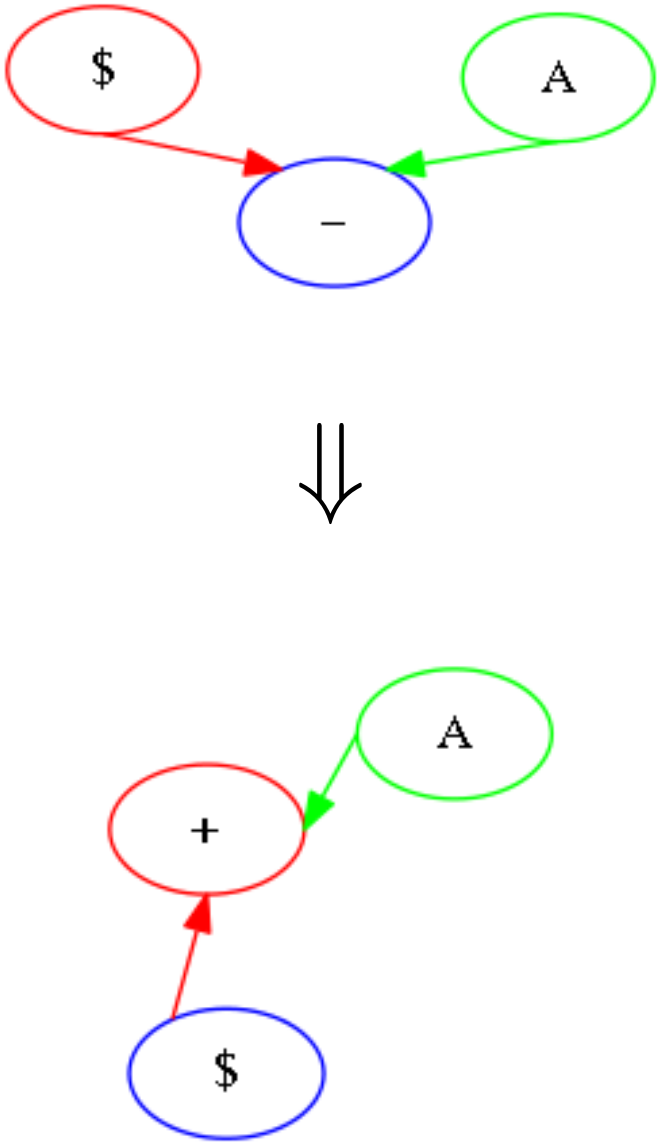
The idea of our algorithm

- We want to prove that the protocol is actively private, i.e. that the adversary does not learn anything about the private inputs of other parties while the aadag is executed
- We may assume the adversary to be deterministic, any necessary randomness can be included in the adversary
- If all nodes whose values are sent to $\mathcal{A}$ are random then privacy is obvious
- Our idea is to transform the aadag so that the nodes whose value is sent to $\mathcal{A}$ become random but the protocol (i.e. the value of each node) does not change
  - The order or timing of the calculation of each value is allowed to change
  - The values that are sent to $\mathcal{A}$ must be calculated at the same time or earlier than before
- We try to find unique paths from a random vertex to the sink where every operation on the path is reversible

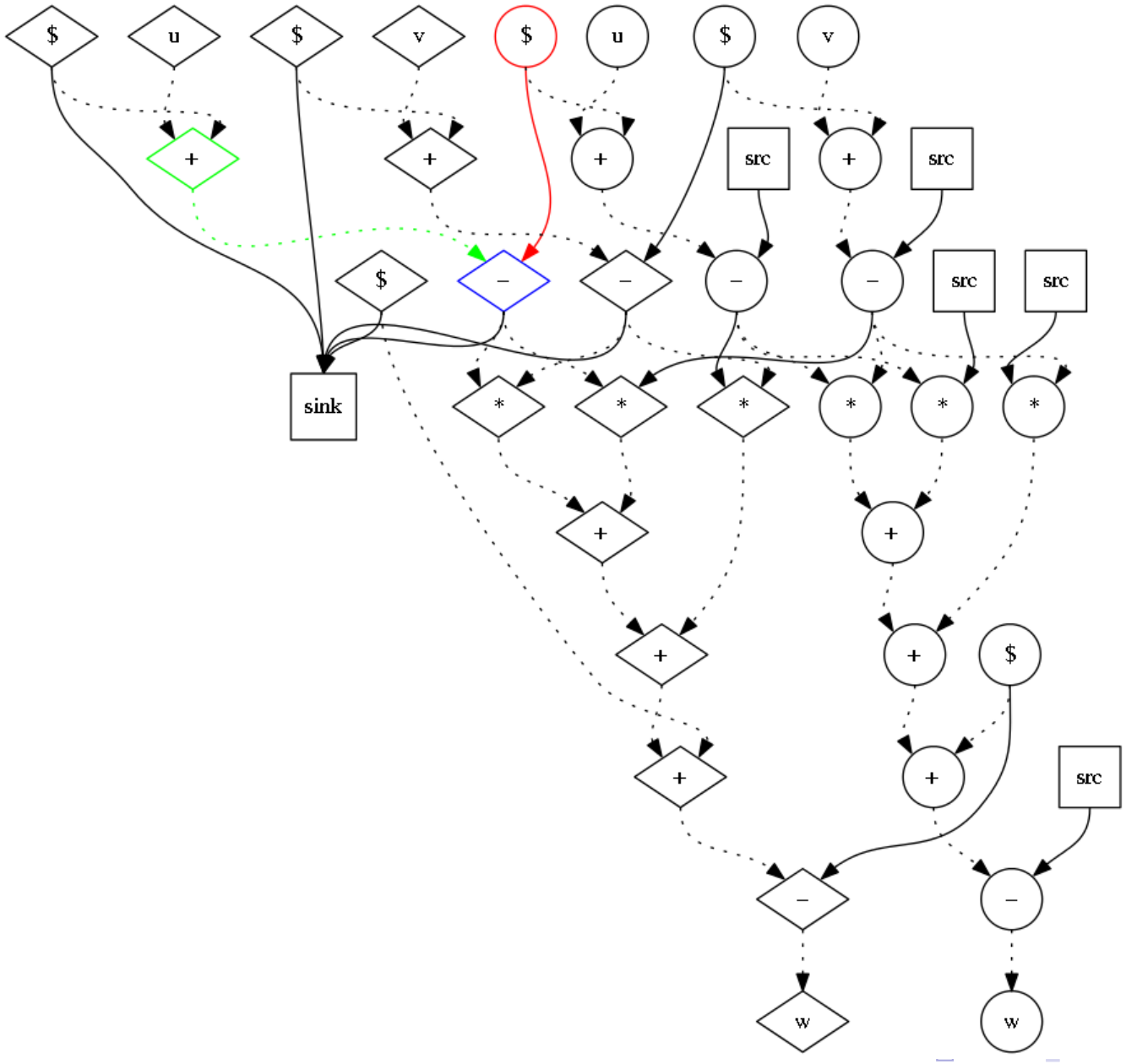
An example: multiplication protocol as aadag (3)



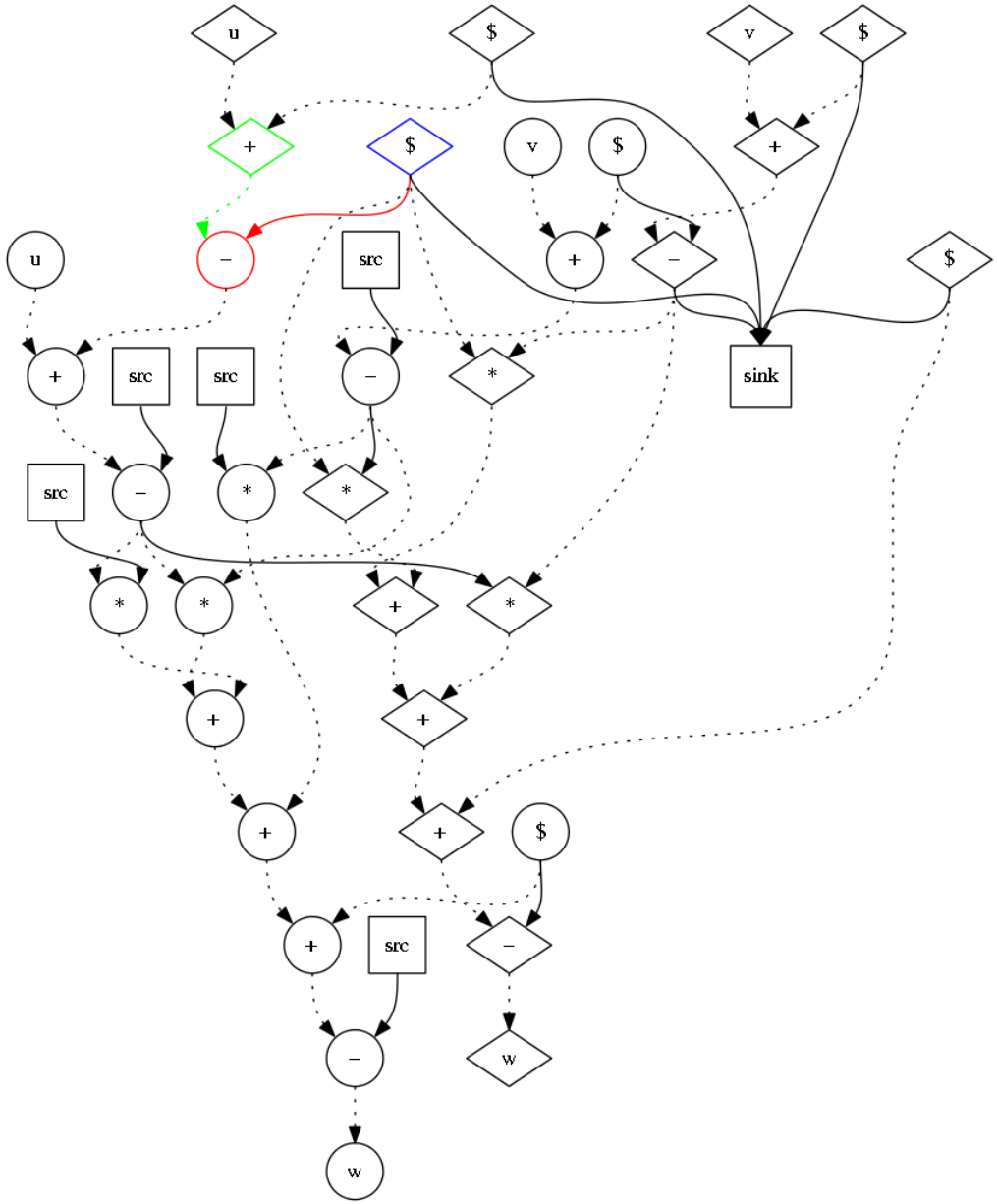
The transformation step (1)



An example: multiplication protocol as aadag (4)



An example: multiplication protocol as aadag (5)



Path transformation

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 a random vertex v_0 to the vertex $v = v_n$
- The operation of v_0 is random
- The operations of $v_1, \ldots, v_n$ are reversible and not sink
- All paths from the random vertex v_0 to the adversarial sink in G go through the vertex v

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

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

The transformation step (2)

Suppose all of the following hold in an aadag G:

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

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

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

The transformation step (3)

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

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

Global variables used by the algorithm (1)

These are part of the input to the algorithm (the circuit itself is given implicitly):

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

Global variables used by the algorithm (2)

These are filled by the **for each** loop:

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

The algorithm (1)

for each vertex v (in topological order of vertices) **do**
  if $v \in A$ **then**
    $\text{isSensitive}[v] \leftarrow \text{false}$
    $\text{leakRandoms}[v] \leftarrow \emptyset$
    $\text{usableRandoms}[v] \leftarrow \emptyset$
  else if $\lambda_G(v) = \text{input}$ **then**
    $\text{isSensitive}[v] \leftarrow \text{true}$
    $\text{leakRandoms}[v] \leftarrow \emptyset$
    $\text{usableRandoms}[v] \leftarrow \emptyset$
  else if $\lambda_G(v) = \text{random}$ **then**
    $\text{isSensitive}[v] \leftarrow \text{false}$
    $\text{leakRandoms}[v] \leftarrow \emptyset$
    $\text{usableRandoms}[v] \leftarrow \{i\}$, where $i = \text{randomIndex}(v)$

The algorithm (2)

else if $\lambda_G(v) \in$ **OpR then**

Let $v_1 \cdots v_k = \mathsf{pr}_G(v)$

$\text{isSensitive}[v] \leftarrow \bigvee_{i=1}^{k} \text{isSensitive}[v_i]$

$\text{leakRandoms}[v] \leftarrow$

$\mathsf{thr}_{\geq 2}(\text{usableRandoms}[v_1], \ldots, \text{usableRandoms}[v_k]) \cup \bigcup_{i=1}^{k} \text{leakRandoms}[v_i]$

$\text{usableRandoms}[v] \leftarrow$

$\mathsf{thr}_1(\text{usableRandoms}[v_1], \ldots, \text{usableRandoms}[v_k]) \setminus \text{leakRandoms}[v]$

else

Let $v_1 \cdots v_k = \mathsf{pr}_G(v)$

$\text{isSensitive}[v] \leftarrow \bigvee_{i=1}^{k} \text{isSensitive}[v_i]$

$\text{leakRandoms}[v] \leftarrow$

$\bigcup_{i=1}^{k} \text{leakRandoms}[v_i] \cup \bigcup_{i=1}^{k} \text{usableRandoms}[v_i]$

$\text{usableRandoms}[v] \leftarrow \emptyset$

end for

The algorithm (3)

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
$i \in \mathrm{usableRandoms}[v] \wedge$
for each vertex $w \in S$ different from v:
$i \notin \mathrm{leakRandoms}[w] \wedge i \notin \mathrm{usableRandoms}[w]$
do
Modify the aadag into a simulating aadag where the operation of v is random (as described before)
Remove v from S
end while

The algorithm (4)

if for each vertex $v \in S$: $\mathrm{isSensitive}[v] = \mathrm{false}$ **then**
    exit(the protocol is private)
else
    exit(cannot prove that the protocol is private)

- We have proved the soundness of the algorithm, i.e. that if it returns *the protocol is private* then the protocol is actively private
- We have also proved that the property verified by the algorithm is composable
- Thus, if we have verified the basic protocols with our algorithm then we can use them as subprotocols and the larger protocol is still actively private

Implementation

- We have implemented the algorithm
- It can prove privacy of several useful protocols on additively shared secrets
  - These protocols are used as building blocks in the Sharemind framework
- The time complexity of the algorithm (with some optimizations) is $O(N \cdot R)$, where $N = \mathrm{numVertices}$ and $R = \mathrm{numRandoms}$
  - The implementation 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 largest circuit we tested (corresponding to a private shift-right protocol for 64 bits) had $N = 44039$ and $R = 3428$. Three runs of the algorithm (one against each of three parties as an adversary) on this circuit took a total of 2.5 seconds on a 2.50 GHz laptop

Conclusion

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

The End