International Journal of Information Security (2019) 18:505–518
https://doi.org/10.1007/s10207-018-0424-2

REGULAR CONTRIBUTION

Breaking MPC implementations through compression

João S. Resende[1] · Patrícia R. Sousa[1] · Rolando Martins[1] · Luís Antunes[1]

Published online: 2 January 2019

Abstract
There are many cryptographic protocols in the literature that are scientifically and mathematically sound. By extension, cryptography today seeks to respond to numerous properties of the communication process beyond confidentiality (secrecy), such as integrity, authenticity, and anonymity. In addition to the theoretical evidence, implementations must be equally secure. Due to the ever-increasing intrusion from governments and other groups, citizens are now seeking alternatives ways of communication that do not leak information. In this paper, we analyze multiparty computation (MPC), which is a sub-field of cryptography with the goal of creating methods for parties to jointly compute a function over their inputs while keeping those inputs private. This is a very useful method that can be used, for example, to carry out computations on anonymous data without having to leak that data. Thus, due to the importance of confidentiality in this type of technique, we analyze active and passive attacks using complexity measures (compression and entropy). We start by obtaining network traces and syscalls, then we analyze them using compression and entropy techniques. Finally, we cluster the traces and syscalls using standard clustering techniques. This approach does not need any deep specific knowledge of the implementations being analyzed. This paper presents a security analysis for four MPC frameworks, where three were identified as insecure. These insecure libraries leak information about the inputs provided by each party of the communication. Additionally, we have detected, through a careful analysis of its source code, that SPDZ-2's secret sharing schema always produces the same results.

Keywords Multiparty computation · Normalized compression distance · Entropy · Secret sharing · Privacy · Cryptography

1 Introduction

In the process of designing cryptographic protocols, it first has to be proposed and submitted with associated theoretical proof, to provide evidences that the protocol is secure under a predefined set of assumptions. However, these proofs do not provide any guarantees regarding practical implementations, thus it is necessary to demonstrate that the theoretical model, that supports the protocol, matches the technical security assumptions made in an actual implementation. This task is hard and normally relies on domain-specific knowledge, as such novel ways are needed to test the robustness and security of crypto protocols implementations.

There is a vast literature on theoretical security proofs of cryptographic protocols that are poorly implemented that lead to risks associated with the exposure of private, anonymous and sensitive data. Due to the importance of these security issues, there has been a growing interested on analysis and detection of such errors in applications and libraries of such protocols [1–4].

Communications are one of the first factors that we must protect in cyberspace. There are cryptographic protocols for various types of communications; one of the most interesting is to protect each person's personal information, even when both parties want to make computations using anonymized data. The well-known millionaire's problem is an example, where both parties want to know whom is richer but without knowing the salary of each other. In this case, MPC protocol can be used and consists of two or more parties, where each party has their own secret input. MPC computes some joint function f that receives as input each party's secret data (in the case of the millionaire's problem, the function

✉ João S. Resende
jresende@dcc.fc.up.pt

Patrícia R. Sousa
psousa@dcc.fc.up.pt

Rolando Martins
rmartins@dcc.fc.up.pt

Luís Antunes
lfa@dcc.fc.up.pt

1 University of Porto, R.Campo Alegre,1021/1055, 4169-007 Porto, Portugal

is something like: $f = input1 \leq input2$, where *input*1 and *input*2 corresponds to the salary of each party). Neither party obtains any information on the inputs of the other. At the end of the protocol, each party receives the result of the function, without the inputs being revealed [5].

General MPC algorithms are inefficient in practice with regards to communication complexity; most solutions rely on the existence of a synchronous network and are often not secure against dynamic adversaries [6]. For this reason, the main motivation focuses on the study of potentially (in)secure practical implementations, which do not guarantee that they comply with the security properties of the protocol, despite being usually written by expert cryptographers.

As there are a considerable number of libraries, with completely different implementations and technical assumptions, an automated method is necessary to validate these libraries without resorting to the analysis of the whole source code. In 2004/2005, Cilibrasi and Paul [7] introduced a new method of data clustering based on compression. Good results have already been obtained with this method in several domains, namely applied to internet traffic analysis [8], which is similar to what we intend to do. The method starts by determining the similarity (distance) between each pair of objects, computed from the length of the compressed data files, and then a hierarchical clustering method is applied. This method is very interesting as it is feature-free, there are no parameters to tune, and no domain-specific knowledge to use [9].

We argue that this method may help to reduce the time it takes for security vulnerability analysis. In fact, as the hierarchical clustering method can be quite slow, we also use standard clustering techniques to speedup this process. Generally, the analyses are performed by exploring the code to find security vulnerabilities, or by collecting information from pieces of data that the protocol sends to other devices. This tool has the ability to do the security vulnerability analysis automatically. Our analysis has been performed by collecting a set of program traces and applying techniques from (algorithmic) information theory to perform cluster based on the input. In case of changes in the protocol, we can also easily change and perform a new security analysis, instead of reanalyzing the whole code.

This work focuses on testing a relevant set of MPC protocols by using software packages and tools [10–14] which the research community is using to test and develop the MPC systems. We mainly focus on systems that use secret sharing, garbled circuits and/or oblivious transfer.

1.1 Contributions

This work has the following main contributions:

1. Setup and analyze the privacy security property of MPC in some of the most cited MPC frameworks: SPDZ-2, ABY, TinyLego and DUPLO;
2. Simulation of a passive attack using the *tcpdump* tool, which allows the adversary to "sniff" all traffic passing through the data network;
3. Simulation of an active attack through the use of *STrace*, which traces the execution of a particular program (or process), and intercepts and records the calls it makes to the system functions and their respective return signaling;
4. Evaluate the traces of both tools (for passive and active attacks) by using *CompLearn* and similar tools to apply compression techniques to the process of discovering and learning patterns, i.e., to identify the similarity between the objects. Then, apply clustering techniques in order to understand if it is possible to cluster the data, i.e., if it is possible to join communications with the same inputs in the same cluster;
5. Explore and analyze the source code of one of the libraries to understand the problem or implementation errors. We analyze the source code of the SPDZ-2 [15].
6. Explore and identify similar vulnerabilities on other MPC frameworks. This does not require the analysis of the theoretical model because we can cluster the information only by analyzing the information extracted from the network packets. We can produce results in other paradigms or some variants of MPC: secret sharing, garbled circuits and/or oblivious transfer.

Our work is useful to other researchers in both security and multiparty computation domains. The combination of the normalized compression distance (NCD) with the tools for obtaining network traces and system calls, as well as the combination of clustering techniques, provides a complete tool-set that can be used to analyze other frameworks that implement other cryptographic protocols. As already stated, it is important to know if there are errors that compromise security in cryptographic protocol implementations, as theory alone is not enough. This tool can aid this detection and find data leak problems, namely with MPC algorithms as explored in this work.

1.2 Related work

We begin with a description of the NCD and MPC preliminaries in Sect. 2. In Sect. 3, we describe the libraries' setup, and all the setups of the MPC problem and inputs. In Sect. 4, we present the attack overview with a description of the remote attacks (passive) and local attacks (active) with all the premises of each other. In Sect. 5, we present the results of these attacks. Finally, Sect. 6 presents the low-level code analysis of the SPDZ-2 framework, to identify the vulnerabilities of the protocol implementation.

1.3 Related work

There are already some approaches in the literature that address privacy computation and its adversaries. We will present some approaches related to the detection of network attacks and also some work that has been done in this area to contextualize and motivate the use of compressors, which have already been tested in other contexts.

Acar [2] presents analysis of some libraries that implement some cryptographic protocols. In this paper, it is analyzed the usability and the security of the library, by analyzing the features and documentation that also matter for security. This is an example of a work in the literature that makes tests of the libraries available on GitHub.

Orlandi [16] presented security models and the latest advances in multiparty computation in a paper titled "Is Multiparty Computation Any Good in Practice?". For an MPC protocol to be secure, some properties must be satisfied, such as input privacy and output correctness.

Wehner [8] showed how techniques based on Algorithmic Information Theory, also known as Kolmogorov complexity, can help in the analysis of internet worms and network traffic. Using compression, different species of worms can be clustered by type. Compression is also used to understand different types of network traffic and to help detect traffic anomalies. This technique could become a useful tool to detect new variations of an attack and thus help to prevent IDS evasion.

CompLearn [17] is an open-source implementation of NCD used for clustering and classification with a wide range of applications. Its creators originally demonstrated its application in: genomics, virology, languages, literature, music, character recognition and astronomy [7]. Subsequent work has applied it to plagiarism detection, image distinguishable, machine translation evaluation, database entity identification, detection of internet worms, malware phylogeny and malware classification, to name a few [18]. Borbely presented a work [18] with an example of the application of NCD for classifying malware.

2 Theoretical background

2.1 Normalized compression distance

NCD first proposed by Li et al. [19] is the real-world approach to the notion of normalized information distance (NID). Clustering by NCD is a practical implementation of the mathematical notion of Kolmogorov complexity. In the scope of statistical or clustering methods, it is important to measure the absolute information distance between individual objects. NID measures the minimal quantity of information sufficient to translate between two objects; however, it is also non-computable. NCD is a computable approximation of NID using standard compressors. These concepts can be combined in order to create a way of analyzing data (data mining) and to remove redundancy of objects, thereby allowing their agglomeration.

Some experiments regarding the impact of NCD in clustering [7] show that NCD is a (quasi-)universal similarity metric relative to a normal reference compressor. To apply NCD to the difference between files, we need to choose a standard compressor to make an approximation of the smallest representation of the program.

$$\mathrm{NCD}(x, y) = \frac{C(xy) - \min(C(x), C(y))}{\max(C(x), C(y))} \tag{1}$$

In Formula 1, we give as the input two different files (x and y). $C(xy)$ represents the size of the resultant file by the compression of the concatenation of x with y, and $C(x)$ and $C(y)$ represent the size of the compression of x and y, respectively.

NCD is a function that gives values in the interval $0 \leq r \leq 1+e$ representing how different the two files are. NCD closer to 0 represents more similar files, and results closer to 1 are more distinguishable files. The e is due to imperfections in the compression algorithms, but for most standard compression tests performed by Li et al. [19], it is unlikely to see an e above 0.1.

With NCD, we can compute the NCD matrix. This matrix allows us to compare and cluster files according to their similarity with other files in the dataset, depending on the NCD value.

2.2 Multiparty computation

MPC was formally introduced as secure two-party computation in 1982 [20–22].

Andrew Yao introduced the millionaires' problem in 1982, the seminal secure multiparty computation example/problem. The scenario consists of two parties whom are both interested in knowing which of them is richer without revealing their inputs (i.e., their actual wealth). In this scenario, each party uses respective inputs x and y denoting their salaries. The goal is to find the highest salary, without revealing their respective salaries. Mathematically, this can be achieved by computing:

$$f(x, y) = \max(x, y)$$

At the end of the protocol, each participant will get only the result of the function f, without getting anything else about the other party's input, i.e., the secret inputs will not be revealed.

This protocol has to ensure two main security properties:

Privacy The inputs are never revealed to other parties;

Correctness The output given at the end of the computation is correct.

These security guarantees are to be provided in the presence of adversarial behavior. There are two classic adversary models that are typically considered: semi-honest (where the adversary follows the protocol specification but may try to learn more than allowed from the protocol transcript) and malicious (where the adversary can run any arbitrary polynomial-time attack strategy) [23].

3 MPC libraries overview

For the security tests, the MPC frameworks presented in [24] were used. First, we chose the most cited libraries that in turn implemented the most cited protocol variants of secret sharing and garbled circuits: SPDZ-2 [10] and ABY [11]. Also, we chose two different libraries of the LEGO paradigm: TinyLEGO [13] and DUPLO [14] to see if there are problems in implementation, as DUPLO is descending from TinyLEGO with different paradigms.

In each library, it was used the Yao's millionaire's problem with two parties.

To simulate this situation in terms of input, we use the world's wealthiest tech billionaires according to the list compiled and published by Forbes [25]. The total net worth of each individual on the list is estimated, in US dollars, based on their assets and debt accounting.

3.1 SPDZ-2

SPDZ-2 is a software implementation for the SPDZ and MASCOT secure multiparty computation protocols. This implementation is open-source and is available on GitHub [15].

In the SPDZ-2 library, there are some examples of MPC programs. One of them was presented in the SPDZ tutorial at the TPMPC 2017 [24] workshop in Bristol. It is called *tpmpc_tutorial* and provides a millionaires problem. We choose this one because we want to use a well-known multiparty computation problem as Yao's millionaire's problem [20].

In this case, it is necessary to perform a real MPC computation—P1 shares a and P2 shares b—and reveal the comparison of the values.

The code of the millionaire's problem is provided in the (Code 1). $val = sint.get\_input\_from(i)$ is used to get input for party i and share secretly into val. This reads input for party i from *Player-Data/Private-Input-i* which is the location where the private input is placed when it is generated. The $s.reveal()$ function reveals the response 0 if the numbers are different and 1 if the numbers are equal. For these tests, we use the commit that has the SHA: *7d55d01010ca69c310d74272c5b991fbebe3a7b7*.

Code 1: SPDZ-2 implementations of Yao's Millionaire's Problem

```
alice = sint.get_raw_input_from(0)
bob = sint.get_raw_input_from(1)

b = alice < bob
print_ln('The richest is: %s', b.reveal())
```

3.2 ABY

ABY is a mixed-protocol framework that efficiently combines secure computation schemes based on arithmetic sharing, Boolean sharing and Yaos garbled circuits. It makes available the best practice solutions in secure two-party computation [11]. The implementation of this protocol is open-source and is available on GitHub [26].

This implementation includes the millionaire's problem in its source code on GitHub. Therefore, we only had to change the input of the millionaire's problem from the random input that is provided by default (Code 7 in "Appendix A"). The commit used in this work was: *c189ff6d45000890185e503bbfbec89914bb497f*.

3.3 TinyLEGO

TinyLEGO is a protocol for general secure two-party computation (2PC) [13], the first published implementation of the LEGO paradigm for 2PC. There is an open-source implementation available on GitHub [27] with a C++14 implementation. The code builds on the *SplitCommit* implementation of the [FJNT16].

To prepare the library for the millionaires problem, we must choose the circuit representations of common cryptographic functions that are typically used to benchmark MPC protocols. In the library, circuits are available from Bristol [28], and so we chose the "*Comparison 32-bit Unsigned LTEQ*" circuit which is in accordance with the focus of the millionaires problem. The circuit corresponds to the millionaires problem, since it does *LTEQ* which means *less than or equal*. We converted the values from the list of billionaires to 32-bit binary. For these tests, we use the commit *336e116e9f6db7430d3c56648d7fd9b7ed3e48f2*.

3.4 DUPLO

DUPLO is an approach for malicious secure two-party computation [14]. There is an implementation in C++ of this approach available on GitHub [29].

The preparation of the library for inputs and the millionaire's problem is the same as for TinyLEGO. In the same way, we chose the "*Comparison 32-bit Unsigned LTEQ*" circuit which is in accordance with the focus of the millionaire problem, because the circuits available for this library are the same as the ones provided in TinyLEGO by Bristol [28]. We also convert the values from the list of billionaires to 32-bit binary. We use the commit *0d14ad0a225d8f2355549d64eac6330ceba 1f47c* for these tests.

4 Attacks overview

In this section, we will describe two types of attacks that we will make on the frameworks: passive and active attacks. In the passive attack simulation, the attacker only needs to be in the same network layer as one of the victims, and in this way, be able to sniff the packets that pass through the network by *tcpdump*. However, we can also execute attacks without need of elevated privileges to the machine, just by having permissions to exchange the executable file (user permissions). For this, we perform an active attack, where the attacker needs to find the executable of the MPC program in the local file system of one of the players in the communications. Then, it has to replace this executable with a new one, where the attacker collects the information provided by the *STrace* tool. The use of these tools allows us to do a statistical analysis of the protocol, and to categorize the information that at a given moment is passing in the network. We use these two tools to demonstrate that there is leak of information in a local environment (a malicious user can tries to subvert the protocol through *STrace*). Alternatively, an attacker can be passive by only inspecting the contents of the packets that are observed with *tcpdump*.

We constructed a middleware system that consisted in two parts:

4.1 First phase: the attacker computes all the input's possibilities

The middleware stores all possible combinations of *tcpdump* captures. Then, it must apply a clustering task to sort and correlate the captures. For this, we choose a technique called NCD, which measures the similarity between the captures, generating a similarity matrix of $N \times N$ dimensions, where it compares all the elements with each other. So, now, the middleware has all the datasets generated (i.e., the similarities between the captures), and now, it needs to apply a clustering method (such as clear in data mining) to the similarity matrix that will generate the clusters using the table generated by the NCD.

In brief, the first phase consists in four steps:

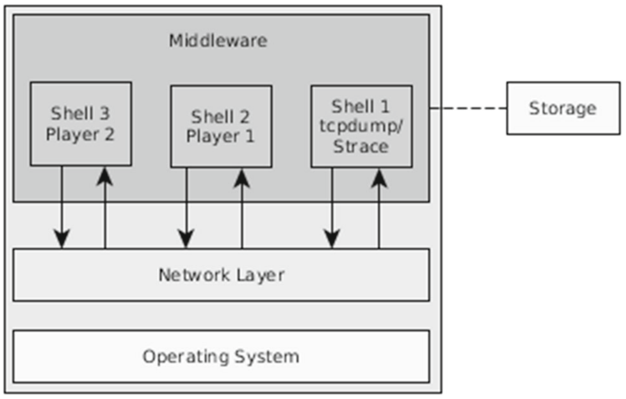

Fig. 1 First phase

1. Gather all possible inputs;
2. Generate *tcpdump* or *STrace* captures for all input combinations;
3. Compute NCD matrix;
4. Cluster aggregation based on NCD.

We can see in Fig. 1 the connection between the middleware and the network layer to gather the captures of all inputs.

4.2 Second phase: the attacker collects/compares real information

The middleware has the information of the previous phase, that is, all information clustered by inputs. It does not categorize any information regarding the identity of the person in the communication, but rather of their data.

In this case, the middleware needs to receive or collect real information through captures (*STrace* or *tcpdump*) and compare with all the captures it has stored, so it can cluster the data using the *clara* algorithm, and then repeat the process using this new sample according to the cluster that belongs to, i.e., that most closely resembles. In this way, the middleware can disclose the input through the group to which it belongs.

In brief, the second phase consists in three steps:

1. Intercept communications and gather the captures of the real communication;
2. Collect the cluster aggregation from the step 4 of the first phase;
3. Classify the real data into one of the clusters.

For demonstration purposes, from now on, we will refer to the information transmitted as the user's name, that is, what we are categorizing is the inputs of each millionaire, but we refer to that information by the name of each one.

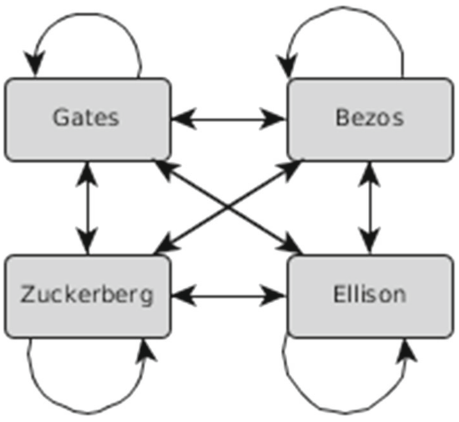

Fig. 2 Communications between four players

4.3 Setup

For both attacks (passive and active), we use the millionaire's problem described in Sect. 2 with the top four wealthiest tech billionaires as inputs. For each sample, it is used as a communication between two parties, i.e., between two out of four billionaires, as the inputs. Each communication is repeated five times with the same input to have more samples of the same communication, to be able to apply the clustering method.

In the first phase, we started by testing with all the inputs provided by the list of the billionaire's (96 inputs) using 10 runs. However, as we are going to use the *CompLearn* to demonstrate the results, we had to reduce the inputs and the repetitions, as the construction of the square distance matrix and the computation of the best-fitting unrooted binary tree is a computationally hard problem [30]. In order to solve this problem, we used only 4×5 (inputs $\times$ repetitions) to speed up and simplify the process.

In these tests, we are simulating a real communication situation between four billionaire's. The idea is to cluster the information of the communications according to the input's of the billionaire's. The attack is not focused on the identity of the people, but rather on the input they exchange, as one of MPC's properties is that the only information that can be inferred about the private data is whatever could be inferred from seeing the output of the function alone.

As we can see in Fig. 2, we used four billionaires: *Bill Gates*, *Jeff Bezos*, *Mark Zuckerberg*, and *Larry Ellison*, who have net worth of \$84.5 B, \$81.7 B, \$69.6 B and \$59.3 B, respectively. We used each of the inputs of the billionaires as both sender and receiver to simulate all communications. The generation of all the inputs allows producing a result similar to a rainbow table where all the possible combinations of inputs are computed. The idea is to gather a new communication sample, where the inputs are not previously known, and compare with the table of inputs.

4.4 Global premises

There are some assumptions that have to be considered in the setup of each framework in order to be able to carry out the attacks:

– All libraries have a limitation on the size of each input;
– The set of inputs that the tool accepts is finite and can be binary or an integer in the case of the millionaire's problem;
– DUPLO and TinyLEGO frameworks: The inputs have to vary more than 0.04 in entropy[1] value, considering the entropy value between 0 and 1.

4.5 Remote attacks (passive)

A passive attack involves someone listening to the communication exchanges. An example of this is an attacker sniffing network traffic using a protocol analyzer or some other form of introspection. The attacker finds a way to connect to the local network and captures the traffic for further analysis (including port mirroring).

For this, we start by using *tcpdump*[2] to analyze the traffic through the network (Listing 3 on "Appendix B") on each communication.

The traffic is captured and then used to evaluate the NCD between them. The use of compressors, implicitly, allows the attack to analyze the similarity between traces of communications traffic, in order to comprehend the kind of information being sent and to be able to distinguish the communications through their similarity.

The captured network traffic has a timestamp as an attribute that is a random property that can change the compression value. As we are measuring the distance between objects by applying NCD, it makes sense to remove timestamps because it is a component that is not controlled by us and varies between captures. This makes the compression index different between otherwise identical samples. For this reason, the attribute *-t* is placed in *tcpdump*. By taking the normal interface parameters, we have only the *-w* which writes the raw packets to a file rather than parsing and printing them out. We also use the attribute *-n* to filter only TCP packets because that is what MPC uses. Also, we do not consider IPs at all, we only gather information regarding the communication data.

[1] Entropy is a measure of unpredictability of information content [31].

[2] *tcpdump* is a tool that allows to inspect the traffic passing through the data network. Like all sniffers, *tcpdump* can be used for good (e.g., detecting communication errors), but also for evil (e.g., capturing personal data).

In this case, *tcpdump* is used to capture the traffic passing through the network by multiparty computation, so that it can be cluster by compression techniques.

Attack Specific Premises
In the passive attack with *tcpdump*, we have some assumptions:

- The attacker does not need access to the machine;
- The attacker needs access to the same network (running the multiparty computation protocol), as one of the communication parties.

4.6 Local attacks (active)

Active attacks adulterate the normal flow of information by changing its content and producing untrustworthy information, usually with intent to compromise the security of a system.

For this, we start by using *STrace*[3] to save the captures for analysis with compressors and data mining clustering.

The use of *STrace* is related to the high number of permissions required by *tcpdump* if the attacker does not have access to the network where one of the victims is, but can gain access to the users device. For this reason, we do not use *tcpdump* because it makes an active attack more difficult.

In brief, when we make an active attack in which we have to enter the victims machine, we have two possible situations:

- We make a passive attack by running the *tcpdump*, but we would have the same problem of obtaining root access to the machine, which in many situations is complicated;
- Otherwise, we use *STrace* and just by changing the executable (this step only requires user normal permissions), we are able to gather information through *STrace*.

We used *STrace* in different modes (Listing 4 in "Appendix B" with the example of the network mode). It is possible to collect information of specific system call group, i.e., the access to local files. We used three different options of *trace*, namely: *memory*, *network* and *file*. First, we start by testing an approach similar to *tcpdump*, but with *STrace*, to capture the network traffic with the *network* option. Then, to get better insight into the memory usage and system calls, we used the option *memory*. Finally, we used the option *file* to track file-related syscalls.

Attack Specific Premises
In the active attack with *STrace*, we have some assumptions:

- The attacker does not need network access (unlike the passive attack), but needs access to the machine;
- The attacker does not need to have any elevated privileges; only a standard user account on the machine, at most.

4.7 Evaluation metrics

This section describes the results obtained with the correspondent setup of *CompLearn* to produce the NCD matrix.

CompLearn
In order to analyze the dataset from the brute force attack, we used the *CompLearn* library to compute the NCD, constructing a distance matrix. The entries of this matrix will be the distances between each pair of traces, i.e., how similar they are. The first line of the Listing 1 represents the computation of the NCD matrix, the *-d* represents the directory where the user wants to perform the analysis with the NCD, and the next argument is the name of the directory. After this, the output is the NCD matrix produced by *CompLearn*. As this is a more compact form of gathering information, we are in a position to apply clustering algorithms. After this, we could pass the NCD matrix file to *maketree* (line 2) to create an unrooted binary tree. The result of *maketree* is a file with the name *treefile.dot* that describes how the nodes of the tree are related to each other and the correspondent label. After this, we can use the command line script *neato* available in the package *graphviz* (presented in the line 3) filter for drawing undirected graphs in a *.png*. In brief, *CompLearn* uses the exact quartet method to produce a ternary tree in which the leaves represent the files and the branches represent similarities between them. At each step, the algorithm tends to refine the tree by modifying the sub-trees. Each resultant tree is associated with a value of $S(t)$ which indicates how well the matrix is represented by the tree.

Listing 1 Executed commands to produce the NCD results

```
1 ncd -d 427000/ 427000 > NCDmatrix.clb
2 maketree NCDmatrix.clb neato
3 -Tpng treefile.dot > ncd-unrooted.png
```

5 Evaluation

The communication schemes presented in Fig. 2 contain sixteen communications repeated five times each, to perform the clustering. However, to perform the *maketree*, the number of communications is too high, as would make the tree too big to analyze (Fig. 6 in the "Appendix E"). In order to simplify this process, only one billionaire, Bill Gates, will be the sender of the communication.

[3] *STrace* allows the attacker to observe the system calls used by an application. *STrace* is useful because it can help the user to better understand what the system does during program execution, which can be a great help in tuning performance and resource management.

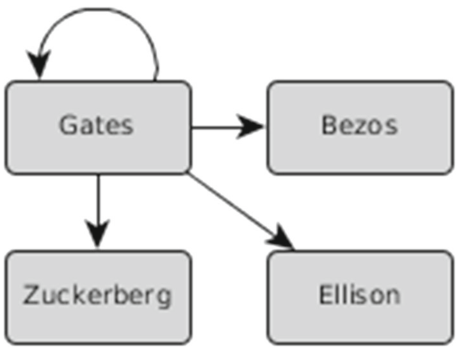

Fig. 3 Gates communications scheme

In order to do these tests, we used input sizes nine and twelve because of the limitations of the input size of ABY and SPDZ-2, respectively. We also tested with input size six because we found a reasonable salary value below which would be a very low value to be compared, i.e., with a lower value, the binary would be small and the number of zeros would be very high. In addition, the only test that would make sense in this case would be the SPDZ-2 so we chose not to do these tests.

5.1 Passive attack

Figure 3 represents the first communications established by Bill Gates. In this attack, we take advantage of the information captured by *tcpdump* in a way that the capture can be clustered and compared with new traces. This will create a real model where, based on this cluster, we can classify any new communications into one of the four groups of connections (by clusters).

To show the results of this attack, two binary trees will be used, the first with an example of a vulnerable library and the second for a secure library; the problems associated with the implementation of the vulnerable library will be tested with multiple input sizes.

Figure 4 shows an unrooted binary tree of the SPDZ-2 framework that clusters as leafs the *tcpdump* captures of the communications between two peers analyzed using NCD. Each name represented in the leaves has the following configuration: first the number of the iterations (from 0 to 4) followed by the name of the sender and then by the second peer of the communication. An example of such communication is one of the leaves under the red circle: (*3_gates_bezos*) that represents the fourth iteration of the communication between Gates and Bezos. However, the names of billionaire's represent their inputs, and not their identity.

The red circle in Fig. 4 represents the cluster of the communications between Gates and Bezos; these communications include five captures of the network traces. The representation of Fig. 4 also allows us to see that the remaining inputs of billionaires are split also in branches where we

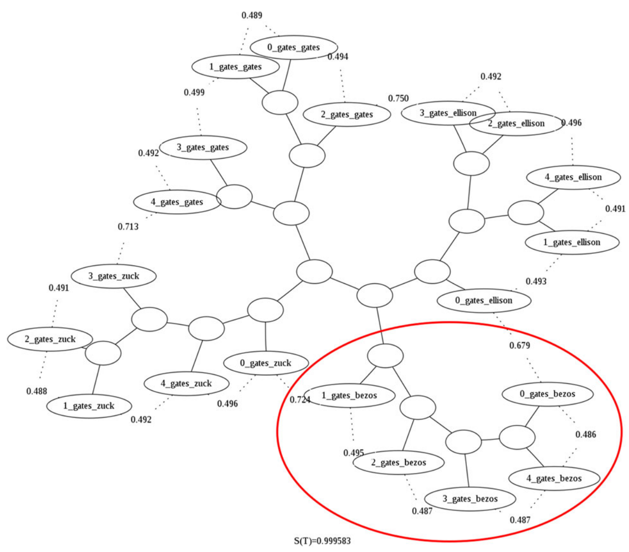

Fig. 4 Unrooted binary tree with the entire gates communications scheme in SPDZ-2

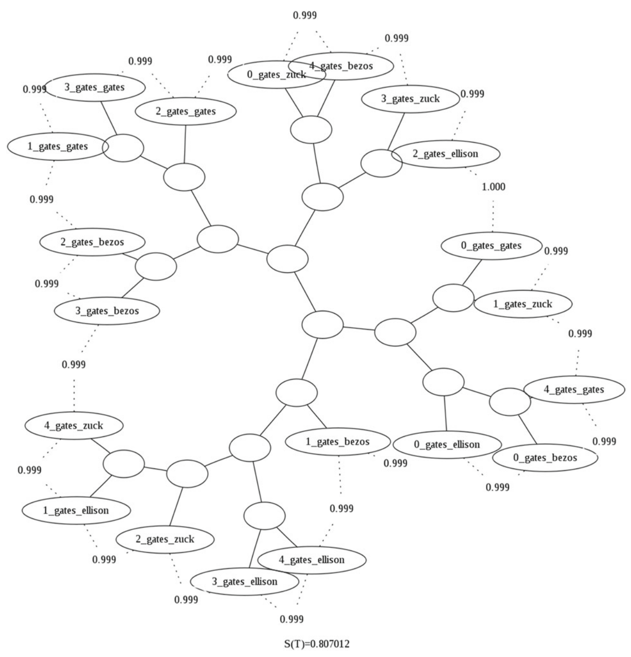

Fig. 5 Unrooted binary tree with the entire gates communications scheme in ABY

can split the communication input represented by Gates with the remaining inputs of the billionaires, separated by clusters. This is a small representation of Fig. 6 in "Appendix E" where we show all of the communications matched in a tree, and where all repetitions of the same communication are kept in a separate branch of the tree.

Figure 5 represents the cluster of secure communications between Gates and the other players in the ABY framework. The resultant tree is associated with the value of $S(t)$ which indicates how well the matrix is represented by the tree. However, in this case, the $S(t)$ is approximately 0.80, and this means that the fit between the matrix and the tree is not as good as the tree represented in Fig. 4, where the value of $S(t)$ is approximately 0.99. However, in this unrooted binary tree (Fig. 5), we can see that the data are not split by communications between two players, because there is not any part of the tree that has Gates communicating with the same player in the same cluster. This means that the ABY framework, using NCD, is not leaking information to the network in this situation, as opposed to the SPDZ-2 framework.

Figures 4 and 5 represent the results for an input size of 6.

Table 1 shows the results that we obtained through the NCD and data mining tools (*clara* algorithm) for all the MPC frameworks tests with different inputs for *tcpdump*.

The information in Table 1 considers the inputs given to the brute force middleware. In cases where the input is an integer, the program receives an input size of 6 (the net value of the billionaire, e.g., 845,000); otherwise, it receives a binary conversion of this input to 32 bits (because we use the default binary-circuit-based MPC and FHE implementations as already described in Sect. 3.1). This conversion to binary leads to situations where a library is vulnerable and others where it is not, as we can see in Table 1.

Table 1 represents three insecure libraries and one that is secure. The most insecure library is *SPDZ-2*, where we can see that for any input, the library is leaking information only with *tcpdump*. The *TinyLEGO* and *DUPLO* libraries have a leak of information when using a standard 32-bit comparison circuit. The secure library, or the most secure based on this test, is ABY, where we cannot cluster information based on the network captures.

In the search for the reason why both *DUPLO* and *TinyLEGO* libraries were vulnerable to an input of size 12 rather than size 6 led to the study of the inputs based on the entropy values between 0 and 1 (represented in Table 2). To obtain the values in the table, we used Approximate Entropy (ApEn)[4] by using *ap_entropy* from *PyEEG* Python Module (Listing 7 from "Appendix D").

The security of both libraries is ensured by the similarity between the output samples provided by the libraries to the network (related with the similarity of the entropy values), but when the difference between the entropy values increases, the cluster methodology can detect the difference in the information exchanged through the network. This vulnerability is interesting because security properties tend to increase in order to enhance robustness. However, in this situation, the opposite is happening: When we increase the values being exchanged, we increase the entropy and consequently the ability to exploit this vulnerability.

[4] Approximate Entropy is a technique used to quantify the amount of regularity and the unpredictability of fluctuations over time-series data [32].

Table 1 Comparison table

MPC framework	Input type	Input size 6	9	12
ABY	Int	○	○	X
TinyLEGO	Bin	○	○	●
DUPLO	Bin	○	○	●
SPDZ-2	Int	●	●	●

● = Vulnerable;
X = Limitation of the framework; not applied;
○ = Not vulnerable;

Table 2 Entropy table

Billionaires	Input size 6	9	12
Bill Gates	0.498	0.593	0.637
Jeff Bezos	0.489	0.602	0.569
Mark Zuckerberg	0.497	0.571	0.571
Larry Ellison	0.497	0.609	0.637

The entropy shows that the difference between the binary representations of these numbers varies according to the size of the integers which are represented in binary. For numbers less than 9, the binary has to pad 0s to the left of the number, which makes the entropy values very similar. However, for entropy values for the input of size 12, we have a range of values of [0.569 to 0.637], which makes the values more distinct. An example of the binary values of Gates with inputs size 6 and 12, respectively, can be seen in the Listing 6 in the "Appendix D."

5.2 Active attack

This attack has been done using the same communications rounds used in the previous passive attacks. In this test, we used *STrace* focused on the network traffic, but without the need of access to the network or escalated privileges. Table 3 shows the results of the vulnerability tests. ABY has a limitation, because we cannot use an input of the size 12. We can assess that SPDZ-2 is vulnerable to all input sizes, while ABY is not. DUPLO starts to leak information when using a small input size (6), but does not classify all the information within the correct cluster. However, it can partially classify the information, as it is capable of having good results in the 20 samples (5 times for each of 4 communications). With an input size of 9, we have 6 communications incorrectly classified, and with an input size of 12 we only have 3 communications in the wrong cluster. For this reason, we consider it a "partial vulnerability" because it has some incorrectly classified instances.

Table 3 Comparison table, of *STrace* command results

MPC framework	Input type	Input size 6	9	12
ABY	Int	○	○	X
TinyLEGO	Bin	○	○	●
DUPLO	Bin	○	◐	◐
SPDZ-2	Int	●	●	●

● = Vulnerable;
X = Limitation of the framework; not applied;
◐ = Partial vulnerable;
○ = Not vulnerable;

6 Low-level code analysis

We will now discuss the implementation issues that we uncovered during the assessment of *SPDZ-2*. We have described in the previous section that both passive and active attacks use compressors, they are capable of extracting more information than they should, due to the similarity between samples within the same millionaire's problem values.

In this section, we describe the code-level analysis used to understand the security properties of MPC that were missing from this implementation. For this, we start by searching for the similarity (NCD-based) between TCP traffic in the code. There were three steps that we used to assess this:

1. Network socket detection;
2. Unpack the network socket information;
3. Check the implementation of secret sharing.

In the network socket detection (*Step 1*), we started by searching the entry points of the network socket[5] and collected the information sent to the socket, in order to perform the brute force attack. We performed multiple tests with the same input on both clients, and the result is the same information sent to the other side. In *Step 2*, it is possible to see the unpacked version of this data in a readable example.

The information extracted from *Step 1* is packed before being sent to the network layer of the program. In order to understand the information sent, we analyzed the moment after the *unpack* and obtained some numbers. The code from the Listings 2 is responsible for receiving the values from the network socket where, in line 9, they are unpacked. When the program needs the input, it calls the function *GetValues*, also available in Listings 2. In this function, it is possible to print the entire information from the *vals* variable.

[5] A network socket is an endpoint to the communication flow between two programs running over a network.

Listing 2 Code from SPDZ-2

```
1  void MAC_Check<T>::AddToValues(vector<T>&
       values)
2  {
3    vals.insert(vals.end(), values.begin(),
         values.end());
4  }
5  void MAC_Check<T>::ReceiveValues(vector<T>&
       values, const Player& P, int sender)
6  {
7    P.receive_player(sender, os, true);
8    for (unsigned int i = 0; i < values.size();
         i++)
9      values[i].unpack(os);
10   AddToValues(values);
11 }
12 void MAC_Check<T>::GetValues(vector<T>&
       values)
13 {
14   for (typename vector<T>::iterator it =
         vals.begin() ; it != vals.end(); ++it)
15   {
16     cout << "\t \033[1;31m " << *it <<
           "\033[0m\n ";
17   }
18   ...
19 }
```

The variable *vals* prints the information that came from the network (Listing 5 in "Appendix B"), and these numbers are always equal to the results from the previous interactions. The numbers that we got before the *unpack*, we suspected that they were related with the secret sharing schemes according to the stages displayed. Also, there was a correspondence between the number of stages with the total number of elements in the vector *vals*. So, SPDZ-2 implementation in regards to the secret sharing and MAC produces always the same result for the same values of input. This is clearly an implementation issue.

This vector has another interesting feature. The SPDZ-2 has the ability to create and modify the programs, for example, the millionaire's problem. The programs are specified in a specific language *".mpc,"* created by the library. There is the possibility of getting private or public input; but for both, the option $reveal()$ can be used. The $reveal()$ function reveals the secret values and, for example, we can reveal the inputs for both parties.

To test this difference in the implementation, we used the $reveal()$ option to print the information and see the difference. We found the number in clear text ready to be received by the other client; so, in this situation, the $reveal()$ option removes the entire security of the library. We know that, in fact, if we use a $reveal()$ call in the program, that means that all parties have agreed to this and that the computed function also reveals this secret; however, the problem here is that SPDZ-2 sends the value in clear text to perform this $reveal()$ function. Despite the agreement of both parties, the value is not only shared between them, but are also readable to any attacker on the network.

As specified in [10], “the high-level idea of the online phase is to compute a function represented as a circuit, where privacy is obtained by additive secret sharing the inputs and outputs of each gate, and correctness is guaranteed by adding additive secret sharing of MACs on the inputs and outputs of each gate.” The two security properties are not assured in this implementation. These properties (privacy and correctness) are the most basic ones that a multiparty computing protocol must ensure.

7 Conclusions

In this article, we reviewed the security of relevant MPC implementations and assess their level of security. In our assessment, only one library showed good results ensuring security in our attack scenario.

Our results suggest that existing implementations of MPC protocols are vulnerable to a network traffic analysis, just by computing all the possible combinations of inputs. We distilled high-level findings derived from our initial attack in the SPDZ-2, where the implementation of the secret sharing always produces the same results in the source code. We offer a suggestion for debugging this type of system, as a future direction for library design and possibly to further research in the field.

For future work, we would like to extend the deployment of our tests in the remaining open-source libraries and other cryptographic protocols, and to also patch the current implementations in such a way that they could become secure against this type of attack. Also, there is the need for tools that could reliably test these security systems regarding the user interference in the process. Furthermore, NCD implementations need to be revamped to allow their deployment and use by non-experts.

Some of the libraries, not featured in this article, do not allow this test by using virtual machine communication. Some of the tools use *ssh*, which is detrimental to this line of research as they use mechanisms that rely/trust on centralized approaches, and worst, try to achieve security through obfuscation.

The insights and results gained throughout this work highlight the necessity of using open-source resources where researchers can study and deploy this type of approach, in an effort to build a secure open-source ecosystem).

Acknowledgements The work of João S. Resende was supported by a scholarship from the Fundação para a Ciência e Tecnologia (FCT), Portugal (scholarship Number PD/BD/128149/2016). The work of Patrícia R. Sousa and Luís Antunes was supported by Project “NanoSTIMA: Macro-to-Nano Human Sensing: Towards Integrated Multimodal Health Monitoring and Analytics/NORTE-01-0145-FEDER-000016,” financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF). The work of Rolando Martins was supported by a scholarship from the Fundação para a Ciência e Tecnologia (FCT), Portugal (scholarship Number SFRH/BPD/115408/2016). This work is financed by National Funds through the FCT—Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within the project CMU Portuga CMU/CS/0042/2017.

Appendix A: Millionaire’s problem code

The Code 7 represents the ABY code that allows the implementation of the millionaire’s problem. In this case, we read the input from a local file and, to perform the brute force, we just need to rewrite a file and rerun the program to have a different example.

Appendix B: Commands

The listings shows the commands used to produce the file of the output traffic of the protocol: network (Listing 3) and *STrace* (Listing 4). The machine that runs this process needs to have all the communications in the interface *lo* in idle, in order to have only the information correlated with the MPC process in the captures.

Listing 3 tcpdump command

```
tcpdump -eni lo -t -n tcp -w example.pcapng
```

Listing 4 STrace command

```
strace -f -e trace=memory -s 10000 python
    startMPC.py
```

Appendix C: Output of code analysis

We have to analyze the values printed from the secret sharing (Listing 5), in order to see whether this output is always equal in all the iterations with the same set of inputs.

Listing 5 Values variable

```
1797484165
-61705647039824681916694312663146678231
-36761208276779888592191149434901166533
15902840325947545570244179796300984094
...
33815892615659681644195482679118018626
1
```

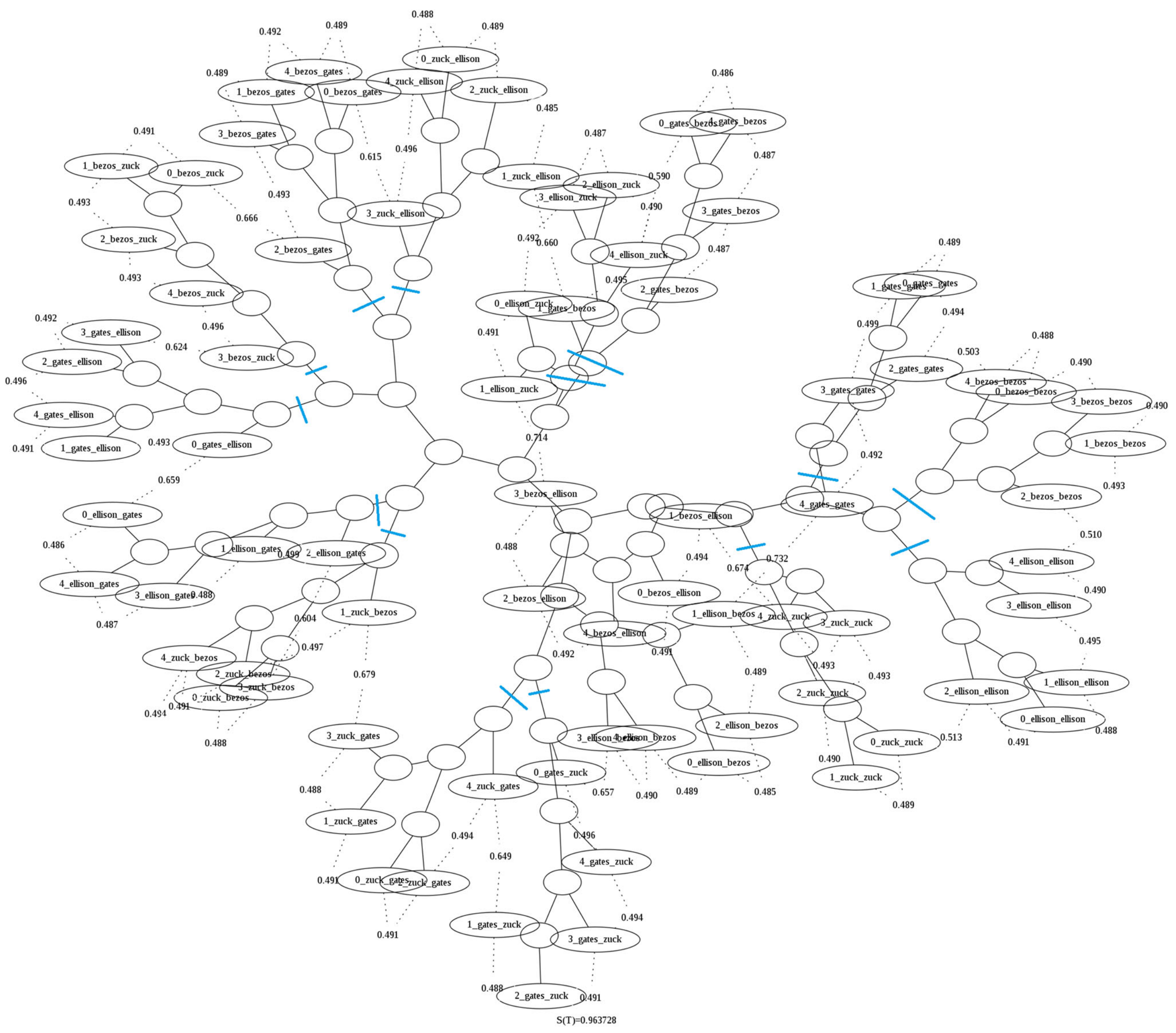
Fig. 6 Tree with the entire clustering of the brute force attack

Appendix D: Entropy value influence

The padding of zeros can influence the entropy values. The following example shows two different input sizes (salary of Bill Gates) converted to 32 bits binary. We can see that if we convert only 6 digits of salary (896 000), we have a lot of padding of zeros at the left (00000000000011001110010011001000). The same does not occur in the input size 12 (896000000000 - 11000100101111011110110011000010). It may be detrimental to have padding of zeros left, as entropy is influenced in the wrong way.

Listing 6 Binary values of Bill Gates with input size 6 and 12

```
Size 6: 00000000000011001110010011001000
Size 12: 11000100101111011110110011000010
```

We use the *python* command *pyeeg* to calculate the approximate entropy, used to present the entropy results in this paper.

Listing 7 Approximate Entropy Command

```
pyeeg.ap_entropy(a,2,0.1)
```

Appendix E: SPDZ-2 *maketree* with *complearn*

Figure 6 represents the entire tree generated by *complearn*. The tree contains a high $S(T)$ value, but the representation in some situation is complicated to visualize. In order to help the identification of different clusters, it has performed a set

Code 7: Input ABY

```
(role == SERVER) {
    std::ifstream file1("input1.txt");
    std::string str1;
    std::getline(file1,str1);
    std::istringstream reader1(str1);

    uint32_t bob_money;
    reader1 >> bob_money;
    s_alice_money =
        circ->PutDummyINGate(bitlen);
    s_bob_money = circ->PutINGate(bob_money,
        bitlen, SERVER);

    cout << "\nBob Money:\t" << bob_money;
  } else { //role == CLIENT
    std::ifstream file2("input2.txt");
    std::string str2;
    std::getline(file2,str2);
    std::istringstream reader2(str2);

    uint32_t alice_money;
    reader2 >> alice_money;
    s_alice_money =
        circ->PutINGate(alice_money, bitlen,
        CLIENT);
    s_bob_money = circ->PutDummyINGate(bitlen);
    cout << "\nAlice Money:\t" << alice_money;

  }
```

of "blue cuts" in the tree. This way, we can visualize a cluster flowing branch until the leaves.

In a highlight perspective, the tree can split each one of the traces connected to a similar example where we see all the 16 different types of communications perfectly split accordingly in the tree. The similar traces are formed of the same party's communicating with a number from 0 to 4 where an example can be 0_*gates*_*zuck* and 2_*gates*_*zuck*. Here, both are the same communication but in a different iteration, where 0 represents the first communication and 2 the third communication. This example has a communication between Gates and Zuck, where Gates is the initiator of the communication and Zuck the other party in the communication protocol.

References

1. Anderson, R.: Why cryptosystems fail. In: Proceedings of the 1st ACM Conference on Computer and Communications Security. ACM (1993)
2. Acar, Y., et al.: Comparing the usability of cryptographic APIs. In: Proceedings of the 38th IEEE Symposium on Security and Privacy (2017)
3. Georgiev, M., et al.: The most dangerous code in the world: validating SSL certificates in non-browser software. In: Proceedings of the 2012 ACM conference on Computer and communications security. ACM (2012)
4. Reaves, B., et al.: Mo (bile) money, Mo (bile) problems: analysis of branchless banking applications in the developing world. In: USENIX Security Symposium (2015)
5. Sousa, P.R., Antunes, L., Martins, R.: The present and future of privacy-preserving computation in fog computing. In: Rahmani, A., Liljeberg, P., Preden, J.-S., Jantsch, A. (eds.) Fog Computing in the Internet of Things, pp. 51–69. Springer, Berlin (2018)
6. Back, A., Moller, U., Stiglic, A.: Traffic analysis attacks and trade-offs in anonymity providing systems. In: Information Hiding, vol. 2137 (2001)
7. Cilibrasi, R., Paul, M.B.V.: Clustering by compression. IEEE Trans. Inf. Theory **51**(4), 1523–1545 (2005)
8. Wehner, S.: Analyzing worms and network traffic using compression. J. Comput. Secur. **15**(3), 303–320 (2007)
9. Santos, C.C., et al.: Clustering fetal heart rate tracings by compression. In: 19th IEEE International Symposium on Computer-Based Medical Systems. CBMS 2006. IEEE (2006)
10. Damgrd, I., et al.: Practical covertly secure MPC for dishonest majority or: breaking the SPDZ limits. In: European Symposium on Research in Computer Security. Springer, Berlin (2013)
11. Demmler, D., Schneider, T., Zohner, M.: ABY-a framework for efficient mixed-protocol secure two-party computation. In: NDSS (2015)
12. Kolesnikov, V., et al.: Efficient batched oblivious PRF with applications to private set intersection. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM (2016)
13. Frederiksen, T.K., et al.: TinyLEGO: an interactive garbling scheme for maliciously secure two-party computation. IACR Cryptology ePrint Archive 2015/309 (2015)
14. Kolesnikov, V., et al.: DUPLO: unifying cut-and-choose for garbled circuits. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2017
15. Multiparty Computation with SPDZ Online Phase and MASCOT Offline Phase. https://github.com/bristolcrypto/SPDZ-2. Accessed 11 Sept 2017
16. Orlandi, C.: Is multiparty computation any good in practice? In: 2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE (2011)
17. Cilibrasi, R., Cruz, A.L., de Rooij, S., Keijzer, M.: Complearn. http://www.complearn.org. Accessed 09 Jan 2017
18. Borbely, R.S.: On normalized compression distance and large malware. J. Comput. Virol. Hacking Tech. **12**(4), 235–242 (2016)
19. Li, M., et al.: The similarity metric. IEEE Trans. Inf. Theory **50**(12), 3250–3264 (2004)
20. Yao, A.C.: Protocols for secure computations. In: 23rd Annual Symposium on Foundations of Computer Science, SFCS'08. IEEE (1982)
21. Yao, A.C.-C.: How to generate and exchange secrets. In: 27th Annual Symposium on Foundations of Computer Science. IEEE (1986)
22. Yao, A.C. Theory and application of trapdoor functions. In: 23rd Annual Symposium on Foundations of Computer Science, SFCS'08. IEEE (1982)
23. Araki, T., et al.: High-throughput semi-honest secure three-party computation with an honest majority. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM (2016)
24. Theory and Practice of Multi-party Computation Workshops. http://www.multipartycomputation.com/mpc-software. Accessed 11 Sept 2017
25. The Richest People in Tech. https://www.forbes.com/richest-in-tech/list/. Accessed 11 Sept 2017
26. ABY: A Framework for Efficient Mixed-protocol Secure Two-party Computation. https://github.com/encryptogroup/ABY (2015)

27. A C++ Implementation of the TinyLEGO Cryptographic Protocol [NST17] for General Secure Two-party Computation. https://github.com/AarhusCrypto/TinyLEGO (2016)
28. Circuits of Basic Functions Suitable for MPC and FHE. https://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/. Accessed 11 Sept 2017
29. A C++ implementation of the DUPLO cryptographic protocol. https://github.com/AarhusCrypto/DUPLO. Accessed 11 Sept 2017
30. Souto, A.: Traffic analysis based on compression. In: Proc Conferȇncia sobre Redes de Computadores CRC'15, Évora, Portugal, Vol. 1, pp. 1–7, November 2015
31. Entropy (Information Theory). http://www.basicknowledge101.com/pdf/km/Entropy%20(information%20theory).pdf. Accessed 11 Sept 2017
32. Pincus, S.M., Gladstone, I.M., Ehrenkranz, R.A.: A regularity statistic for medical data analysis. J. Clin. Monit. Comput. **7**(4), 335–345 (1991)