TUC: Time-sensitive and Modular Analysis of Anonymous Communication

Michael Backes[1,2] Praveen Manoharan[1]
Esfandiar Mohammadi[1]

[1]Saarland University, Germany
[2]Max Planck Institute for Software Systems, Germany

backes@cs.uni-saarland.de
manoharan@cs.uni-saarland.de
mohammadi@cs.uni-saarland.de

October 18, 2013

Abstract

The anonymous communication (AC) protocol Tor constitutes the most widely deployed technology for providing anonymity for user communication over the Internet. Tor has been subject to several analyses which have shown strong anonymity guarantees for Tor. However, all previous analyses ignore time-sensitive leakage: timing patterns in web traffic allow for attacks such as website fingerprinting and traffic correlation, which completely break the anonymity provided by Tor. For conducting a thorough and comprehensive analysis of Tor that in particular includes all of these time-sensitive attacks, one of the main obstacles is the lack of a rigorous framework that allows for a time-sensitive analysis of complex AC protocols.

In this work, we present TUC (for Time-sensitive Universal Composability): the first universal composability framework that includes a comprehensive notion of time, which is suitable for and tailored to the demands of analyzing AC protocols. As a case study, we extend previous work and show that the onion routing (OR) protocol, which underlies Tor, can be securely abstracted in TUC, i.e., all time-sensitive attacks are reflected in the abstraction. We finally leverage our framework and this abstraction of the OR protocol to formulate a countermeasure against website fingerprinting attacks and to prove this countermeasure secure.

Contents

1 Introduction

Anonymous communication protocols, as provided by the Tor network [45], are an increasingly popular way for users to protect their privacy by hiding the user's location. The Tor network is currently used by hundreds of thousands of users around the world [44]. Sudden increases in Tor's usage have furthermore been shown to correlate to privacy-invasive political events which demonstrates the global political importance of the Tor network [13].

In order to precisely understand the anonymity guarantees provided by Tor, several rigorous analyses have been conducted [43, 8, 18, 2]. These analyses show strong anonymity guarantees for the onion routing protocol used by Tor; however, all of these analyses use an asynchronous communication model and abstract from all attacks that involve measuring timing patters, which arguably form the most important class of attacks against Tor's anonymity guarantees [38, 20, 11, 47, 39, 37, 27, 16, 29, 35]. One of the main obstacles in including such time-sensitive attacks into a rigorous analysis is the lack of a theoretical framework for the modular analysis of complex protocols against time-sensitive adversaries.

In this paper, we follow the successful line of research on simulation-based, composable security started with Goldreich et al. [22] and put forward by Canetti [9], which enable the modular analysis of complex cryptographic protocols by using the notion of secure realization: a complex cryptographic protocol is proven to be as secure as a simpler protocol (called ideal functionality), which is easier to analyze and sometimes even trivially secure. All such previous frameworks however are asynchronous and thus do not allow for analyses which consider time-sensitive adversaries.

Contribution. We introduce a simulation-based composability framework (with sequential activation) that includes a comprehensive notion of time that is suitable for and tailored to the demands of analyzing anonymous communication protocols. In particular, we extend a modified version of GNUC [25], which addresses many of the problems faced by earlier designs, by introducing a local clock to every party in the GNUC's sequential activation network model.

We discuss how the basic network model has to be altered in order to adequately account for time and show solutions for problems that occur when handling time-sensitive interaction between different parties over the network. We then show that, even after adding a fine-grained notion of time, classic properties of composable security such as universal composability and the joint-state theorem for secure realization also hold in our time-sensitive network model.

Finally, we exemplify the usefulness of our model by showing that a previously introduced abstraction for the onion routing protocol [2] is also securely realized in our time-sensitive model, albeit with small modifications in order to account for timing-related leakage. This paves the way for future analyses of the onion routing protocol which also account for timing attacks. As a result of our analysis, we additionally propose a small change to the onion routing protocol which counters the website fingerprinting attacks [7, 40, 24] known for Tor traffic using the HTTP [19] protocol for web surfing.

Outline. Section 2 discusses related work. Section 3 presents our extension of the sequential network model presented in the GNUC framework [25] with time. We present the general layout of the network model and discuss alterations made to account for timing. Section 4 then introduces the notion of secure realization into this time sensitive network model and shows that classic results of composable security are preserved in the time sensitive setting.

In Sections 5 and 6 we then exemplify the use of our framework by giving as secure abstraction of the onion routing protocol in the timed setting based on the abstraction provided in [2] and provide an improvement for the onion routing protocol that counters website fingerprinting attacks.

2 Related Work

This work contributes to the successful line of work on simulation-based universal composability frameworks [9, 41, 10, 5, 33, 25]. Composability frameworks allow for a modular analysis of large and complex multi-party protocols, where the security of the whole protocol is derived from the security analysis of the sub-protocols of which it is composed. The GNUC framework, presented by Hofheinz and Shoup [25], elegantly solves some of the problems that haunt earlier designs, such as correct definitions for protocol composition and polynomial time run time bounds in a system of interacting Turing machines. All previous work, however, only considered asynchronous systems, in particular time-sensitive attackers were ignored, i.e., attackers that can perform time-measurements. Thus, all previous frameworks were not

suited for analyzing low-latency anonymous communication protocols, such as the onion routing protocol, which underlies Tor [45]. Our framework TUC is based on the asynchronous GNUC framework but introduces a comprehensive notion of time that is suitable for analyzing anonymous communication protocols yet preserving all valuable properties such as universal composability or the Joint State Theorem.

There has been work [12, 28, 6, 34, 31, 32] on the time sensitive analysis of cryptographic protocols using timed automata or networks of timed automata. These analyses however are mainly concerned with protocols which make use of timestamps and the verification of safety properties of such protocols, and less with time sensitive adversaries which use timing information to break security. Furthermore, only simplified adversary models are considered, such as Dolev-Yao style adversaries. In particular, the timed automata model is not expressive enough to model arbitrary, turing complete adversaries, which is of large interest in the analysis of anonymous communication protocols. We therefore stick with a network model which consists of interacting turing machines.

Tor [45] is one of the most used anonymous communication protocols to date [44] and implements the onion routing protocol introduced by Goldschlag et al. [23]. There has been significant work in analyzing the anonymity guarantees provided by Tor using different approaches [17, 18, 8, 43, 3]. In [2] an UC-secure abstraction of Tor is presented and in [3] the AnoA framework is presented, which uses this abstraction to quantify the anonymity guarantees provided by Tor. The major shortcoming of all these pieces of work is that they do not consider timing features of network traffic, which in particular lead to timing based traffic analysis. Considering the amount of proposed attacks [15, 38, 20, 11, 7, 40, 24] and countermeasures [48, 49, 36, 1] in the literature that use these timing features, it is clear that a formal model for the analysis of anonymous communication protocols against time-sensitive adversaries is required. TUC provides such a model, and we present how TUC can be used to prove a countermeasure secure against a variant of the prominent class of website fingerprinting attacks [7, 40, 24].

3 Time-sensitive Network Model

In this section we present TUC, the first simulation-based composability framework that considers a time-sensitive adversary. TUC builds upon previous asynchronous simulation-based frameworks, such as GNUC [25] and the framework by Unruh [46], but fundamentally extends these frameworks by incorporating a notion of time while preserving the highly desired properties such as universal composability and the joint state theorem.

A general overview. We introduce time by capturing via a timer the current global time for every machine in the network. Whenever a machine is activated, its timer is updated based on the number of steps done by the machine and the speed of it. The speed of a machine is either predetermined if it already existed at initialization, or is determined by the protocol that it executes if the machine is created during runtime. Furthermore, we require that the actual local time experienced by each machine is given by a strictly monotonically increasing function of its current global time, thereby modeling unsynchronized clocks.

We stick to the classic sequential activation model; however, by introducing a notion of time we inherently also allow parallel computation. Therefore, it can happen that one machine is already far in the future while all other machines are still in the past. In order to achieve consistency, i.e. to achieve that no party receives messages from the future, we introduce a distinguished machine, called the **execution**. This execution basically manages the timer of each machine and the timely delivery of messages between machines.

The execution attaches to each message that is sent through the network a **time-stamp**, which is only visible to the execution. This time-stamp, loosely speaking, denotes the local time of the sending party when the message was sent.

The environment and the attacker might consist of several machines that work in parallel. A natural way of modeling this capability is to represent these environment and the attacker as a set of parallel machine. While such a model is more accurate, we decided for the sake simplicity to over-approximate this strength of the environment and the attacker by allowing both parties to make an arbitrary (but poly-bounded) amount of computation steps in one time-step.

As in GNUC, a protocol is formalized as a runtime library that assigns to each machine the program code to be executed by the respective machine and the speed of the machine that executes the code. We stress that a network has only one such runtime library, i.e. one protocol.

Initialization: All input tapes are set to empty, all timer-variables are set to the initial value and no links are compromised.
Machine Activation: Every time a party $M_i \in \mathcal{M}$ gives control to the network execution, to current global time T_i for M_i is updated according to

$$T_i := T_i + \frac{n}{c_i},$$

where n is the number of steps done by M_i in its last activation and c_i is its throughput.

upon input (time) **from** $M_i \in \mathcal{M}$
1: retrieve T_i
2: compute local time $t_i := f_i(T_i)$.
3: activate M_i with input t_i on the time tape.

upon input (increase) **from** $M \in \{\mathbf{Env}, \mathcal{A}\}$
1: set $T_M := T_M + 1$
2: activate M

Figure 1: Timing and Initialization in EXEC with machine set $\mathcal{M}$

3.1 Execution

The network is run inside a machine, called the **execution** (EXEC). The execution runs all parties in the network as sub-machines, delivers messages between these sub-machines, and maintains a timer for every sub-machine.

The network execution starts when the execution EXEC is activated with the security parameter k and an input x, which is forwarded to the environment. We define the output of the network execution as the output of the environment ENV after observing the communication between the involved parties during the network execution.

We capture this output by introducing the random variable $\mathsf{EXEC}_k(\Pi, \mathcal{A}, \text{ENV}, x)$, where Π is the set of protocols used in the network, ENV denotes the environment and $\mathcal{A}$ the network adversary. The value of $\mathsf{EXEC}_k(\Pi, \mathcal{A}, \text{ENV}, x)$ is the result of the random experiment of running EXEC with all of the aforementioned components.

We first describe the single aspects of the execution EXEC in the subsequent subsections, and at the end of this section we present the full description of EXEC in Figure 9.

3.1.1 Timing

In order to introduce the notion of time into our model, we assign to every machine in the system a **local time**. The local time of a machine is a function of the **global time** managed by the execution. We call this the machine's **local-time function**.

For every machine M_i, the execution maintains a **machine timer** T_i. This timer records the current global time of M_i and is updated every time M_i returns control to EXEC. T_i is initialized to 0 at the beginning of the execution.

In order to capture that each machine provides different performance, each machine M_i is characterized by a throughput constant c_i, also called M_i's **speed**, which specifies how many computation steps M_i does per time unit. Hence, the timer T_i for M_i is updated by

$$T_i := T_i + \frac{n}{c_i}$$

where n is the number of steps M_i did in its last activation. The only exception to this rule are the machines representing the adversary and the environment: These machines are **timeless**, i.e. they decide when their timers are set forward, and tell EXEC when to do it (see Fig. 1).

Another important aspect here is how you assign a throughput coefficient to machines in the network. While the coefficients can be predetermined for machines which already exist at initialization, they somehow have to be determined for the machines which are dynamically created during runtime.

We propose following simple solution: The protocol Π used in the network not only determines the code executed for each basename, but also provides a distribution over throughput coefficients. The execution can now draw the throughput coefficient from these distributions whenever a new machines is created during runtime. Note that this offers only one of many possible solutions. Other variants might be used for specific application scenarios.

Whenever a message m is sent from machine M_i to machine M_j through the network, EXEC records a **time stamp** T_m for this message. This time stamp is set to the current time T_i of the sender M_i

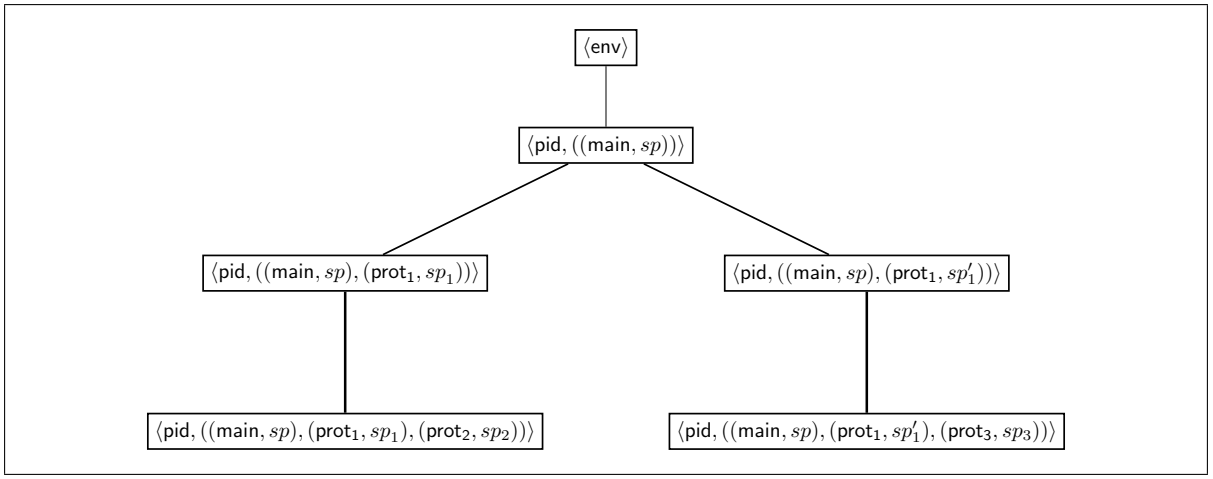

Figure 2: The Tree Structure of Parties in TUC

after updating it. When delivering m, EXEC puts the tuple (m, T_m, p) into the input queue Q_j of M_j, from which messages are retrieved whenever M_j listens for new incoming messages. Here p denotes the port through which M_j receives the message m. We use ports to differentiate between network and inner party communication and elaborate further on our communication model in Section 3.1.4. A tuple (m, T_m, p) retrieved from Q_j is only forwarded to M_j if $T_m \leq T_j$, i.e. M_i has progressed far enough in time in order to receive the message. The methods involved in message passing are presented in Figure 4 in Section 3.1.4.

Each machine M_i can requests its local time by sending a (time) request to the execution (see Fig. 1). EXEC then computes the local time of M_i by applying M_i's local time function f_i to its current global time T_i. Formally, the local-time function is a strictly monotonically increasing function from rational numbers to rational numbers, $f : \mathbb{Q} \to \mathbb{Q}$, which is efficiently computable and invertible. This is necessary as EXEC needs to invert the local time function in order to process delayed message sending, which is an option for protocol machines in the network and will be required in our constructions in Section 6.

3.1.2 Protocol Machines & Session Identifiers

In order to adequately represent complex protocols in our model, we adopt the notion of protocol machines from GNUC [25].[1] Here, each party P participating in network communication is represented by a tree of machines, which each provide the sub-protocols used by P. This tree structure allows for a clean definition of composition which we present in Section 3.2.1. This is structure is in line with what is presented as a structured system of interactive machines presented in [25, Section 3].

Each machine M in the network is identified by a unique **machine ID** id_M. The machine ID is a tuple $\mathsf{id} = \langle \mathsf{pid}, \mathsf{sid} \rangle$, where pid is the **party identifier** and sid the **session identifier**.

Machines can have the same party ID. They then belong to the same party, defining the set of machines this party works with. The form of this set is defined by the session IDs. These session IDs are structured as paths $(\alpha_1, \ldots, \alpha_k)$. We call a machine M **parent** of another machine M', if $\mathsf{pid}_M = \mathsf{pid}_{M'}$ and $sid_{M'}$ is a one-step extension of sid_M, i.e. $\mathsf{sid}_M = (\alpha_1, \ldots, \alpha_{k-1})$ and $\mathsf{sid}_{M'} = (\alpha_1, \ldots, \alpha_k)$. We then also call M' a **child** of M.

Machines in different parties can still have the same session IDs. We call machines with different pid, but the same sid **peers**.

The last component α_k of a session ID is the **basename** of the respective machine. The basename will be of the form $\alpha_k = (\mathsf{protNAME}, sp)$, where protNAME specifies the name of the protocol executed by the machine, and sp contains specific **session parameters**. The protocol name is used to determine the code executed by the respective machine, as detailed in Section 3.2.

We also adopt all of the constraints listed in [25, Section 4,5]. These make sure that in the end, our network is well-formed and that each party really consists of a tree of machines.

[1] Currently, our framework uses Turing machines as a machine model, but for analyzing timing leakage of algorithms other machine models might be better suited, such as Random Access Machines, or a machine model that even incorporates cache. We leave such extension for future work.

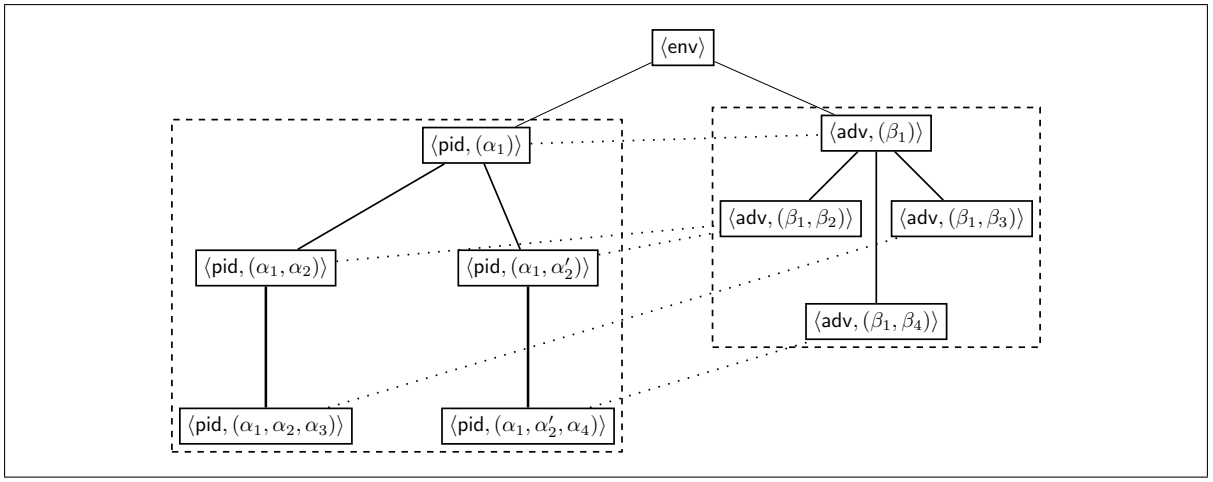

Figure 3: Network Adversary and Communication in TUC

Inside a party, a node in the machine tree can create new machines as children nodes by simply sending a message to the yet non-existent machine. The execution then checks whether the message sent induces a valid extension – where validity is defined by the protocol used in the network, see Section 3.2– of the machine tree and creates the new machine. Figure 2 illustrates the machine trees used in our model.

The tree structure for machines presented here is the same as in GNUC; we therefore refer to the GNUC paper [25, Section 4] for a more extensive presentation.

3.1.3 Environment and Adversary

Influences to network communication outside of the regular parties are traditionally captured in two special parties called **environment** and **adversary**: the environment represents user behavior, operating systems or other entities that control the actions of the network parties, while the adversary represents adversarial behavior in the network.

We identify the network adversary and the environment with special machine IDs: all machines that belong to the party representing the adversary $\mathcal{A}$ have the party ID $\mathsf{pid} = \mathsf{adv}$, whereas the environment Env consists of only one machine with machine ID $\mathsf{id}_{\text{Env}} = \langle \mathsf{env} \rangle$.

The adversary $\mathcal{A}$ also serves as **scheduler** that is activated every time it is not clear what happens next in the network. Similarly to the other parties, $\mathcal{A}$ consists of a machine-tree, where $\mathcal{A}$ has a sub-adversary for each basename in the network. Each of these sub-adversaries has its own clk port, which is triggered by the execution EXEC whenever a machine with the corresponding basename causes a scheduling exception. The tree-structure of the adversary is depicted in Figure 3.

In the real world, the environment and the adversary might consist of several machines that work in parallel. A natural way of modeling this strength is to represent the environment and the adversary as a set of parallel machines. While such a model is more accurate, for the sake of simplifying proofs, we abstract this strength of the environment and the adversary by allowing both parties to make an arbitrary (but poly-bounded) amount of computation steps in one time-step, i.e., by making these parties **timeless**. Technically, a party P is timeless if its timer is not increased by the network execution but by the party P itself. This design decision will also be crucial in the proofs for the composability results we present in Section 4.1.

3.1.4 Communication Model

We differentiate between inner party communication and network communication: inner party communication captures all communication between children and parent nodes inside a machine tree, but also communication between the environment and root machines, while network communication comprises all communication between parties, which we would typically expect to happen over open networks. Figure 3 illustrates this with thick lines for inner party communication and dashed lines for network communication.

Formally, we model these communication options using **ports**, which roughly correspond to the real world equivalent of ports which enable a single machine to communicate over different channels. We make

upon (m, id) **on port** *net* **from** $M_i \in \mathcal{M}$ **at time** T_i

1: **if** M_{id} is a peer of M_i **then**
2: **if** $(i, \mathsf{id}) \in \mathcal{C}$ **then**
3: put (m, T_i, net) into $Q_{\mathcal{A}_i}$ of the corresponding sub-adversary $\mathcal{A}_i$.
4: activate $\mathcal{A}_i$.
5: **else**
6: put (m, T_i, net) into Q_{id}
7: activate M_{id}'s scheduler $\mathcal{A}_i$ with scheduling request
8: **else**
9: return error to M_i

upon (delay, m, t) **on port** q **from** $M_i \in \mathcal{M} \setminus \{\mathbf{Env}, \mathcal{A}\}$

1: **if** $t \geq f_i(T_i)$ **then**
2: compute global time $T = f_i^{-1}(t)$
3: execute appropriate send message for m with time stamp T for port q
4: **else**
5: return error to M_i

upon m **on port** $p \neq net$ **from** $M_i \in \mathcal{M}$ **at time** T_i

1: **if** there is a machine with input port p **then**
2: let id be the machine ID of the unique machine with input port p
3: put (m, T_i, p) into Q_{id}
4: activate M_{id}'s scheduler $\mathcal{A}_i$
5: **else**
6: **if** $p = \mathsf{pid}(M_i).\mathsf{sid}(M_i).\mathsf{sid}' \wedge$ sid' proper extension of $\mathsf{sid}(M_i)$ **then**
7: let $(\mathsf{protNAME}, sp) = \mathsf{basename}(\mathsf{sid}')$
8: let $(cd, \mathcal{S}) = \Pi(\mathsf{protNAME})$
9: sample speed c' from $\mathcal{S}$
10: create a new machine M with code cd,
11: $\mathsf{sid}(M) := \mathsf{sid}'$
12: $c(M) := c'$
13: set up translation of M's environment port to p
14: put (m, T_i, p) into $Q_{\langle \mathsf{pid}, \mathsf{sid}' \rangle}$
15: activate M's scheduler $\mathcal{A}_i$
16: **else**
17: return error to M_i

Figure 4: Communication methods in EXEC with machine set $\mathcal{M}$ and protocol Π

use of four different kinds of ports: network ports, environment ports, subroutine ports, and scheduling ports. Technically, each port consists of an input-output port pair. We give each port pair a single name p and subscripts *in* for the input port p_{in} and *out* for output port p_{out}.

For communication over the network, M sends its messages over its **network port**, addressing the recipient using the recipient's machine id. All incoming messages are received through M's **network port**. A machine M can only send messages over the network to another machine M' if either M' is a peer of M or M' is the network adversary.

Each machine M has an environment port $\mathsf{sid}(M).\textsc{Env}$ which it uses to communicate with its parent node: If M is a root machine of a party, this is used to communicate directly to the environment. On the other hand, if M is a child of another node in the same party, M uses this port to communicate with its parent.

For all connections towards children inside the same party, a machine M has a set $\mathcal{S}$ of subroutine ports: every message sent by M through a subroutine port $p \in S$ is sent to the unique child node M' with a corresponding port of the same name. In case M wants to create a new machine M' as a child, M creates a new port p' in S and addresses M' through this port. EXEC then recognizes that p' is not in use yet and creates a new machine M', as detailed in Figure 4.

Inner party ports follow the naming convention $\mathsf{pid}.\mathsf{sid}_1.\mathsf{sid}_2$. Here pid is the process ID of the party, sid_1 is the session ID of the parent node M_p, and sid_2 the session ID of the child node M_c. Note that M_c communicates to its parent via its environment ports. The execution therefore makes an implicit port translation between environment ports of children nodes and inner party communication ports as defined above. Through this, we realize a variant of what is introduced as **Caller ID Translation** in [25, Section 4].

Additionally, all machines with party id adv have a special input port *clk* called the **scheduling port**. These machines are activated whenever a machine in the network releases control without sending a message to another machine. Which scheduler is activated at a given point is determined by the sid of the scheduler and the sid of the machine that was activated last. The network execution is responsible for correctly activating the correct scheduler. The methods used for message passing inside EXEC are presented in Figure 4.

3.1.5 Scheduling

Other simulation-based traditionally make use of a network model that uses a sequential activation model: machines in the network directly activate each other by sending messages. Keeping to this traditional sequential activation model, however, causes several problems as soon as you introduce time: messages

upon input (listen) **from** $M_i \in \mathcal{M}$

1: **if** Q_i is not empty **then**
2: Pull next message (m, T_m, p) from Q_i
3: **if** $T_m \leq T_i \wedge \forall M_j \in \mathcal{M} \setminus \{M_i\} : T_j \geq T_m$ **then**
4: activate M_i with m on port p
5: **else**
6: Put (m, T_m, p) into Q_i
7: activate M_i's scheduler $\mathcal{A}_i$ with scheduling request

upon activation by $M_i \in \mathcal{M}$ **without output**

1: activate M_i's scheduling machine $\mathcal{A}_i$ with scheduling request

upon input (activate, M_j) **from** $\mathcal{A}_i$

1: **if** M_j in listen state **then**
2: **if** Q_j is not empty **then**
3: Pull next message (m, T_m, p) from Q_j
4: **if** $T_m \leq T_j \wedge \forall M_i \in \mathcal{M} \setminus \{M_j\} : T_i \geq T_m$ **then**
5: activate M_j with input m on port p.
6: **else**
7: activate $\mathcal{A}_i$ with a scheduling request
8: activate M_j without input

Figure 5: Scheduling methods in EXEC with machine set $\mathcal{M}$

from the past arrive at nodes which are already in the future or the environment can push certain nodes arbitrarily far into the future.

Example 1: Inconsistencies with regular sequential scheduling.
Consider machines M and M' which go into a timeout state if they do not receive a message upto some point in time T^*

1: ENV repeatedly actives machine M through $\mathcal{A}$, which causes M to activate for one step and then return to the listening state. This effectively pushes M to time $T > T^*$ the future.
2: M goes into the timeout state, as it did not receive any message until time T^*
3: ENV tells machine M' to send a message to M at time T_0. Including processing the command, the message is sent at time T'
4: M receives a message from time $T' < T^*$ at time T^*.

M now erroneously went into the timeout state, even though M' sent a message to M before the timeout should have occurred. ◇

To avoid such inconsistencies, we deviate slightly from the traditional sequential activation model to what we call **consistency enforcing scheduling**: Whenever a machine M_i listens for a new message, M_i will only be activated again as soon as all other machines M_j have at least progressed as far in time as the earliest message in M_i message queue Q_i. If Q_i is empty, M_i will remain inactive until $T_j \geq T_i$ for all other machines M_j.

Consistency enforcing scheduling resolves inconsistencies regarding timing that might otherwise occur in decisions made by machines in the network: for example, a machine deciding to cause a time-out after not receiving messages upto some point in time T can be sure that it will not receive any messages "from the past" after doing so, contrary to above example. Methods in EXEC involved with scheduling are shown in Figure 5.

3.1.6 Shared Memory

As in other simulation-based framework, our goal is to analyze complex protocols by simplifying them to ideal functionalities which have additional capabilities. We captures these capabilities in form of **shared memory** between all ideal peers in the network. Access to shared memory is granted via a special port, through which parties can request read/write actions on the memory.

With regard to timing, the shared memory is special: We want to allow for data in the shared memory to be accessed at all times, while each request is answered with the current version of the data for the time of the request. We therefore call the shared memory **omni-time**: If a machine accesses the memory, it gets the version from the current time of the machine. If some other machine in the future already modified the memory, this modification will not be visible to the requesting machine until it lives in the same time as well.

Figure 6 gives a possible pseudo-code implementation of a shared-memory unit in the time-sensitive network model.

Consistency enforcing scheduling for shared memory. Similar to the scheduling of machine activations (see Section 3.1.5), the concept of an omni-time shared memory causes consistency issues:

Upon activation:

1: **if** there is an input $(\mathsf{lookup}, D, M_i, T_i)$ **then**
2: put the request $(\mathsf{lookup}, D, M_i, T_i)$ into the lookup queue Q_L
3: remove request $(\mathsf{lookup}, D', M_j, T_j)$ with smallest time stamp T_j from Q_L
4: call $\mathsf{lookup}(D', M_j, T_j)$

Upon input $(\mathsf{write}, D, d, M_i, T_i)$**:**

1: write value d into data set D with time stamp T_i
2: return confirmation to M_i

lookup(D, M_i, T_j)

1: request current time of all machines $M_j \in \mathcal{M}_{\mathsf{MEM}}$ with access to MEM.
2: **if** $\forall M_j \in \mathcal{M}_{\mathsf{MEM}} : T_j \geq T_i$ **then**
3: return data set D to requesting party M_i at time T_i
4: **else**
5: queue up the request $(\mathsf{lookup}, D, M_i, T_i)$ in the lookup queue Q_L

Figure 6: The shared memory MEM

For example, what happens if a party that lives in the past changes a data set a party living in the future already read (in the future)?

Example 2: Consistency issues in omni-time memory.
Consider two machines M (living at time T_M) and M' (living at time $T_{M'} << T_M$) which both access the shared memory MEM to read/modify a data-set d.

1: Env tells M to read d from MEM
2: M reads d from MEM at time $T_M + \delta$
3: Env tells M' to modify d in MEM
4: M' modifies d in MEM at time $T_{M'} + \delta' < T_M + \delta$

M now has read the value of d which is actually no longer correct, as M' modified d in the past after M read it. ◇

We solve these consistency issues by using a variant of consistency enforcing scheduling, which restricts the scheduling of memory access requests as follows: Lookup requests by a machine M are only processed by the shared memory MEM if all machines with access to MEM, denoted by the set $\mathcal{M}_{\mathsf{MEM}}$, are at least at the same time as M (or further).

If this condition is not true, the shared memory puts the request together with its time stamp into a time-ordered queue Q_L, which sorts all unanswered requests by their time stamps. Upon every activation, MEM checks Q_L for unanswered lookup requests and retrieves the one with the smallest time stamp. MEM then checks the lookup request for validity (based on its time stamp), and processes it if it is.

In case MEM cannot process any lookup requests, it finishes the execution without sending a confirmation message, causing a scheduling request to the scheduler.

As a consequence we get Corollary 1 which ensures consistency of shared memory entries read by machines in the network.

Corollary 1. *If a data set D is read by a party M from a shared memory* MEM *at time T, then any changes to D will only happen at a point in time $T' \geq T$.*

3.1.7 Compromisation

We assume that inner-party communication, i.e. communication between children and parent nodes, cannot be intercepted.

Previous composability frameworks assume a global adversary which intercepts all messages sent between parties over the network. This is a necessity for realization proofs between protocols which do not inherently leak information to the adversary. However, in the special case of anonymous communication (AC) protocols, a global adversary poses a problem, as e.g. Tor is not secure against global adversaries [47, 39, 37, 11], and is not even designed against global adversaries.

Partial compromisation of the network can be modeled by introducing special network functionalities, which are used as a link between parties. This approach is exemplified in [2].

To simplify the analysis however, especially with regards to AC protocols, we assume an, initially uncompromised network. The environment Env however can compromise network communication links between two machines by sending a compromise message to the execution EXEC indicating which link

upon $(\mathsf{compromise}, \mathsf{id}_1)$ **from parent of** M_{id_1}

1: **if** $M_{\mathsf{id}_1} \notin \mathcal{C}_M$ **then**
2: replace M_{id_1} code to cd_{comp}
3: $\mathcal{C}_M := \mathcal{C}_M \cup \{M_{\mathsf{id}_1}\}$
4: send $(\mathsf{compromise})$ to M_{id_1}
5: **else**
6: return error to Env

upon input $(\mathsf{compromise}, \mathsf{id}_1, \mathsf{id}_2)$ **from Env**

1: $\mathcal{C}_L := \mathcal{C}_L \cup \{(\mathsf{id}_1, \mathsf{id}_2)\}$
2: activate Env with $(\mathsf{compromised}, \mathsf{id}_1, \mathsf{id}_2)$ as input.

Figure 7: Corruption Commands in EXEC for Adaptive Compromisation

upon $(\mathsf{compromise}, \mathsf{id}_1)$ **from parent of** M_{id_1}

1: **if** $M_{\mathsf{id}_1} \notin \mathcal{C}_M \wedge M$ did not receive a message yet **then**
2: replace M_{id_1}'s code to cd_{comp}
3: $\mathcal{C}_M := \mathcal{C}_M \cup \{M_{\mathsf{id}_1}\}$
4: send $(\mathsf{compromise})$ to M_{id_1}
5: **else**
6: return error to Env

Figure 8: Machine Corruption in EXEC for Static Compromisation

should be compromised. Afterwards, any communication on the compromised link is forwarded to the adversary, who then decides on how to proceed with the message (see Fig 7).

The machines themselves can be compromised by the environment Env by sending a special compromise message to the respective machine. Upon receiving this compromisation message, EXEC replaces the code executed by the receiving machine M to the code of a compromised machine cd_{comp}: Whenever M receives a message, it is forwarded to the adversary, who can then tell M how to proceed. EXEC then forwards the corruption message to M, which in turn responds with an answer to Env containing the current state of M and from then on is under full control of the adversary (see Fig. 7).

The analysis of AC protocols usually differentiates between two important classes of Compromisation: On the one hand static compromisation, where the adversary can only compromise at the beginning of the execution, and on the other hand adaptive compromisation, where the adversary can also compromise during the execution. While the presentation of EXEC in Figure 9 works for the adaptive case, we need to make some changes for the static case.

EXEC for Static Compromisation. In the static case, a set of machines and links is already compromised at the beginning of the execution and corruption commands are no longer available for the environment during the execution.

Compromised machines however can still create new machines. These new machines should be compromisable before the start to interact with the rest of the network. We therefore allow for a modified machine compromisation method in the static case, which only forwards the compromise command if it is the first message the newly created machine receives (see Fig. 8).

3.1.8 Runtime Bounds

Correctly addressing polynomial runtime bounds for networks of machines has been a point of major debate in the literature [26]. We adopt the solution put forward in [25, Section 6]. We only give a high level idea of the notion of a probabilistic polynomial time network and refer to the GNUC paper for a thorough discussion [25, Section 6].

We require that each message sent through the network begins with the string 1^η, where η is the security parameter used in the execution. If a machine is activated without a message, it receives the string 1^η on a special activation input port. We call a machine in the network **probabilistic, polynomial time** (PPT) if it is probabilistic and makes a polynomial number of steps in its input length on each activation.

We then limit the accumulated length of all messages sent through the network during the execution, denoted $\mathsf{Flow}_\eta[\Pi, \mathcal{A}, \text{Env}]$, to be bound by a polynomial in η, and call a protocol Π probabilistic,

Initialization: All input tapes are set to empty, all timer-variables are set to the initial value and no links are compromised.
Machine Activation: Every time a party $M_i \in \mathcal{M}$ gives control to the network execution, to current global time T_i for M_i is updated according to $T_i := T_i + \frac{n}{c_i}$, where n is the number of steps done by M_i in its last activation and c_i is its throughput.

upon m **on port** $p \neq net$ **from** $M_i \in \mathcal{M}$ **at time** T_i

if there is a machine with input port p **then**
  let id be the machine ID of the unique machine with input port p
  put (m, T_i, p) into Q_{id}
  activate M_{id}'s scheduler $\mathcal{A}_i$
else
  if $p = \mathsf{pid}(M_i).\mathsf{sid}(M_i).\mathsf{sid}' \wedge$ sid' proper extension of $\mathsf{sid}(M_i)$ **then**
    let $(\mathsf{protNAME}, sp) = \mathsf{basename}(\mathsf{sid}')$
    let $(cd, \mathcal{S}) = \Pi(\mathsf{protNAME})$
    sample speed c' from $\mathcal{S}$
    create a new machine M with code cd,
    $\mathsf{sid}(M) := \mathsf{sid}'$
    $c(M) := c'$
    set up translation of M's environment port to p
    put (m, T_i, p) into $Q_{\langle \mathsf{pid},\mathsf{sid}' \rangle}$
    activate M's scheduler $\mathcal{A}_i$
  else
    return error to M_i

upon (m, id) **on port** net **from** $M_i \in \mathcal{M}$ **at time** T_i

if M_{id} is a peer of M_i **then**
  if $(i, \mathsf{id}) \in \mathcal{C}$ **then**
    put (m, T_i, net) into $Q_{\mathcal{A}_i}$ of the corresponding sub-adversary $\mathcal{A}_i$.
    activate $\mathcal{A}_i$.
  else
    put (m, T_i, net) into Q_{id}
    activate M_{id}'s scheduler $\mathcal{A}_i$ with scheduling request
else
  return error to M_i

upon input $(\mathsf{activate}, M_j)$ **from** $\mathcal{A}_i \in \mathcal{A}$

if M_j in listen state **then**
  if Q_j is not empty **then**
    Pull next message (m, T_m, p) from Q_j
    if $T_m \leq T_j \wedge \forall M_i \in \mathcal{M} \setminus \{M_j\} : T_i \geq T_m$ **then**
      activate M_j with input m on port p.
    else
      activate $\mathcal{A}_i$ with a scheduling request
activate M_j without input

upon input (time) **from** $M_i \in \mathcal{M}$

retrieve T_i
compute local time $t_i := f_i(T_i)$.
activate M_i with input t_i on the time tape.

upon input $(\mathsf{increase})$ **from** $M \in \{\mathbf{Env}\} \cup \mathcal{A}$

set $T_M := T_M + 1$
activate M

upon (delay, m, t) **on port** q **from** $M_i \in \mathcal{M} \setminus \{\mathbf{Env}\} \cup \mathcal{A}$

if $t \geq f_i(T_i)$ **then**
  compute global time $T = f_i^{-1}(t)$
  execute appropriate send message for m with time stamp T for port q
else
  return error to M_i

upon activation by $M_i \in \mathcal{M}$ **without output**

activate M_i's scheduling machine $\mathcal{A}_i$ with scheduling request

upon input $(\mathsf{compromise}, \mathsf{id}_1, \mathsf{id}_2)$ **from Env**

$\mathcal{C} := \mathcal{C} \cup \{(\mathsf{id}_1, \mathsf{id}_2)\}$
activate Env with $(\mathsf{compromised}, \mathsf{id}_1, \mathsf{id}_2)$ as input.

upon $(\mathsf{compromise}, \mathsf{id}_1)$ **from parent of** M_{id_1}

if $M_{\mathsf{id}_1} \notin \mathcal{C}_M$ **then**
  replace M_{id_1} code to cd_{comp}
  $\mathcal{C}_M := \mathcal{C}_M \cup \{M_{\mathsf{id}_1}\}$
  send $(\mathsf{compromise})$ to M_{id_1}
else
  return error to Env

upon input (listen) **from** $M_i \in \mathcal{M}$

if Q_i is not empty **then**
  Pull next message (m, T_m, p) from Q_i
  if $T_m \leq T_i \wedge \forall M_j \in \mathcal{M} \setminus \{M_i\} : T_j \geq T_m$ **then**
    activate M_i with m on port p
  else
    Put (m, T_m, p) into Q_i
activate M_i's scheduler $\mathcal{A}_i$ with scheduling request

Figure 9: The full description of the execution EXEC for the time-sensitive network execution with adaptive compromisation. The machine set $\mathcal{M}$ denotes all machines, including environment and adversary, $\mathcal{A}$ denotes all machines in the adversary party.

polynomial-time, if the accumulated number of steps of all machines running Π at the end of the execution does not exceed a polynomial in $\mathsf{Flow}_\eta[\Pi, \mathcal{A}, \text{Env}]$.

This gives us a system of interacting Turing machines, which overall use a polynomial number of steps in the security parameter η.

3.2 Protocols

A protocol is a runtime library that assigns to each protocol name used in session IDs the respective program code to be executed by the respective machine and the speed of the machine that executes the code. Recall that a network has only one such runtime library, i.e., one protocol. This library assigns to all parties the respective code and speed for the single machines.

In the definition below, we denote with $Dist(\mathbb{N}) \subset \mathbb{N} \to [0,1]$ the set of distributions over the natural numbers (without 0).

Definition 2. *A **protocol** is a runtime library $\pi : D \to \{0,1\}^* \times Dist(\mathbb{N})$, which for every **protocol-name** d in its **domain** D gives the code $c \in \{0,1\}^*$ run by every machine with protocol name d and a efficiently computable **speed-distribution** $\mathcal{S} \in Dist(\mathbb{N})$ from which the execution* EXEC *can draw the speed coefficient for newly created machines.*
*A protocol π' is a **subprotocol** of π over domain D' if $D' \subseteq D$ and π restricted to D' equals π.*

A protocol Π also restricts the set of protocol-names $\{d_1, \dots d_l\} \subset D$ that a protocol-name $d \in D$ can call as subroutines. By the requirements listed in in [25, Section 5], these restrictions constitute an **acyclic call graph** on the protocol-names with a unique root r. We then call Π **rooted** at r. With this, the machine-trees representing a party in the network effectively are a **protocol-tree**, representing the different protocols and subprotocols used by a party for communication in the network.

Note that this design requires protocol names to be unique: machines having the same protocol name will execute the same code. If differentiation in behavior is required, this has to be either encoded in the session parameter included in the basename of each machine, or different protocol names have to be utilized.

Example 3: Protocol. Consider a network run by an execution EXEC with protocol Π and a machine M with machine ID $\mathsf{id}_M = (\mathsf{pid}, ((\mathsf{main}, x)))$ wants to invoke a new TLS connection to another machine in the network. M would then address a new machine M' with machine ID $\mathsf{id}_{M'} = (\mathsf{pid}, ((\mathsf{main}, x), (\mathsf{tls}, x')))$ over the port $\mathsf{pid}.((\mathsf{main}, x)).((\mathsf{main}, x), (\mathsf{tls}, x'))$. EXEC recognizes that M' does not yet exist and checks whether $((\mathsf{main}, x), (\mathsf{tls}, x'))$ is a proper extension of $((\mathsf{main}, x))$ (i.e. main is allowed to invoke tls as a subprotocol). If the check succeeds, EXEC creates a new machine, queries $(cd, \mathcal{S}) \leftarrow \Pi(\mathsf{tls})$ and assigns the new machine cd as its code and a throughput coefficient c' drawn from $\mathcal{S}$ as its speed. ◇

3.2.1 Composition

Composition of protocols is a useful tool for analyzing complex protocols by breaking them down into simpler to analyze, smaller sub protocols. In Section 4 we present the universal composability theorem which allows us to derive the security of a composed protocol from the security of its parts.

The following definitions are in line with the definitions for composition in GNUC [25, Section 5].

Definition 3. *The **sub-protocol** $\Pi' = \Pi|x$ of Π is the restriction of Π to D', the set of all base-names reachable from the base-name x.*

We denote with $\Pi \setminus x$ the protocol over all protocol names which are reachable from the root r without going through a node with base-name x.

Definition 4. *Let $\Pi' = \Pi|x$ be a sub-protocol of Π and let Π'_1 be a protocol rooted at x. Π'_1 is **substitutable** for Π' if for all $y \in D(\Pi \setminus x)$ it holds that $\Pi(y) = \Pi'_1(y)$*

We denote the substitution of Π' in Π as $\Pi_1 = \Pi[\Pi'/\Pi'_1]$. That is, $\Pi_1|x = \Pi'_1$ and $\Pi_1 \setminus x = \Pi \setminus x$.

Composition in our network model comes down to replacing sub-trees inside the machine-tree of a party. Figure 10 gives an example for such a substitution. On the protocol level, composition comes down to replacing the code provided for all base names in a sub-tree of the **acyclic call graph** of base names.

3.3 Ideal Functionalities

In our time-sensitive network model, machines which interact with each other might live in different points in time. This time-difference does not allow us to use a central ideal functionality as the abstraction of a multi-party protocol, as it is done in other asynchronous composability frameworks [9, 25].

To solve this problem, we require that every party contains a copy of the ideal functionality in its protocol tree, and all of these copies share a common state (see Section 3.1.6). Since such a copy of the

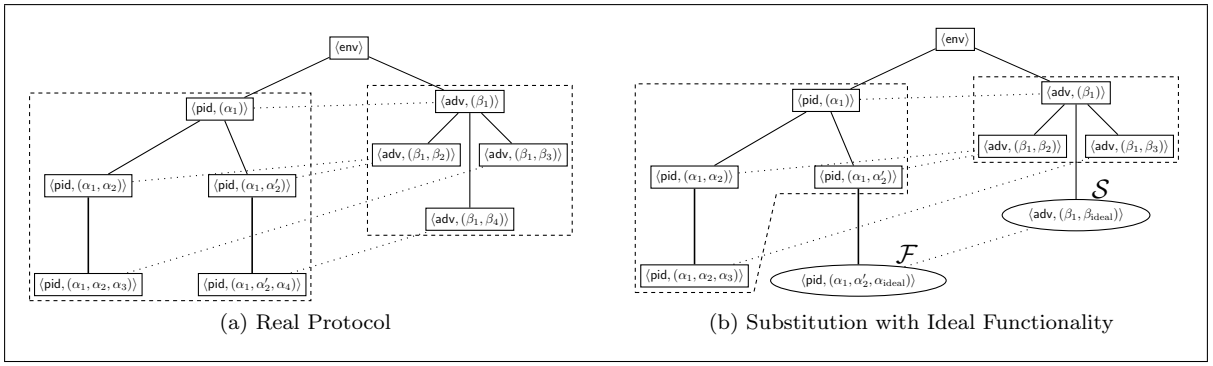

Figure 10: Substitution operation and the construction for the universal composability theorem – a sub-protocol is substituted for an ideal functionality $\mathcal{F}$ and its sub-adversary by the simulator $\mathcal{S}$ used in the universal composability proof

ideal functionality is part of the protocol tree, we allow the basenames of each machine to require ideal or real protocol code, instead of classifying ideal machines by their party IDs, as in GNUC [25].

We furthermore make the relaxation that we allow ideal machines to have children. Whenever ideal machines use common routines, such as communication channels, it is very convenient to be able to formalize such a routine as a child, e.g., as an ideal functionality for communication channels.

Apart from these changes, we adopt the restriction from GNUC that ideal machines can only communicate with ideal peers in the network and that ideal machines cannot be compromised by the adversary.

3.3.1 Centralized Ideal Functionalities and Dummy Nodes

The following counter example shows why using a central ideal functionality, as done in UC [9] or GNUC [25], does not work as soon as we allow the distinguisher to measure the response time. Thereafter, we show that using a replicated ideal functionality is as secure as using a central ideal functionality, and without having a notion of time, we show that having a central ideal functionality is as secure as having a replicated ideal functionality.

In order to be able to define a setting with centralized ideal functionalities, we need to introduce so-called **dummy nodes**. A dummy node is linked to one machine M, typically a (centralized) ideal functionality. Upon receiving a messages from its parent node, a dummy node forwards this message to the machine M. Analogously, upon receiving a message from the machine M, the dummy node forwards this message to its parent node. A dummy node can not have any children. Similar to functionality nodes, dummy node can not be compromised.

If dummy nodes were ordinary nodes they would always add one additional step by forwarding the messages. Hence, we treat dummy nodes as re-wirings, i.e., reroutings. Formally, dummy nodes do not have a machine timer, i.e., they live in all points in time at once.

A **centralized ideal functionality** is a machine without parent that is linked, as defined above, to dummy parties.

In contrast to a centralized ideal functionality, we call an ideal functionality as considered in our framework, i.e., that consists of several nodes with the same code and a shared memory, a **replicated ideal functionality**. We call each of these nodes a **replica** of that ideal functionality.

We define for every centralized ideal functionality a corresponding replicated ideal functionality. Each replica uses the same code of the centralized ideal functionality but we replace each memory access with an access to the shared memory. Analogously, we define for every replicated ideal functionality a centralized ideal functionality by using the one code that all machines share and replacing each access to the shared memory with access to the local memory.[2]

Example 4: Centralized ideal functionality. Assume that a real protocol is replaced by a dummy party and a centralized ideal functionality. Then, there is a pair of protocols Π and ideal functionality $\mathcal{F}$ such that Π cannot securely realize $\mathcal{F}$. The counter-example works for any pair of multi-party protocol Π and functionality $\mathcal{F}$ that answers with an acknowledgement message upon a single bit as input the subroutine (implemented by Π or $\mathcal{F}$, respectively); in particular Π and $\mathcal{F}$ do not do any communication.

[2]Formally, we actually require that the states of the Turing machine of the centralized ideal functionality are the cartesian product S^k of the states S of the Turing machine of the replicated ideal functionality, where k is the number of parties. Otherwise, moving the reading head of the program tape does not take the same amount of steps for the centralized ideal functionality.

Initially,let Env be in time t_0. Let f_i be the local-time functions of party i and c_i be the throughput (i.e., the speed) of party i. Let δ be the number of steps that the a ping operation takes. As a response towards this initial operation, party i sends an acknowledgment message `ack`. A environment Env that performs the following steps can distinguish the real protocol Π from the ideal functionality $\mathcal{F}$.

1: Send a bit 0 to party 1 in time t_0 over the environment input port towards party 1.
2: Send a bit 0 to party 2 in time t_0 over the environment input port towards party 2.
3: Proceed in time, and wait for the acknowledgement messages `ack` from party 2.
4: Measure the time at which the acknowledgement message arrives and store it in t_1.
5: Send a bit 0 to party 2 in time t_2.
6: Proceed in time, and wait for the acknowledgement messages `ack` from party 2.
7: Measure the time at which the acknowledgement message arrives and store it in t_3.
8: **if** $t_3 - t_2 > t_1 - t_0$ **then**
9: Output 0
10: **else**
11: Output 1

For the real protocol Π, both acknowledgement messages can be sent in parallel, i.e., $t_1 - t_0 = \delta/c_2 = t_3 - t_2 = \delta/c_2$. For a centralized ideal functionality $\mathcal{F}$, the activation order requires that first the first bit is processed $t_1 - t_0 = \delta/c_{\mathcal{F}} + \delta/c_{\mathcal{F}} > \delta/c_{\mathcal{F}} = t_3 - t_2$.[3] ⋄

Time-unaware network attackers and environments. As a next step, we show that for attackers and environments that cannot measure the time, a replicated ideal functionality is the same as a centralized ideal functionality. We call a machine **time-unaware** if it consists of a sandbox U_M that runs another machine M inside such that the sandbox intercepts all time measurements request towards the network execution and replaces the response of the network execution with 0. This lemma basically follows from the consistency requirement of a shared memory (see Section 3.1.6). We say $\pi \geq_{\mathrm{t}} \rho$ **for time-unaware attackers and environments** if for all PPT adversaries $\mathcal{A}$ there is a PPT simulator $\mathcal{S}$ such that for all PPT environments Env,

$$\mathsf{EXEC}(\pi, U_{\mathcal{A}}, U_{\text{Env}}) \approx \mathsf{EXEC}(\rho, \mathcal{S}, \mathcal{U}_{\text{Env}})$$

Corollary 5. *Let $\mathcal{F}_r$ be a replicated ideal functionality and $\mathcal{F}_c$ be the corresponding centralized ideal functionality. Then, $\mathcal{F}_r \geq_{\mathrm{t}} \mathcal{F}_c$ and $\mathcal{F}_c \geq_{\mathrm{t}} \mathcal{F}_r$ for time-unaware attackers and environments.*

Proof. This lemma directly follows from Corollary 1, i.e., from the consistency enforcing scheduling of a shared memory: every write operation is scheduled before any other read operations that take place by parties that are already in the future. □

This concludes the presentation of the time-sensitive network model used in TUC on which we want to base our time-sensitive analysis of anonymous communication protocols. The next section presents the security notion we will use for this analysis.

4 Secure Realization

We present the notion of security adopted in TUC and show that important properties such as the completeness of the dummy adversary and universal composability also hold in TUC.

In the same spirit as in other simulation-based frameworks, we adopt the notion of **secure realization**. A protocol π is compared to a simplified protocol ρ and is shown to be at least as secure: π securely realizes ρ, if every attack against π is also possible against ρ.[4]

More formally we require that the output distribution of the execution running the protocol π, an adversary $\mathcal{A}$ and an environment Env is indistinguishable from the output distribution of the execution running the simplified protocol ρ with a simulator $\mathcal{S}$ and the same environment Env. We define the indistinguishability of different execution as follows. This definition is a reformulation of the indistinguishability of binary random variable ensembles in [9].

[3] Reading the input and sending a response takes more than one step.

[4] Recall that the speed of a protocol party is determined by the protocol description (see Section 3.2), more specifically by speed-distribution that is assigned to every protocol role.

Definition 6 (Indistinguishability). *Two ensembles* $(\mathsf{EXEC}_\eta(\Pi, \mathcal{A}, \text{Env}, x))_{\eta\in\mathbb{N}}$ *and* $(\mathsf{EXEC}_\eta(\Pi', \mathcal{A}', \text{Env}', x))_{\eta\in\mathbb{N}}$ *are indistinguishable, denoted*

$$\mathsf{EXEC}(\Pi, \mathcal{A}, \text{Env}) \approx \mathsf{EXEC}(\Pi', \mathcal{A}', \text{Env}'),$$

if for every $c \in \mathbb{N}$ *there is a* $\eta_0 \in \mathbb{N}$ *such that for all security parameters* $\eta > \eta_0$ *and all inputs* x *we have that*

$$|Pr[\mathsf{EXEC}_\eta(\Pi, \mathcal{A}, \text{Env}, x) = 1] - Pr[\mathsf{EXEC}_\eta(\Pi', \mathcal{A}', \text{Env}', x) = 1]| < \eta^{-c}$$

Using this definition, we can now formalize secure realization.

Definition 7 (Secure Realization). *A protocol* π *securely realizes another protocol* ρ*, written* $\pi \geq_{\mathsf{t}} \rho$*, if for all* PPT *adversaries* $\mathcal{A}$ *there is a* PPT *simulator* $\mathcal{S}$ *such that for all* PPT *environments* Env

$$\mathsf{EXEC}(\pi, \mathcal{A}, \text{Env}) \approx \mathsf{EXEC}(\rho, \mathcal{S}, \text{Env})$$

As the notion of secure realization is based on the definition of indistinguishability above, we get as a direct consequence the transitivity and reflexivity of $\geq_{\mathsf{t}}$.

Corollary 8 (Transitivity and Reflexivity).

$$\Pi_1 \geq_{\mathsf{t}} \Pi_2 \wedge \Pi_2 \geq_{\mathsf{t}} \Pi_3 \implies \Pi_1 \geq_{\mathsf{t}} \Pi_3 \qquad \textit{and} \qquad \Pi \geq_{\mathsf{t}} \Pi$$

Proof. The theorem follows directly from the transitivity and reflexivity of $\approx$, on which we based our definition of $\geq_{\mathsf{t}}$: Due to $\Pi_1 \geq_{\mathsf{t}} \Pi_2$, there is a simulator $\mathcal{S}_1$ such that $\mathsf{EXEC}(\Pi_1, \mathcal{A}_d, \text{Env}) \approx \mathsf{EXEC}(\Pi_2, \mathcal{S}_1, \text{Env})$. But since $\Pi_2 \geq_{\mathsf{t}} \Pi_3$, there exists a simulator $\mathcal{S}_2$ for $\mathcal{S}_1$ such that $\mathsf{EXEC}(\Pi_2, \mathcal{S}_1, \text{Env}) \approx \mathsf{EXEC}(\Pi_3, \mathcal{S}_2, \text{Env})$ holds. Due to the transitivity of $\approx$ we therefore get as required

$$\forall \text{Env} : \ \mathsf{EXEC}(\Pi_1, \mathcal{A}_d, \text{Env}) \approx \mathsf{EXEC}(\Pi_3, \mathcal{S}_2, \text{Env}).$$

Reflexivity can be shown similarly. □

4.1 Properties of $\geq_{\mathsf{t}}$

In order to simplify the analysis of complex protocols, traditional composability frameworks depend on central properties of secure realization in their frameworks. We show that these properties hold in our model as well. The most important design decisions in this regard include making Env and $\mathcal{A}$ timeless (see Section 3) as well as having machines run with different performance coefficients (speed).

The proofs for the following results are largely the same as the proofs for their counterparts in classic composability frameworks, such as presented in [25]. However they need to be extended in order to account for timing.

4.1.1 Completeness of the Dummy Adversary

The definition of secure realization quantifies over all possible adversaries for the realizing protocol. In order to simplify this, we show that is is enough to only consider the dummy adversary $\mathcal{A}_d$, which just forwards all messages between environment and network parties.

Lemma 9 (Completeness of the dummy adversary). *If there exists an adversary* $\mathcal{S}$ *for a protocol* Π *such that for all environments* Env*,*

$$\mathsf{EXEC}(\Pi', \mathcal{A}_d, \text{Env}) \approx \mathsf{EXEC}(\Pi, \mathcal{S}, \text{Env})$$

then $\Pi' \geq_{\mathsf{t}} \Pi$.

Proof. Take any adversary $\mathcal{A}$ for Π. We split this adversary into two machines $\mathcal{A}'$ and $\mathcal{A}_d$ are simulated inside one adversary machine $\mathcal{A}^*$ s.t. $\mathcal{A}_d$ simply forwards all messages from the network to $\mathcal{A}'$, which in turn communicates with the environment and executes the same code as $\mathcal{A}$.

As $\mathcal{A}^*$ is timeless, it can simulate the interaction between $\mathcal{A}'$ and $\mathcal{A}_d$ without introducing any additional delays. As Env does not learn about the existence of A_d in any way, both of these scenarios are indistinguishable.

In the next step, we replace Env and $\mathcal{A}$' with a single machine Env', which internally simulates Env and $\mathcal{A}$': All messages incoming from $\mathcal{A}_d$ are internally sent to $\mathcal{A}$'. Then the communication between

Env and $\mathcal{A}$' is simulated and the output of Env' is defined as the output of the simulated instance of Env.

As Env' generates the same output as Env and does not generate any additional delays due to its timelessness, this third configuration is not distinguishable from the second above.

Now we are in a situation with some environment, the dummy adversary and the network. Thus we can use our assumption and replace Π with Π' and $\mathcal{A}_d$ with $\mathcal{S}$. By assumption, this fourth scenario is indistinguishable from above third.

Reverting Env' to Env and $\mathcal{A}'$ and combining $\mathcal{S}$ with $\mathcal{A}'$ to a new simulator $\mathcal{S}'$ then gives us the theorem. □

4.1.2 Composition Theorem

The central building block of simulation-based security is the notion of **composability**: the composition of secure protocols is secure as well. The construction in the proof is exemplified in Figure 10.

Theorem 10 (Composition Theorem)**.** *Let* Π *be a protocol and* $\Pi' = \Pi|x$ *a sub-protocol of* Π *rooted at* x*. Suppose that* Π'_1 *is a protocol rooted at* x *such that* $\Pi'_1 \geq_{\mathrm{t}} \Pi'$*. Then*

$$\Pi[\Pi' \setminus \Pi'_1] \geq_{\mathrm{t}} \Pi.$$

Proof. We construct a simulator $\mathcal{S}$ for Π, which tries to simulate the interaction between Π_1 and $\mathcal{A}_d$. By the completeness of the dummy adversary, this is enough to show realization.

Scenario 1: Original network consisting of parties running protocol Π, environment Env and the dummy adversary $\mathcal{A}_d$.

Scenario 2: We split the sub-protocol Π' from the protocol Π and the dummy-sub-adversary $\mathcal{A}_d$ that belongs to Π' from $\mathcal{A}_d$. We do not create separate entities, but just see that these are different machines with the following property: All scheduling requests and messages from Π' go directly to $\mathcal{A}'_d$.

As the network is otherwise the same as in Scenario 1, Scenario 2 and Scenario 1 are both indistinguishable.

Scenario 3: We define a new environment Env$'$ which internally simulates Env, all protocol parties not running Π', which we denote as $\Pi \setminus \Pi'$ and part of the dummy Adversary that does not communicate with Π', which we denote with $\mathcal{A}_d \setminus \mathcal{A}'_d$. Unfortunately, $\Pi \setminus \Pi'$ is not timeless, hence we will have to take care with the simulation and make sure that all timestamps are propagated correctly.

Env$'$ simulates the interaction of $\mathcal{A}_d \setminus \mathcal{A}'_d$, $\Pi \setminus \Pi'$ and Env as usual, but only increases its time whenever Env does. This especially means that messages from $\Pi \setminus \Pi'$ to Π' are not forwarded directly, but queued for sending when the time is set forward(that is, whenever Env increases its timer, Env' does not increases it timer by the full difference, but in minimal steps, making sure all messages are sent out at the right time). In order to make sure that Env and $\mathcal{A}_d \setminus \mathcal{A}'_d$ still get the correct messages from the rest of the network at the right time, Env$'$ also simulates the rest of the network internally, based on the messages sent out by $\Pi \setminus \Pi'$.

As in the proof of the dummy adversary, the output of Env$'$ is defined as the output of Env.

With this construction, all timestamps remain the same as in Scenario 2. As all messages exchanged are also the same, Scenario 3 and Scenario 2 are indistinguishable.

Scenario 4: In scenario 3 we have the situation where parties running protocol Π' communicate with the corresponding dummy adversary $\mathcal{A}'_d$ and the environment Env$'$. Using our assumption, we can now replace Π' with Π'_1 and $\mathcal{A}'_d$ with the simulator $\mathcal{S}'$ which is constructed in the realization proof of Π' being realized by Π'_1, while remaining indistinguishable from Scenario 3.

Scenario 5: We now split Env$'$ and recombine $\Pi \setminus \Pi'$ with Π'_1 to get Π_1 and $\mathcal{A}_d \setminus \mathcal{A}'_d$ with $\mathcal{S}'$ to get the simulator $\mathcal{S}$. Due to what Env$'$ simulated, Scenario 5 is indistinguishable to Scenario 4.

Using the transitivity of the indistinguishability, we get our claim. □

4.2 Joint State Theorem

A Joint State Theorem [10, 25] simplifies the analysis of a multi-session protocol Π which uses several instances of a single-session protocol π with a joint state between the multiple instances. For example, consider the situation where you give several sub-processes on your machine access to a key exchange process which always uses the same private-/public-key-pair. Here, the Joint State Theorem allows for the reduction of the security analysis of Π to the analysis of a single session ideal functionality $\mathcal{F}$ which is realized by π.

The Joint State Theorem also holds in our time-sensitive model. While the proof is along the lines of the proof in [25, Section 9], we have to make several changes to the general construction of multi-session functionalities due to the existence of time: we cannot combine several machines of the network into a single machine, as the single machine cannot adequately simulate the different timers of each machine it simulates. Instead we introduce the concept of an interface, through which communication to a set of nodes in the network is filtered. This allows us to view the machines behind such an interface as a combined unit while still preserving the independence of their timers.

4.2.1 Multi-Session Functionality

In order to analyze multi-session protocols with a joint-state, we will need to combine several instances of a single session functionality $\mathcal{F}$ into a single instance. The multi-session functionality $\hat{\mathcal{F}}$ collects instances of $\mathcal{F}$ in the protocol tree of a single party and combines them with a single interface to their callers. This interface filters and distributes incoming messages to the single instances of $\mathcal{F}$ based on virtual session IDs called **vsid**.

To this end, $\hat{\mathcal{F}}$ requires messages to be of the form $(m, vsid)$. The message m is then forwarded to the instance F of $\mathcal{F}$ with session id $vsid$. Any message m' going out from an instance F' of $\mathcal{F}$ is again brought into the form $(m', vsid')$, where $vsid'$ is the corresponding session id of F'.

In the regular network model without time, $\hat{\mathcal{F}}$ can be realized by a single machine which internally simulates the instances of $\mathcal{F}$, as done in GNUC [25]. In our model however, the existence of time makes this approach infeasible: Having one machine simulate all instances of $\mathcal{F}$ forces these instances to all live in the same point of time, as $\hat{\mathcal{F}}$ would not be able to simulate several instances of $\mathcal{F}$ living in different points in time. We would thus neglect all situations where the single sessions operate independently in time.

We solve this problem by instead introducing a interface through which all communication to the instances of $\mathcal{F}$ are filtered. This allows us to sum up all single instances of $\mathcal{F}$ into one entity $\hat{\mathcal{F}}$ while still preserving the time-independence of each instance. Note that the interface we add is omni-time, i.e. the interface receives messages at any point in time and works without progressing in time. Figure 11 exemplifies the construction of $\hat{\mathcal{F}}$.

4.2.2 Boxed Protocols

Unfortunately we get following problem as soon as we introduce multi-session protocols and functionalities: As these can be used by many different nodes in the protocol tree of a single party, the single caller rule that is used in GNUC [25] is no longer be enforced. As this rule is essential for the notion of composition and the universal composition theorem, we need to circumvent this problem by introducing so-called **boxed protocols**, as introduced in [25]: the protocol tree outside of the instances of $\mathcal{F}$ is boxed inside a single TM M, which simulates every node in the tree. Additionally, all messages sent to a multi-session protocol or functionality are modified to come from M. Messages received from the multi-session instance are distributed to the nodes in the tree by M based in the session ids used in the messages.

Again, as we consider time as well, we have to make small alterations: We cannot have a single TM simulate the full protocol tree, as a single TM cannot simulate several nodes which live in different points in time. We solve this problem by implementing boxing in terms of an additional, omni-time interface N which is put between a protocol Π and the multi-session instance. This interface basically works as a network address translation: ports used for communication between machines in the machine tree and instances of the ideal functionality are instead redirected to N: A message m incoming on input port $\mathsf{pid}.\mathsf{sid}_1.\mathsf{sid}_2$ is then forwarded as (m, sid_2) to the interface of $\hat{\mathcal{F}}$, which receives this message on its environmental port. On the other hand, a message (m, sid) received from $\hat{\mathcal{F}}$ is forwarded as message m through the unique output port $\mathsf{pid}.\mathsf{sid}_1.sid_2$, where $sid_2 = sid$.

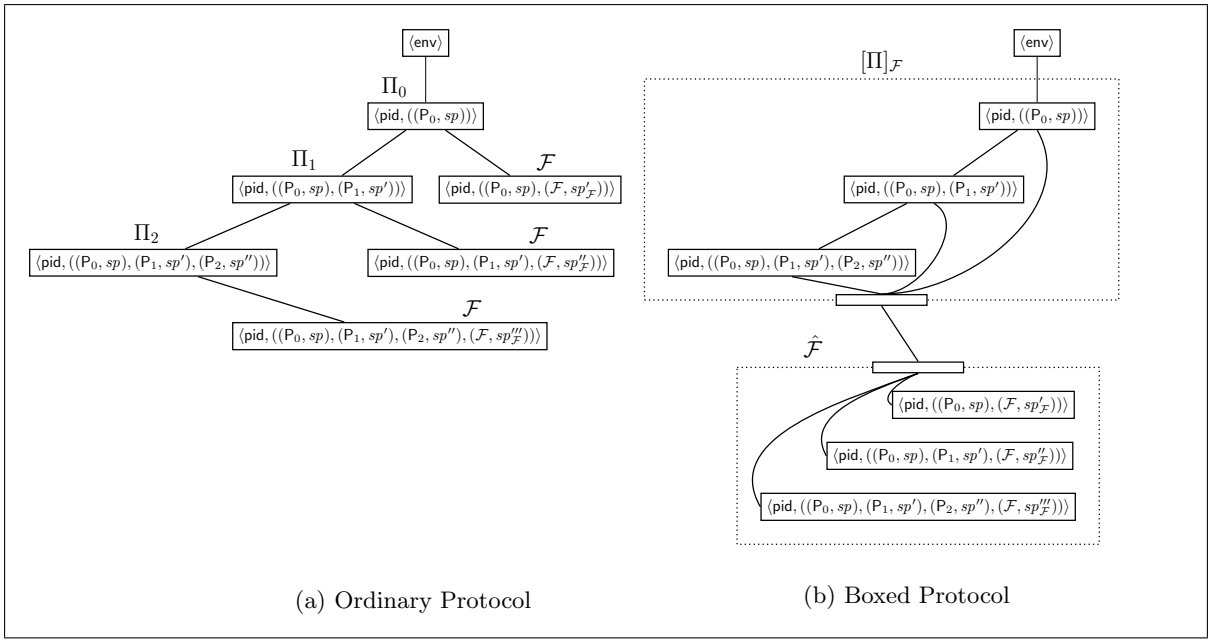

Figure 11: Boxed Protocol $[\Pi]_{\mathcal{F}}$ and Multi-Session Functionality $\hat{\mathcal{F}}$

From now on, we denote with $[\Pi]_{\mathcal{F}}$ the boxed protocol which consists of the $\mathcal{F}$-hybrid protocol Π, which is virtually boxed by the interface N as described above and interacts with the multi-session variant $\hat{\mathcal{F}}$ of $\mathcal{F}$. Using the boxing technique described above gives multi-session functionalities and protocols an interface through which they communicate with only a single caller, allowing us to keep the tree-like structure of parties in the network and use the same notions of composition we also use for single session functionalities and protocols. Figure 11 exemplifies the construction for the boxed protocol $[\Pi]_{\mathcal{F}}$ together with a multi-session functionality $\hat{\mathcal{F}}$.

In contrast to classic constructions, our boxing technique allows us to keep the time-independence of each single session instance of a multi session protocol. This can be used to more accurately model real world multi-session protocols, where for example different session are executed on different machines.

The classic construction would imply that all sessions are parallelized on the same machine and progress through time at the same pace. This can still be captured in our model by requiring that a set of sessions run on the same machine in $\hat{\mathcal{F}}$. This requires a small modification in the proof below, where set of instances F of $\mathcal{F}$ that are on the same machine in $\hat{\mathcal{F}}$ are delayed by the simulator by the same amount whenever a single instance $F_i \in F$ is activated.

4.2.3 Timed Joint State Composition Theorem

We now present the Joint State Composition Theorem in the time-sensitive setting.

Theorem 11 (Timed Joint State Composition Theorem)**.** *Let $\mathcal{F}$ be a poly-time ideal functionality and Π be a poly-time, $\mathcal{F}$-hybrid protocol. Then $[\Pi]_{\mathcal{F}} \geq_{\mathsf{t}} \mathcal{F}$.*

Proof. We construct a simulator $\mathcal{S}$ such that following holds:

$$\mathsf{EXEC}(\Pi, \mathcal{S}, \textsc{Env}) \approx \mathsf{EXEC}([\Pi]_{\mathcal{F}}, A_d, \textsc{Env})$$

Due to our constructions above, everything remains the same between both scenarios, if $\mathcal{S}$ works the same as the dummy adversary. Boxing only introduces virtual interfaces which do not create additional delays or change the behavior of parties. As the virtual interfaces only communicate with each other and cannot be addresses by the environment, both scenarios are indistinguishable. □

This concludes the introduction of secure realization in TUC and the presentation of its properties. The next sections exemplify the use of TUC based on a security analysis for Tor [45].

upon input (setup):

1: Generate an asymmetric key pair $(sk_P, pk_P) \leftarrow G(1^\eta)$.
2: send a cell $(\mathsf{register}, P, pk)$ to the $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ functionality
3: **wait for** a cell $(\mathsf{registered}, \langle P_j, pk_j \rangle_{j=1}^n)$ from $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$
4: output $(\mathsf{ready}, \mathcal{N} = \langle P_j \rangle_{j=1}^n)$

upon input $(\mathsf{createcircuit}, \mathcal{P} = \langle P, \langle P_j \rangle_{j=1}^\ell \rangle)$:

1: set *Start* to the current time
2: store $\mathcal{P}$ and $\mathcal{C} \leftarrow \langle P \rangle$; call $\mathit{ExtendCircuit}(\mathcal{P}, \mathcal{C})$

upon input $(\mathsf{send}, \mathcal{C} = \langle P \overset{cid_1}{\Longleftrightarrow} P_1 \Longleftrightarrow \cdots P_\ell \rangle, m)$:

1: **if** $\mathit{Start}(cid_1) + ttl_C >$ current time **then**
2: look up the keys $(\langle k_j \rangle_{j=1}^\ell)$ for cid_1
3: $O \leftarrow \mathit{WrOn}(m, (k_j)_{j=1}^\ell)$
4: send a cell $(cid_1, \mathsf{relay}, O)$ to P_1 over $\mathcal{F}_{\text{SCS}}$
5: **else**
6: call $\mathit{DestroyCircuit}(\mathcal{C}, cid_1)$
7: output $(\mathsf{destroyed}, \mathcal{C}, m)$

Figure 12: Π_{OR}: User commands for a Party P

5 Time Sensitive Analysis of the Onion Routing Protocol

In this section we take the framework we presented in Sections 3 and 4 and exemplify its use in the time-sensitive analysis of the anonymous communication protocol Tor [45].

Anonymous communication protocols as provided by the Tor network are an increasingly popular way for users to protect their privacy by hiding the user's location. The Tor network is currently used by hundreds of thousands of users around the world [44]. Sudden increases in Tor's usage have furthermore been shown to correlate to privacy-invasive political events which demonstrates the global political importance of the Tor network [13].

Section 5.1 defines the onion routing (OR) protocol as a protocol Π_{OR} in the TUC framework. Section 5.2 presents our abstraction of the OR protocol by defining the ideal functionality $\mathcal{F}_{\text{OR}}$. Finally, Section 5.3 presents two statements: (*i*) Theorem 16 states that Π_{OR} securely realizes $\mathcal{F}_{\text{OR}}$, i.e., we show that the abstraction $\mathcal{F}_{\text{OR}}$ is sound in the sense that every attack against Π_{OR} can also be mounted against $\mathcal{F}_{\text{OR}}$; (*ii*) Theorem 17 states that $\mathcal{F}_{\text{OR}}$ securely realizes Π_{OR} (for specific delay functions), i.e., we show that the abstraction $\mathcal{F}_{\text{OR}}$ is complete in the sense that every attack against $\mathcal{F}_{\text{OR}}$ (with specific delay functions) can be mounted against the actual onion routing protocol Π_{OR}.

Considering a time-sensitive adversary imposes new challenges on the analysis of complex, cryptographic communication protocols. Previous work [2] required cryptographic properties from the onion algorithms and the key exchange that ensure authenticity, integrity, secrecy and unlinkability. Against an adversary that can measure the time of a computation, we have to additionally require that the computation time of an encryption does not leak anything about the plaintext message, and we have to require that the DDH exponentiation does not leak anything about the exponents. We rigorously formalize these requirements in Section 5.3.1.

5.1 The Onion Routing Protocol

The core idea behind Tor is that, instead of directly communicating with the target, the user reroutes his traffic over a sequence of three onion routers. Smart use of cryptography then ensures that each participant in this chain only knows about his predecessor and successor, thus enabling anonymous communication.

Tor centrally organizes and validates available OR nodes and distributes their public keys to users. After the initial set up in which public keys of the onion routers (OR) are distributed, Tor works in two phases: In the first phase, the user establishes temporary symmetric keys with each of the three chosen onion routers, using the public keys of the ORs in a one-way authenticated key-exchange (1W-AKE) [21].[5] The sequence of ORs together with these symmetric keys is called a **circuit**. The exchanged keys are only used for one session, which typically lasts 10 minutes; then, fresh keys are established and the old keys are securely erased.[6]

[5] Tor is currently migrating to a more efficient and more secure 1W-AKE scheme (the ntor protocol). Recent work (the Ace scheme [4]) further improves on ntor.

[6] Temporary keys enable immediate forward secrecy: after a session is dead (and its temporary key and its communication transcripts is securely erased) even compromised parties do not leak anything about previous communications.

ExtendCircuit($\mathcal{P} = \langle P_j \rangle_{j=1}^{\ell}, \mathcal{C} = \langle P \overset{cid_1,k_1}{\Longleftrightarrow} P_1 \overset{k_2}{\Longleftrightarrow} \cdots P_{\ell'} \rangle$):

determine the next node $P_{\ell'+1}$ from $\mathcal{P}$ and $\mathcal{C}$
if $P_{\ell'+1} = \perp$ **then**
  output $(\mathsf{created}, \langle P \overset{cid_1}{\Longleftrightarrow} P_1 \Longleftrightarrow \cdots P_{\ell'} \rangle)$
else
  $X \leftarrow \mathit{Initiate}(pk_{P_{\ell'+1}}, P_{\ell'+1})$
  if $P_{\ell'+1} = P_1$ **then**
    $cid_1 \leftarrow \{0,1\}^{\kappa}$
    send a cell $(cid_1, \mathsf{create}, X)$ to P_1 over $\mathcal{F}_{\text{SCS}}$
  else
    $O \leftarrow \mathit{WrOn}(\{\mathsf{extend}, P_{\ell'+1}, X\}, (k_j)_{j=1}^{\ell'})$
    send a cell $(cid_1, \mathsf{relay}, O)$ to P_1 over $\mathcal{F}_{\text{SCS}}$

DestroyCircuit($\mathcal{C}, cid$):

if $\mathit{next}(cid) = (P_{next}, cid_{next})$ **then**
  send a cell $(cid_{next}, \mathsf{destroy})$ to P_{next} over $\mathcal{F}_{\text{SCS}}$
else if $\mathit{prev}(cid) = (P_{prev}, cid_{prev})$ **then**
  send a cell $(cid_{prev}, \mathsf{destroy})$ to P_{prev} over $\mathcal{F}_{\text{SCS}}$
discard $\mathcal{C}$ and all streams

Figure 13: Subroutines of Π_{OR} for Party P

In the second phase, the user performs a layered encryption of each message block, and sends the ciphertext, called **onion**, through the established circuit, where each OR decrypts one layer of encryption to learn where the onion should be sent next.

Using the same TCP stream that the last onion router in the circuit opened, the recipient can respond: in this case, each onion router adds a layer of encryption and the user removes all layers.

As presented in [2], the onion routing protocol used in Tor can be formalized by the protocol Π_{OR} presented in Figures 12, 14 and 13. Π_{OR} closely follows the Tor specification [14] and (for simplicity reason) assumes a fixed number $\mathcal{N}$ of protocol participants. We further assume that every party can be both user as well as onion router. We denote the subprotocol of the user as **onion proxy** (OP).

In contrast to the presentation in [2], we do not need to approximate the **time-to-live** (denoted as ttl_C) of a circuit $\mathcal{C}$ as the number of messages a user can send through $\mathcal{C}$: since the network model we introduce further down includes the notion of time, ttl_C can directly give the time for which a circuit can live before it is torn down (e.g. 10 minutes).

The protocol Π_{OR} uses several cryptographic algorithms in order to realize its different tasks: For the 1W-AKE, Π_{OR} uses the three algorithms *Initiate*, *Respond* and *ComputeKey*, which we introduce further below.[7] For adding and removing encryption layers to the payload (plaintext or onion), i.e. as principal onion algorithms, Π_{OR} uses the two algorithms *WrOn* and *UnwrOn*. *WrOn* creates a layered encryption of the payload, given an ordered list of ℓ session keys for $\ell \geq 1$. *UnwrOn* removes ℓ layers of encryptions from an onion to output the payload, given an input onion and an ordered list of ℓ session keys for $\ell \geq 1$.

We consider two kinds of messages in the description of Π_{OR}: network messages and user inputs. Network messages are used by the protocol to exchange **cells** between the parties. These are used for protocol level interactions such as creating a circuit or relaying a message. Input messages are sent by a user to his onion proxy in order to initiate a circuit construction or the sending of a message.

Circuits in Π_{OR}. A circuit $\mathcal{C}$ is represented in Π_{OR} by a sequence of **circuit ids** ($cid \in \{0,1\}^*$), each of which is know only to two consecutive nodes in the circuit $\mathcal{C}$. At a node P_i we denote an established circuit using the terminology $\mathcal{C} = P_{i-1} \overset{cid_i,k_i}{\Longleftrightarrow} P_i \overset{cid_{i+1}}{\Longleftrightarrow} P_{i+1}$. Here, P_{i-1} and P_{i+1} are the predecessor and successor of P_i in the circuit $\mathcal{C}$ and k_i is the session key established between P_i and the OP (who initiated this circuit). The absence of k_{i+1} indicates that the session key between P_{i+1} and the OP is not known to P_i. The functions *prev* and *next* on *cid* correspondingly give information about the predecessor or successor of the current node with respect to *cid*; e.g., $\mathit{next}(cid_i)$ returns (P_{i+1}, cid_{i+1}) and $\mathit{next}(cid_{i+1})$ returns $\perp$.

In the next section we go into detail about the different messages exchanged in Π_{OR}.

[7] Tor currently uses the TAP protocol and is going to switch to the more efficient and secure `ntor` protocol. [21]

upon receiving a cell $(cid, \mathsf{create}, X)$ from P_i over $\mathcal{F}_{\text{SCS}}$:

1: $\langle Y, k_{\mathsf{new}} \rangle \leftarrow Respond(pk_P, sk_P, X)$
2: store $\mathcal{C} \leftarrow \langle P_i \overset{cid,k_{\mathsf{new}}}{\Longleftrightarrow} P \rangle$
3: send a cell $(cid, \mathsf{created}, Y, t)$ to P_i over $\mathcal{F}_{\text{SCS}}$

upon receiving a cell $(cid, \mathsf{created}, Y, t)$ from P_i over $\mathcal{F}_{\text{SCS}}$:

1: **if** $prev(cid) = (P', cid', k')$ **then**
2: $O \leftarrow WrOn(\langle \mathsf{extended}, Y, t \rangle, k')$
3: send a cell $(cid', \mathsf{relay}, O)$ to P' over $\mathcal{F}_{\text{SCS}}$
4: **else if** $prev(cid) = \perp$ **then**
5: $k_{\mathsf{new}} \leftarrow ComputeKey(pk_i, Y, t)$
6: update $\mathcal{C}$ with k_{new}; call $ExtendCircuit(\mathcal{P}, \mathcal{C})$

upon receiving a msg (sid, m) from the parent node:

1: get $\mathcal{C} \leftarrow \langle P' \overset{cid,k}{\Longleftrightarrow} P \rangle$ for sid; $O \leftarrow WrOn(m, k)$
2: send a cell (cid, relay, O) to P' over $\mathcal{F}_{\text{SCS}}$

upon receiving a cell $(cid, \mathsf{destroy})$ from P_i over $\mathcal{F}_{\text{SCS}}$:

1: call $DestroyCircuit(\mathcal{C}, cid)$

upon receiving a cell (cid, relay, O) from P_i over $\mathcal{F}_{\text{SCS}}$:

1: **if** $prev(cid) = \perp$ **then**
2: **if** $getkey(cid) = (k_j)_{j=1}^{\ell'}$ **then**
3: (type, m) **or** $O \leftarrow UnwrOn(O, (k_j)_{j=1}^{\ell'})$
4: (P', cid') **or** $\perp \leftarrow next(cid)$
5: **else if** $prev(cid) = (P', cid', k')$ **then**
6: $O \leftarrow WrOn(O, k')$ /* a backward onion */
7: **switch** (type)
8: **case** extend:
9: get $\langle P_{next}, X \rangle$ from m; $cid_{next} \leftarrow \{0,1\}^\kappa$
10: update $\mathcal{C} \leftarrow \langle P_i \overset{cid,k}{\Longleftrightarrow} P \overset{cid_{next}}{\Longleftrightarrow} P_{next} \rangle$
11: send a cell $(cid_{next}, \mathsf{create}, X)$ to P_{next} over $\mathcal{F}_{\text{SCS}}$
12: **case** extended:
13: get $\langle Y, t \rangle$ from m; get P_{ex} from $(\mathcal{C}, \mathcal{P})$
14: $k_{\mathsf{ex}} \leftarrow ComputeKey(pk_{\mathsf{ex}}, Y, t)$
15: update $\mathcal{C}$ with (k_{ex}); call $ExtendCircuit(\mathcal{P}, \mathcal{C})$
16: **case** data:
17: **if** (P = OP) **then** output $(\mathsf{received}, \mathcal{C}, m)$
18: **else**
19: generate or lookup the unique sid for cid
20: output $(\mathsf{received}, (P, \mathsf{sid}, m'))$
21: **case** *corrupted*: /*corrupted onion*/
22: call $DestroyCircuit(\mathcal{C}, cid)$
23: **case** default: /*encrypted forward/backward onion*/
24: send a cell $(cid', \mathsf{relay}, O)$ to P' over $\mathcal{F}_{\text{SCS}}$

Figure 14: Π_{OR}: Network messages for Party P

5.1.1 User inputs

In this section, we present the commands that a user can send: a initialization command (setup), a command for circuit creation (createcircuit), and a command for sending message (send).

Key registration. Upon an input (setup), an OR node computes its long-term keys (sk, pk) and registers these keys.

In Π_{OR} the key registration and distribution is modeled as an ideal functionality $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$, which is defined as in [9] with the exception that $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ rejects all parties not in $\mathcal{N}$ and only distributes the public keys after all parties in $\mathcal{N}$ have registered with $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$. As soon as all parties have registered, each of them receives the message $(\mathsf{registered}, \langle P_j, pk_j \rangle_{j=1}^n)$, which contains a list of all valid OR nodes, together with their public keys.

Circuit creation. Upon an input createcircuit (see Figure 12), the OP starts the circuit creation process, which consists of the 1W-AKE for establishing the session key, and the actual circuit creation: The OP, as the initiator, runs the *Initiate* algorithm to draw new key-exchange information and sends this to the first node of the circuit inside a create cell (see Figure 13). The first node then runs the *Respond* algorithm and responds with a created cell. After receiving this response, the OP runs the *ComputeKey* algorithm to compute the session key.

For extending a circuit past the first node, the OP runs the *Initiate* algorithm and sends an extend relay cell, which causes the currently last node of the circuit to send a create cell to the next node and so on.

Sending messages. Communication in the forward direction is initiated by a send message from the user to his OP, while communication in the backward direction is initiated by a network message to the exit node (from the recipient).

upon input $(\mathsf{send}, \mathcal{C} = \langle P \overset{cid_1}{\Longleftrightarrow} P_1 \Longleftrightarrow \cdots P_\ell \rangle, m)$:

1: **if** $Start(cid_1) + ttl_C >$ current time **then**
2: let $t \leftarrow v[4]$
3: $SendMessage(P_1, cid_1, \mathsf{relay}, \langle \mathsf{data}, m \rangle, t)$
4: **else**
5: $DestroyCircuit(\mathcal{C}, cid_1)$
6: lookup the current time t
7: let $t' \leftarrow v[5]$
8: output $(\mathsf{destroyed}, \mathcal{C}, m)$ at time $t + t'$

upon input $(\mathsf{createcircuit}, \mathcal{P} = \langle P, P_1, \ldots, P_\ell \rangle)$:

1: set *Start* to the current time
2: store $\mathcal{P}$ and $\mathcal{C} \leftarrow \langle P \rangle$
3: let $t \leftarrow v[3]$
4: $ExtendCircuit(\mathcal{P}, \mathcal{C}, t)$

upon input (setup):

1: draw a fresh handle h; set $\mathsf{registered\_flag} \leftarrow true$
2: store $lookup(h) \leftarrow (\mathsf{dir}, \mathsf{registered}, \mathcal{N})$
3: lookup the current time t; let $t' \leftarrow v[1]$
4: send $(h, \mathsf{register}, P)$ to the network at time $t + t'$
5: **wait for** a msg $(\mathsf{dir}, \mathsf{registered}, \mathcal{N})$ via a handle
6: lookup the current time t
7: let $t' \leftarrow v[2]$
8: output $(\mathsf{ready}, (P_j)_{j=1}^n) = (\mathsf{ready}, \mathcal{N})$ at time $t+t'$

Figure 15: The ideal functionality $\mathcal{F}_{\text{OR}}^{\mathcal{N}}$ (short $\mathcal{F}_{\text{OR}}$) for Party P: Input messages

5.1.2 Network Messages

In Tor, each pair of onion routers establishes a TLS connection for ensuring the integrity of onions and for hiding the circuit identifiers from a network observer. In Π_{OR}, we abstract such a TLS connection by a functionality $\mathcal{F}_{\text{SCS}}$ as proposed by Canetti [9].[8]

Communication between servers (outside of the Tor network) and exit nodes (i.e., the last OR in the circuit) is synchronized using TCP streams. Π_{OR} abstracts from these streams by introducing a session identifier sid.

Relay cells. relay cells are used for tunneling commands such as data, extend and extended through an established (part of a) circuit. Communication between the OP and the exit node in the forward direction is implemented via a *WrOn* call with with all session keys exchanged during the circuit creation, and a series of *UnwrOn* calls at each of the ORs in the circuit with the individual session keys they know.

In the backward direction communication is implemented using a series of *WrOn* calls by the ORs in the network with the individual session keys, and finally a *UnwrOn* call at the OP.

Tearing down a circuit. To tear down a circuit (e.g. if a session expires after ttl_C time), an OR or OP sends the destroy cell to the neighboring nodes in the circuit along with the corresponding *cid* (see Figure 13). Upon receiving a destroy cell, the node frees resources associated with the corresponding circuit. Once the destroy cell has been processed, the node ignores all future cells from the corresponding circuit.

destroy cells are also sent through the circuit in case an integrity check fails during an *UnwrOn* call. A failed integrity check means that the adversary somehow tinkered with the onion that was being processed, and Π_{OR} counters this by dropping the affected circuit and creating a new one.

This concludes the presentation of the onion routing protocol. We will use it in Section 5.3, where we show soundness and completeness of our time-sensitive abstraction of Tor we present in the next section.

5.2 Time-sensitive Abstraction of OR

Tor is a low latency communication protocol and hence is prone to all kinds of traffic pattern analyses, such as traffic confirmation attacks or website fingerprinting attacks. As in previous work, we want to accurately model all weaknesses of the OR protocol. As a consequence, our anonymous channel functionality has to leak all these communication patterns, while still abstracting from all cryptographic operations, thereby allowing to accurately capture the leakage of the OR protocol and the capabilities of a time-sensitive adversary.

Our abstraction goes along the lines of previous work [2]. However we additionally have to compensate for computation time differences that appear in the ideal functionality: the ideal functionality does not

[8] The leakage function l for $\mathcal{F}_{\text{SCS}}$ we use here is $l(m) := |m|$.

upon receiving a handle $\langle P, P_{next}, h\rangle$ from the network:

lookup the current time t; let $t' \leftarrow v[6]$; send $msg \leftarrow lookup(h)$ to submachine P_{next} at time $t + t'$

upon receiving a msg $(P_i, cid, \mathsf{create})$ through a handle:

store $\mathcal{C} \leftarrow \langle P_i \overset{cid}{\Longleftrightarrow} P\rangle$
let $t \leftarrow v[7]$
$SendMessage(P_i, cid, \mathsf{created}, t)$

upon receiving a msg $(P_i, cid, \mathsf{created})$ through a handle:

if $prev(cid) = (P', cid')$ **then**
  let $t \leftarrow v[8]$
  $SendMessage(P', cid', \mathsf{relay}, \mathsf{extended}, t)$
else if $prev(cid) = \perp$ **then**
  let $t \leftarrow v[9]$
  $ExtendCircuit(\mathcal{P}, \mathcal{C}, t)$

upon receiving a msg (sid, m) from the parent node:

obtain $\mathcal{C} = \langle P' \overset{cid}{\Longleftrightarrow} P\rangle$ for sid
let $t \leftarrow v[16]$
$SendMessage(P', cid, \mathsf{relay}, \langle \mathsf{data}, m\rangle, t)$

upon receiving a msg $(P_i, P, h, [\mathsf{corrupt}, T(\cdot)])$ from $\mathcal{A}$:

$(message) \leftarrow lookup(h)$
if $\mathsf{corrupt} = true$ **then**
  $message \leftarrow T(msg)$
  set $corrupted(message) \leftarrow true$
process $message$ as if received normally

upon receiving a msg $(\mathsf{compromise})$ from $\mathcal{A}$:

set $compromised \leftarrow true$; delete local information

upon receiving a msg $(\mathsf{output}, (P, \mathsf{sid}, m))$ through a handle:

let $t \leftarrow v[15]$; output $(\mathsf{received}, (\mathsf{sid}, m), t)$

upon receiving a msg $(P_i, cid, \mathsf{relay}, O)$ through a handle:

if $prev(cid) = \perp$ **then**
  if $next(cid) = \perp$ **then** get (type, m) from O
  else $\{P', cid'\} \leftarrow next(cid)$
else $(P', cid') \leftarrow prev(cid)$
switch (type)
case extend:
  get P_{next} from m; $cid_{next} \leftarrow \{0,1\}^\kappa$
  update $\mathcal{C} \leftarrow \langle P_i \overset{cid}{\Longleftrightarrow} P \overset{cid_{next}}{\Longleftrightarrow} P_{next}\rangle$
  let $t \leftarrow v[10]$
  $SendMessage(P_{next}, cid_{next}, \mathsf{create}, t)$
case extended:
  update $\mathcal{C}$ with P_{ex}
  $ExtendCircuit(\mathcal{P}, \mathcal{C})$
case data:
  if (P = OP) **then** output $(\mathsf{received}, C, m)$
  else
    generate or lookup the unique sid for cid
    let $t \leftarrow v[11]$
    $SendMessage(P', cid', \mathsf{output}, (P, \mathsf{sid}, m'), t)$
case corrupt: /*corrupted onion*/
  let $t \leftarrow v[12]$
  $DestroyCircuit(\mathcal{C}, cid, t)$
case default: /*encrypted forward/backward onion*/
  let $t \leftarrow v[13]$
  $SendMessage(P', cid', \mathsf{relay}, O, t)$

upon receiving a msg $(P_i, cid, \mathsf{destroy})$ through a handle:

let $t \leftarrow v[14]$; $DestroyCircuit(\mathcal{C}, cid, t)$

Figure 16: The ideal functionality $\mathcal{F}_{\text{OR}}^{\mathcal{N}}$ (short $\mathcal{F}_{\text{OR}}$) for Party P: Network messages

perform any cryptographic operations and therefore often has less computation steps to perform than the real protocol. In the functionality we use the delayed sending commands presented in Figure 4 to compensate for these differences.

5.2.1 Review of $\mathcal{F}_{\text{OR}}$

The ideal functionality $\mathcal{F}_{\text{OR}}$, as presented in Figure 15, 16, and 17, is close to the OR protocol Π_{OR} presented in Section 5.1. Due to the similarities of Π_{OR} and $\mathcal{F}_{\text{OR}}$, we concentrate on highlighting the differences between them.

The major difference between Π_{OR} and $\mathcal{F}_{\text{OR}}$ is that $\mathcal{F}_{\text{OR}}$ does not use any cryptography: the session keys, the onion methods *WrOn* and *UnwrOn*, and 1W-AKE methods *Initiate*, *Respond*, and *ComputeKey* are absent in $\mathcal{F}_{\text{OR}}$.

In fact, $\mathcal{F}_{\text{OR}}$ does not need any cryptography: Instead of relying on the security of onion algorithms, messages are exchanged via shared memory: shared memory is an additional abstraction added to $\mathcal{F}_{\text{OR}}$, which allows all parties running $\mathcal{F}_{\text{OR}}$ to exchange messages "off-band".

Now if party P wants to send a message m to party P_{next}, P creates a fresh handle h, saves m in the shared memory under this handle and sends $\langle P, P_{next}, h\rangle$ over the network.

$\mathcal{F}_{\text{OR}}$ also does not require $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ for the initial distribution of public keys (it does not really need any public keys at all): instead, on input (setup), the party P notes its registration in the shared memory, and, as soon as all other parties in the network also noted their registration, outputs a successful registration to the caller.

ExtendCircuit($\mathcal{P} = (P_j)_{j=1}^{\ell}, \mathcal{C} = \langle P \overset{cid_1}{\Longleftrightarrow} P_1 \Longleftrightarrow \cdots P_{\ell'} \rangle$[, time]):

determine the next node $P_{\ell'+1}$ from $\mathcal{P}$ and $\mathcal{C}$
if $P_{\ell'+1} = \perp$ **then**
  output (created, $\mathcal{C}$)
else
if $P_{\ell'+1} = P_1$ **then**
  $cid_1 \leftarrow \{0,1\}^{\kappa}$
  SendMessage(P_1, cid_1, create[, time])
else
  SendMessage(P_1, cid_1, relay, {extend, $P_{\ell'+1}$}[, time])

DestroyCircuit($\mathcal{C}, cid$[, time]):

if $next(cid) = (P_{next}, cid_{next})$ **then**
  SendMessage(P_{next}, cid_{next}, destroy[, time])
else if $prev(cid) = (P_{prev}, cid_{prev})$ **then**
  SendMessage(P_{prev}, cid_{prev}, destroy[, time])
discard $\mathcal{C}$ and all streams

SendMessage(P_{next}, cid_{next}, cmd[, relay-type][, data][, time]):

create *msg* for P_{next} from input
draw a fresh handle h
set $lookup(h) \leftarrow msg$
if *compromised* = *true* **then**
  let P_{last} be the last node in the complete contiguous visible subpath path starting P_{next}
  if ($P_{last} = OP$) or P_{last} is the exit node and data $\neq \perp$ **then**
    $msg' \leftarrow lookup(h')$
    lookup the current time t; send the entire message $\langle P, P_{next}, \ldots, P_{last}, cid_{next}, \mathsf{cmd}, \mathsf{data} \rangle$ to $\mathcal{A}$ at time $t +$ time
  else send $\langle P, P_{next}, \ldots, P_{last}, cid_{next}, \mathsf{cmd}, h \rangle$ to $\mathcal{A}$
else send $\langle P, P_{next}, h \rangle$ to the network

Figure 17: Subroutines of $\mathcal{F}_{\text{OR}}$ for Party P

Compromising parties. A party running $\mathcal{F}_{\text{OR}}$ cannot be compromised: instead, upon receiving a compromise message from the adversary, the respective party sets its *compromised* variable to *true*. Then, all input or network messages that are visible to the compromised entity are forwarded to the adversary. In principle, the adversary runs that entity and can send messages from that entity.

Explicit leakage: visible subpaths. For proving soundness of $\mathcal{F}_{\text{OR}}$, we require a special behavior by compromised parties. In case the adversary manages to compromise an entire subpath S of a circuit, the first node in S needs to leak all information that would have been leaked by each node in S individually in the real world: the simulator constructed for the soundness proof in [2] does not learn about circuits constructed in the network and neither about the messages transmitted through the network. But the simulator would need this information for correctly simulating the behavior of the real parties (running Π_{OR}), if it only had the individual leakage of the parties in the compromised subpath.

We therefore have the **visible subpath** computation in the *SendMessage* function in Figure 17. Parties running $\mathcal{F}_{\text{OR}}$ share their *compromised*-status over the shared memory and based on this leak the required information to the adversary. Unfortunately, the soundness proof we present in Section 5.3 does not solve this problem, and hence we keep this construction.

Messages through a handle. Figure 16 considers messages m that are retrieved **through a handle**. As described above, $\mathcal{F}_{\text{OR}}$ uses shared memory in order to transmit messages through the network. A party P receives a message through a handle h if P found this message after looking up h in the shared memory.

Corrupted messages. While the adversary might corrupt or replay messages in Π_{OR}, these active attacks will be detected by the recipient due to the presence of a secure and authenticated channel between any two communicating parties. The interesting case is when the adversary manages to compromise an onion router in the circuit: the adversary can then propagate corrupted messages, which in Π_{OR} are only detected during *UnwrOn* calls at the OP or the exit node.

This fact is captured in $\mathcal{F}_{\text{OR}}$ by using *corrupted* flags for each message sent through the network. If the adversary wants to modify a message, this flag is set to *true* and propagated until it reaches the last node P_{last} in the circuit.

The adversary also provides a message transformation function $T(\cdot)$, which is applied to the message in the shared memory in order to change it.

5.2.2 Our Modifications to $\mathcal{F}_{\text{OR}}$

In order to correctly capture the notion of time in our abstraction, we modify some aspects of $\mathcal{F}_{\text{OR}}$ which did not allow for a direct translation into a time-sensitive abstraction.

Similar to the adjusted OR protocol Π_{OR}, we change the time-to-live (ttl_C) of a circuit to an actual time-interval (e.g., 10 minutes) instead of a bounding the number of messages that can be transmitted through the same circuit.

A major problem we face after introducing time is that Π_{OR} and $\mathcal{F}_{\text{OR}}$ take a different number of steps for executing specific commands (due to the differences in their code). This results in parties in different worlds (real and ideal) advancing in time with different paces. In order to still be able to show soundness and completeness of our abstraction, we therefore need to adjust the pace in which parties running $\mathcal{F}_{\text{OR}}$ advance in time.

We achieve this by introducing a **delay vector** $v = (d_1, \ldots, d_{15})$ with which we parameterize $\mathcal{F}_{\text{OR}}$. Each entry of v is a delay-distribution, which is inserted at specific points in the code of $\mathcal{F}_{\text{OR}}$ (see Figure 15). $\mathcal{F}_{\text{OR}}$ then draws the number of steps it should delay at these specific points whenever this piece of code is executed.

With this, we make sure that in the abstraction $\mathcal{F}_{\text{OR}}$ as well as in the protocol Π_{OR} the parties progress in time at roughly the same rate.

Special care has to be taken whenever we add delay for a function with a run-time which is not constant, e.g. if we add delay for the various encryption and decryptions methods from Π_{OR}. The delay can then depend on input provided by $\mathcal{F}_{\text{OR}}$. We explain this in more detail in Section 5.3.

In Section 3.1.1 we described how the speed coefficients for newly created machines in the network are determined (i.e. by drawing the speed coefficient from a distribution specific to the protocol role of the new machine). We have to account for these variable speeds by suitably varying the delay vector: the initial delay vectors are defined for fixed speed coefficient for ideal (c_i) and real (c_r) machines. After drawing the coefficient c for the newly crated machine, all delay vector entries are stretched by the factor $\frac{c_r}{c}$, then multiplied by a factor b which makes all entries integer, and increased by the factor $(\frac{c_r b}{c} - 1)s_i$, where s_i is the number of steps the ideal machine does on the activation until this specific delay vector entry kicks in. The new base speed coefficient for the ideal machine will be $c_i \cdot b$ and uses the previously computed delay vector.

We also need to adjust the visible subpath computation in the *SendMessage* function: in the original $\mathcal{F}_{\text{OR}}$ functionality the visible subpath was leaked before the message arrived the observed part of the network. We adjusted *SendMessage*() such that messages are only leaked after the compromised of the network is actually reached.

The ideal functionality $\mathcal{F}_{\text{OR}}$ for OR is tight. We show that it is sound and complete, i.e., we show that the protocol realizes the ideal functionality and the ideal functionality realizes the protocol. We stress that $\mathcal{F}_{\text{OR}}$ basically resembles the protocol except for the cryptographic operations. Instead of ciphertexts and group elements, the $\mathcal{F}_{\text{OR}}$ merely sends freshly drawn handles over the network. The predecessor of this $\mathcal{F}_{\text{OR}}$ has been used for analyzing the anonymity guarantees of the OR protocol, since all cryptographic operations are abstract away in a provably secure way [3].

5.3 Abstracting Tor in TUC

We show that $\mathcal{F}_{\text{OR}}$ is indeed an accurate abstraction of the onion routing protocol Π_{OR}. This includes showing that Π_{OR} securely realizes $\mathcal{F}_{\text{OR}}$ in TUC, which was already shown by Backes et al. [2] for the standard UC-framework. This gives us the soundness of the abstraction. Furthermore we show the other direction: $\mathcal{F}_{\text{OR}}$ securely realizes Π_{OR}, thus giving us completeness of our abstraction.

The soundness allows excluding attacks on the Tor protocol without having to deal with cryptographic operations in the analysis. The completeness on the other hand allows us to analyze the ideal functionality for finding attacks and then translating them to the Tor protocol.

5.3.1 Assumptions

In order to prove the following theorems, we need to make certain assumptions about the cryptographic primitives used in Π_{OR}. These assumptions were already presented in [2], but we require them to also hold against an adversary with timing information. We present these assumptions here and use them later in the proofs.

1W-AKE. We assume that the key exchange that happens whenever a new circuit is created uses a 1W-AKE-protocol as introduced in [21]. From these we need the property of **key secrecy**: for an adversary, which observes the public parts of the key exchange, the generated key is indistinguishable from a randomly chosen one.

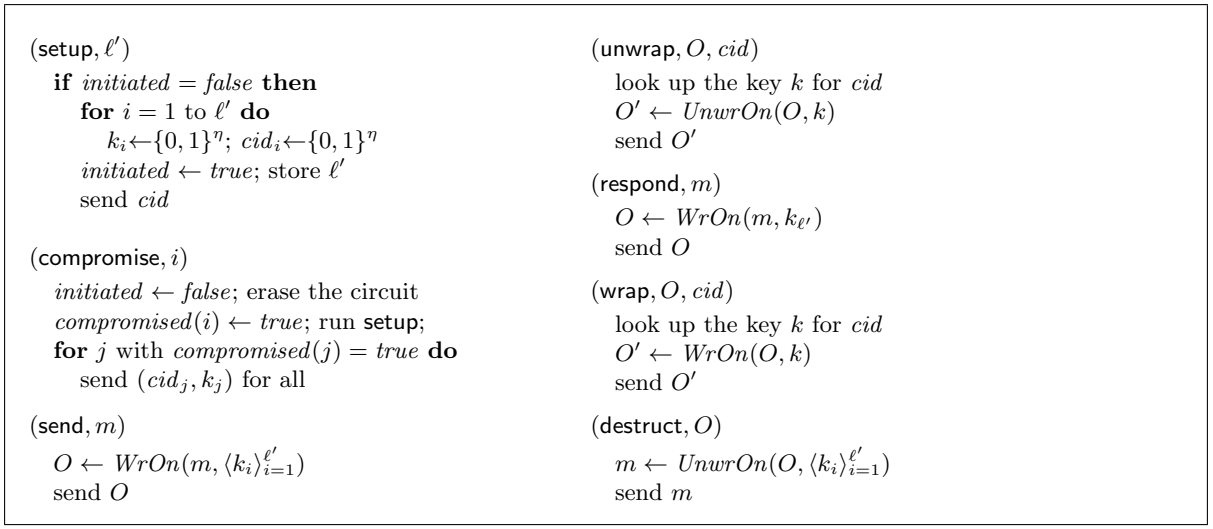

Figure 18: The Honest Onion Secrecy Challenger $\mathsf{OS\text{-}Ch}^0$: $\mathsf{OS\text{-}Ch}^0$ only answers for honest parties

We assume that the encryption and decryption algorithms used in the onion routing protocol Π_{OR} to be secure, i.e. they satisfy following four properties, as presented in [2]. As we consider a time sensitive network model, we assume that these assumptions also hold against time sensitive adversaries:

Onion Correctness. The first property of secure onion algorithms is **onion correctness**. It states that honest wrapping and unwrapping results in the same message. Moreover, the correctness states that whenever the unwrapping algorithm has a fake flag, it does not care about integrity, because for $m \in M(\eta)$ the integrity measure is always added, as required by the end-to-end integrity. But for $m \notin M(\eta)$ but of the right length, the wrapping is performed without an integrity measure. The fake flag then causes the unwrapping to ignore the missing integrity measure. Then, we also require that the state transition is independent from the message or the key.

Definition 12 (Onion correctness)**.** *Let $M(\eta)$ be the message space for the security parameter η. Let $\langle k_i \rangle_{i=1}^{\ell}$ be a sequence of randomly chosen bitstrings of length η.*

Forward: $\Omega_f(m)$
$O_1 \leftarrow WrOn(m, \langle k_i \rangle_{i=1}^{\ell})$
for $i = 1$ *to* ℓ ***do***
$O_{i+1} \leftarrow UnwrOn(O_i, k_i)$
$x \leftarrow O_{\ell+1}$

Backward: $\Omega_b(m)$
$O_\ell \leftarrow WrOn(m, k_\ell)$
for $i = \ell - 1$ *to* 1 ***do***
$O_i \leftarrow WrOn(O_{i+1}, k_i)$
$x \leftarrow UnwrOn(O_1, \langle k_i \rangle_{i=1}^{\ell})$

Let Ω'_f be the defined as Ω_f except that UnwrOn additionally uses the fake *flag. Analogously, Ω'_b is defined. We say that a pair of onion algorithms $(WrOn, UnwrOn)$ is* ***correct*** *if the following three conditions hold:*

(i) $\Pr[x \leftarrow \Omega_d(m) : x = m] = 1$ *for* $d \in \{f, b\}$ *and* $m \in M(\eta)$.

(ii) $\Pr[x \leftarrow \Omega'_d(m) : x = m] = 1$ *for* $d \in \{f, b\}$ *and all* $m \in M'(\eta) := \{m' | \exists m'' \in M(\eta).|m'| = |m''|\}$.

(iii) For all $m \in M'(\eta)$, $k, k' \in \{0,1\}^\eta$ *and* $c, s \in \{0,1\}^*$ *such that c is a valid onion and s is a valid state*

$$\Pr[(c', s') \leftarrow WrOn(m, k, s), \\ (m', s'') \leftarrow UnwrOn(c, k', s) : s' = s''] = 1$$

(iv) WrOn and UnwrOn are polynomial-time computable and randomized algorithms.

Synchronicity. The second property is synchronicity. In order to achieve replay resistance, we have to require that once the wrapping and unwrapping do not have synchronized states anymore, the output of the wrapping and unwrapping algorithms is indistinguishable from randomness. For the following definition we use the modified challenger $\mathsf{OS\text{-}Ch}^{0'}$, which results from modifying $\mathsf{OS\text{-}Ch}^0$ such that along

(setup, ℓ')
do the same as $\mathsf{OS\text{-}Ch}^0$
additionally $k_S \leftarrow \{0,1\}^\eta$

$(\mathsf{compromise}, i)$
do the same as $\mathsf{OS\text{-}Ch}^0$

(send, m)
$q(st_f^1) \leftarrow m$
look up the first visible subpath $(cid_1, \langle k_i \rangle_{i=1}^j)$
if $j = \ell'$ **then** $m' \leftarrow q(st_f^1)$
else $k_{j+1} \leftarrow k_S$; $j \leftarrow j+1$; $m' \leftarrow 0^{|q(st_f^1)|}$
$((O_i)_{i=0}^j, s') \leftarrow WrOn^j(m, \langle k_i \rangle_{i=1}^j, st_f^1)$
update $st_f^1 \leftarrow s'$
store $onions(cid_j) \leftarrow O_1$; send O_j

$(\mathsf{unwrap}, O, cid_i)$
look up the forward v.s. $\langle k_i \rangle_{i=u}^j$ for cid_i
$O' \leftarrow onions(cid_i)$
$T \leftarrow \mathrm{M}(O, O')$; $q(st_f^i) \leftarrow T(q(st_f^i))$
if $j = \ell'$ **then** $m \leftarrow q(st_f^i)$
else $k_{j+1} \leftarrow k_S$; $j \leftarrow j+1$; $m \leftarrow 0^{|q(st_f^i)|}$
$((O_i)_{i=u-1}^j, s') \leftarrow WrOn^{j-u+1}(m, \langle k_i \rangle_{i=u}^j, st_f^i)$
update $st_f^i \leftarrow s'$
store $onions(cid_j) \leftarrow O_u$; send O_j

$(\mathsf{respond}, m)$
$q(st_b^{\ell'}) \leftarrow m$
look up the last visible subpath $\langle k_i \rangle_{i=u}^{\ell'}$
if $u = 1$ **then** $m \leftarrow q(st_b^{\ell'})$
else $k_{u-1} \leftarrow k_S$; $u \leftarrow u-1$; $m \leftarrow 0^{|q(st_b^{\ell'})|}$
$((O_i)_{i=u-1}^j, s') \leftarrow WrOn^{j-u+1}(m, \langle k_i \rangle_{i=u}^j, st_b^{\ell'})$
update $st_b^{\ell'} \leftarrow s'$
store $onions(cid_u) \leftarrow O_u$; send O_j

$(\mathsf{wrap}, O, cid_i)$
look up the backward v.s. $\langle k_i \rangle_{i=u}^j$ for cid_i
$O' \leftarrow onions(cid_i)$; $T \leftarrow \mathrm{M}(O, O')$
$q(st_b^i) \leftarrow T(q(st_b^i))$
get $\langle k_i \rangle_{i=u}^j$ for cid
if $u = 1$ **then** $m \leftarrow q(st_b^i)$
else $k_{u-1} \leftarrow k_S$; $u \leftarrow u-1$; $m \leftarrow 0^{|q(st_b^i)|}$
$((O_i)_{i=u-1}^j, s') \leftarrow WrOn^{j-u+1}(m, \langle k_i \rangle_{i=u}^j, st_b^i)$
update $st_b^i \leftarrow s'$
store $onions(cid_u) \leftarrow O_u$; send O_j

$(\mathsf{destruct}, O, cid)$
$m \leftarrow UnwrOn(, k_1, st_b^1)$
$O' \leftarrow onions(cid_1)$; $T \leftarrow \mathrm{M}(O, O')$
$q(st_b^1) \leftarrow T(q(st_b^1))$
if $m \neq \perp$ **then**
send $q(st_b^1)$

Figure 19: The Faking Onion Secrecy Challenger $\mathsf{OS\text{-}Ch}^1$: $\mathsf{OS\text{-}Ch}^1$ only answers for honest parties. st_f^i, st_b^i is the current forward, respectively backward, state of party i. $((O_i)_{i=u-1}^j, s') \leftarrow WrOn^{j-u+1}(m, \langle k_i \rangle_{i=u}^j, st)$ is defined as $O_{u-1} \leftarrow m$; **for** i = u **to** j **do** $(O_i, s') \leftarrow WrOn(O_{i-1}, k_{j+u-i}, st)$

with the output of the attacker also the state of the challenger is output. The resulting challenger $\mathsf{OS\text{-}Ch}^{0'}$ can, moreover, optionally get a state s as input.

Definition 13 (End-to-end integrity)**.** *Let $S(O, cid)$ be the machine that sends a $(\mathsf{destruct}, O)$ query to the challenger and outputs the response. Let $Q'(s)$ be the set of answers to $\mathsf{construct}$ queries from the challenger to the attacker. Let the last onion $O_{\ell'}$ of an onion O_1 be defined as follows:*

$\mathsf{Last}(O_1)$*:*
for $i = 1$ *to* $\ell' - 1$ ***do***
$O_{i+1} \leftarrow UnwrOn(O_i)$

*Let $Q(s) := \{\mathsf{Last}(O_1) \mid O_1 \in Q'(s)\}$ be the set of last onions answers to the challenger. We say a set of onion algorithms has **end-to-end integrity** if for all PPT attackers $\mathcal{A}$ the following is negligible in the security parameter η*

$$\Pr[(O, s) \leftarrow \mathcal{A}(1^\eta)^{\mathsf{OS\text{-}Ch}^{0'}}, (m, s') \leftarrow S(O, cid)^{\mathsf{OS\text{-}Ch}^{0'}(s)} : m \in M(\eta) \wedge P_{\ell'} \text{ is honest} \wedge O \notin Q(s')].$$

End-to-End Integrity. The third property that we require is **end-to-end integrity**; i.e. the attacker is not able to produce an onion that successfully unwraps unless it compromises the exit node. For the following definition, we modify $\mathsf{OS\text{-}Ch}^0$ such that, along with the output of the attacker, also the state of the challenger is output. In turn, the resulting challenger $\mathsf{OS\text{-}Ch}^{0'}$ can optionally get a state s as input. In particular, $(a, s) \leftarrow A^B$ denotes in the following definition the pair of the outputs of A and B.

Definition 14 (Synchronicity)**.** *For a machine $\mathcal{A}$, let $\Omega_{l,\mathcal{A}}$ and $\Omega_{r,\mathcal{A}}$ be defined as follows:*

Left: $\Omega_{l,\mathcal{A}}(\eta)$
$(m_1, m_2, st) \leftarrow \mathcal{A}(1^\eta)$
$k, s, s' \leftarrow \{0,1\}^\eta$
$O \leftarrow WrOn(m_1, k, s)$
$O' \leftarrow UnwrOn(O, k, s')$
$b \leftarrow \mathcal{A}(O', st)$

Right: $\Omega_{r,\mathcal{A}}(\eta)$
$(m_1, m_2, st) \leftarrow \mathcal{A}(1^\eta)$
$k, s, s' \leftarrow \{0,1\}^\eta$
$O \leftarrow WrOn(m_2, k, s)$
$O' \leftarrow UnwrOn(O, k, s')$
$b \leftarrow \mathcal{A}(O', st)$

For all PPT machines $\mathcal{A}$ the following is negligible in η:

$$|\Pr[b \leftarrow \Omega_{l,\mathcal{A}}(\eta) : b = 1] - \Pr[b \leftarrow \Omega_{r,\mathcal{A}}(\eta) : b = 1]|$$

Predictably Malleable Onion Secrecy. The fourth property that we require is **predictably malleable onion secrecy**, i.e. for every modification to a ciphertext the challenger is able to compute the resulting changes for the plaintext. This even has to hold for faked plaintexts. Note that this property is a stateful and weaker variant of what was introduced as Homomorphic-CCA-Security in [42].

In detail, we define a challenger $\mathsf{OS\text{-}Ch}^0$ that provides, a wrapping, a unwrapping and a send and a destruct oracle. In other words, the challenger provides the same oracles as in the onion routing protocol except that the challenger only provides one single session. We additionally define a faking challenger $\mathsf{OS\text{-}Ch}^1$ that provides the same oracles but fakes all onions for which the attacker does not control the final node.

For $\mathsf{OS\text{-}Ch}^1$, we define the maximal paths that the attacker knows from the circuit. A visible subpath of a circuit $(P_i, k_i, cid_i)_{i=1}^{\ell}$ from an honest onion proxy is a minimal subsequence of corrupted parties $(P_i)_{i=u}^{s}$ of $(P_i)_{i=1}^{\ell}$ such that P_{i-1} is honest and either $s = \ell$ or P_{s+1} is honest as well. The parties P_{i-1} and, if existent, P_{s+1} are called the guards of the visible subpath $(P_i)_{i=u}^{s}$. We store visible subpaths by the first $cid = cid_u$.

Figure 18 and 19 presents $\mathsf{OS\text{-}Ch}^0$, and $\mathsf{OS\text{-}Ch}^1$, respectively. [9]

Definition 15 (Predictably malleable onion secrecy)**.** *Let* onionAlg *be a pair of algorithms WrOn and UnwrOn. We say that the algorithms* onionAlg *satisfy* ***predictably malleable onion secrecy*** *if there is a negligible function μ such that there is a efficiently computable function* M *such that for all PPT machines $\mathcal{A}$ and sufficiently large η*

$$\Pr[b \leftarrow \{0,1\}, b' \leftarrow \mathcal{A}(1^\eta)^{\mathsf{OS\text{-}Ch}^b} : b = b'] \leq 1/2 + \mu(\eta)$$

Timed Standard Assumptions The assumptions above also require standard cryptographic assumptions such as CCA, CPA or the Decisional–Diffie–Hellman (DDH) assumption to hold when the adversary has access to timing information about e.g. how long it took to choose the exponents for the Diffie–Hellman key–exchange. We assume that these assumptions also hold in the timed setting.

Encryption Time. There is another important aspect we need to consider when handling timing information: the running time of encryption and decryption functions depend on the message to be encrypted and the key used to encrypt the message. While the different running times alone are reason enough to include this aspect into the delay-vectors, previous work [30] has shown that this information can leak information about the key and/or the message. Thus we need to accurately capture these small delays in our delay vectors.

We require the following: Let $f(x, m)$ denote the encryption (decryption) time needed to encrypt (decrypt) message m with the key x. Then the encryption (and decryption) times are indistinguishable with regards to the message, i.e. given following two events

$$\begin{aligned} M_1 : b = b^*;\ (m_0, m_1) \leftarrow \mathcal{A}, x \leftarrow KeyGen(1^\eta), \\ t \leftarrow f(x, m_b), b^* \leftarrow \mathcal{A}(t) \\ M_2 : b \neq b^*;\ (m_0, m_1) \leftarrow \mathcal{A}, x \leftarrow KeyGen(1^\eta), \\ t \leftarrow f(x, m_b), b^* \leftarrow \mathcal{A}(t) \end{aligned}$$

we have that

$$|Pr[M_1] - Pr[M_2]| < negl(\eta)$$

[9] We stress that in Figure 19 the onion O_u denotes the onion from party P_j to party P_{j+1}.

Note that this requirement is automatically fulfilled as soon as we assume the timed variant of the CPA assumption, as we could otherwise directly construct an adversary which breaks timed CPA from an adversary which distinguishes plain-texts from encryption times.

We give the above defined function f to the functionality in its delay-vector. The party P running $\mathcal{F}_{\text{OR}}$ gives f a key and a message, and f returns the number of steps P should idle in order to mimic the correct encryption/decryption time (this in particular also takes into account the number of steps required to compute f).[10]

Unfortunately, during the proofs presented below, we get the situation where the simulator uses a different key to do encryptions than were used in computing the delay. We therefore also have to make the assumption that the encryption (decryption) times are also indistinguishable with regards to the key, i.e. given the two events

$$K_1 : b = b^*; (x_0, x_1, m) \leftarrow \mathcal{A}, t \leftarrow f(x_b, m), b^* \leftarrow \mathcal{A}(t)$$
$$K_2 : b \neq b^*; (x_0, x_1, m) \leftarrow \mathcal{A}, t \leftarrow f(x_b, m), b^* \leftarrow \mathcal{A}(t)$$

we again have that

$$|Pr[K_1] - Pr[K_2]| < negl(\eta).$$

5.3.2 Soundness

The proof of soundness we present here is very close to the proof presented by Backes et al. [2] for the realization of $\mathcal{F}_{\text{OR}}$ by Π_{OR} in the standard UC-framework. But we have to make some alteration to take timing properties into account. The main challenge was to avoid time drifting too far apart in the scenario with $\mathcal{F}_{\text{OR}}$ compared to the scenario in which Π_{OR} is used.

Theorem 16. Π_{OR} *securely realizes* $\mathcal{F}_{\text{OR}}$ *in the* $\mathcal{F}_{\text{SCS}}, \mathcal{F}_{\text{REG}}^{\mathcal{N}}-$ *hybrid model for some delay vector* v.

Proof. We adopt the proof of secure UC-realization from [2]. That is, we define a sequence of games, for which we show that these are indistinguishable.

Game 1: This is the initial game in which Π_{OR} interacts with the adversary A_d and the environment Env. Here, being in the $\mathcal{F}_{\text{SCS}}, \mathcal{F}_{\text{REG}}^{\mathcal{N}}$ - hybrid model means that each party consists of a root node running the Π_{OR} code and two children nodes, each running the code for $\mathcal{F}_{\text{SCS}}$ and $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ respectively.

Game 2: In this game we replace the dummy adversary with a simulator $\mathcal{S}_1$. $\mathcal{S}_1$ consists of a root node, which is the main simulator simulating the dummy adversary, and two children nodes, each of which simulate the functionalities $\mathcal{F}_{\text{SCS}}$ and $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$. That is, we move all the children nodes from compromised parties in the network to the simulator and simulate them inside $\mathcal{S}_1$ (this also includes rewiring of all relevant ports, e.g. from the root node of a party to the $\mathcal{F}_{\text{SCS}}$- children node). Remember that $\mathcal{S}_1$ is timeless, while the simulated nodes are all time-ful. Thus we need to be careful with our simulation, making sure that messages going out of the functionalities, back to the parties in the network or the environment, are forwarded at the right time. In order to achieve this, $\mathcal{S}_1$ uses internal queues for all out-ports, marking each outgoing message with a time-stamp. These messages will then be sent out as soon as the timer of $\mathcal{S}_1$ has the correct value. Note that it is enough to internally simulate the $\mathcal{F}_{\text{SCS}}$ and $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ nodes of compromised nodes, as Env does not learn about uncompromised nodes and their behavior.

It is not obvious, that **Game 2** is indeed indistinguishable from **Game 1**. Env might try to push the network into the future while staying in the past itself (by just activating different machines whenever a scheduling request is received by Env). this in turn also forces $\mathcal{S}_1$ to stay in the past, disabling its ability to correctly simulate the impact of $\mathcal{F}_{\text{SCS}}$ and $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ nodes on the network. But staying in the past, Env would not be able to receive the messages sent by the functionalities, and answer them. As the environment would not be able to create a distinguishable situation in the network with staying in the past, he is forced to go forward. As he also has to put $\mathcal{S}_1$ forward at the same time, $\mathcal{S}_1$ will then use this opportunity to catch up. With this insight, its clear that **Game 2** and **Game 1** are indistinguishable.

Game 3: In this game, session keys are no longer generated by a key exchange protocol, but are just chosen randomly and saved in a common shared memory. In order to make sure that the timing

[10] We feel that this assumption is only necessary due to proof we present below. It would be interesting to improve the proof such that this assumption is no longer necessary.

remains correct, we first double the throughput of each party in order to accommodate the additional computation (randomly choosing the key and saving it in the shared memory) and introduce idle commands in the code of Π_{OR}, making sure that all messages are still sent out at the same times as in **Game 2**. Due to the security of the 1W-AKE, no ppt machine can distinguish the randomly chosen key from the key generated by 1W-AKE, hence this game is indistinguishable from **Game 2**.

Game 4: We adopt the visible subpath computations from [2]. While this only changes the messages, but not the amount of messages sent through the network, our main concern is the additional computation done by each party. In order to accommodate this, we again accelerate the party machines, introducing the new code for the visible-subpath computations and additional idle commands, making sure that messages are sent out at the same time as in **Game 3**. We make use of shared memory in order to enable parties to compute the visible sub paths: compromised parties indicate in the shared memory that they are compromised, and parties doing the visible subpath computation get all necessary information from the shared memory. This work around is necessary as in the original model [2], there is only a single protocol machine P which internally simulates all participating parties, and does the visible subpath computations. This approach is not feasible in our model, as this would require P to live in several points in time simultaneously.

Due to onion secrecy and synchronicity of the used onion encryption algorithms, and as we make sure that the time stamps remain the same, no ppt adversary can distinguish between **Game 3** and **Game 4**.

Game 5: In this games, each party internally simulates $\mathcal{F}_{\text{OR}}$ for doing the visible-subpath computations. That is, every input from the environment is directly forwarded to $\mathcal{F}_{\text{OR}}$, which in turn returns the computed visible subpaths and messages to be sent through the network.

A major difference to Π_{OR} here is in the key registration. Upon input $(\mathsf{Register}, P)$, $\mathcal{S}_1$ internally simulates the interaction with the key registration functionality and makes sure that all required network messages are sent to Env.

Other small differences are discussed in [2] and will be skipped here. Our main concern will be making sure that the time stamps of each message remain correct, and also that time variables of each party progress at the same rate as in **Game 4**. While compared to **Game 4**, we save code for the parties (from outsourcing the visible subpath computation), we still have to accommodate the simulation of $\mathcal{F}_{\text{OR}}$ in our time budget. Again we accelerate our parties and introduce idle commands as required in order to make sure that messages are sent into the network at the right time.

As we make sure that the timestamps of all messages remain the same, the indistinguishability of **Game 4** and **Game 5** follows from the anonymity of the $1W - AKE$ protocol as discussed in [2].

Game 6: Here, we replace the protocol code by the functionality $\mathcal{F}_{\text{OR}}$. In **Game 5**, Π_{OR} directly forwarded all inputs from Env to $\mathcal{F}_{\text{OR}}$, hence the messages sent by $\mathcal{F}_{\text{OR}}$ remain the same. As these are sent to the adversary, $\mathcal{S}_1$ will receive them and can then compute the correct network messages by internally simulating Π_{OR}. This will now be our final simulator $\mathcal{S}$.

At this points, $\mathcal{F}_{\text{OR}}$ will require much less time than Π_{OR} in **Game 5**. In order to close this gap, we correctly set all the delay values in $\mathcal{F}_{\text{OR}}$'s delay vector, by adding up all the idle commands added in the previous games, taking account of the accelerations and including the encoding-time-distribution functions whenever encryption would happen in the real world scenario.

As the timestamps therefore remain the same as in **Game 6**, and as $\mathcal{S}$ correctly computes all network messages as in **Game 5**, **Game 5** and **Game 6** are indistinguishable.

□

5.3.3 Completeness

We show that $\mathcal{F}_{\text{OR}}$ is a complete abstraction of Π_{OR}, i.e. $\mathcal{F}_{\text{OR}}$ securely realizes Π_{OR} in TUC. The proof is in the same spirit as for the soundness, just in the other direction: we add one cryptographic detail after the other until we finally get the Π_{OR} protocol. Special care is required in the construction of the simulator, which now also has to filter scheduling requests coming from the network.

Theorem 17. *$\mathcal{F}_{\text{OR}}$ with some delay vector v securely realizes Π_{OR} in the $\mathcal{F}_{\text{SCS}}, \mathcal{F}_{\text{REG}}^{\mathcal{N}}-$ hybrid model.*

Proof. We again construct a sequence of games which are pairwise indistinguishable.

Game 1: Parties running the ideal functionality interacting with $\mathcal{A}_d$ and Env. We assume a delay vector containing just the encoding time distribution functions mentioned earlier.

Game 2: We modify the ideal functionality $\mathcal{F}_{\text{OR}}$ to $\mathcal{F}_{\text{OR}2}$, in which the $SendMessage$ method is modified to no longer compute visible sub paths. Instead, a compromised party sends the message $\langle P, P_{next}, cid_{next}, h\rangle$ to the adversary. We replace the dummy adversary $\mathcal{A}_d$ by the simulator $\mathcal{S}_2$, which computes the visible subpaths and communicates these informations to Env. For this, $\mathcal{S}_2$ remembers established *cid*'s and circuit information between compromised parties and keeps track of visible subpaths. Now, whenever $\mathcal{S}_2$ receives above message, it evaluates known *cid* information, and leaks the required information to Env.

To make sure that the parties leaking information to $\mathcal{S}_2$ still proceed in time as fast as in **Game 1**, we add idle commands right before the information leak happens. This makes sure that the leak to $\mathcal{S}_2$ happens at the same time as it would have happened in **Game 1**, and $\mathcal{S}_2$ does not need to concern itself with correctly timing the messages to Env, as it is timeless.

As now all messages received by Env in **Game 2** are the same as in **Game 1**, they are both indistinguishable.

Game 3: We modify $\mathcal{F}_{\text{OR}3}$ to no longer use handles to propagate messages, but use the onion encryption algorithms to encrypt the messages. Encryption keys are drawn randomly and stored in the shared memory. Onions are created using secure, stateful onion algorithms.

We drop the encryption time delays from the delay vector. They are no longer required, as encryptions and decryptions are directly executed in $\mathcal{F}_{\text{OR}3}$.

Whenever $\mathcal{S}_3$ receives onions from compromised parties in the network, it randomly draws a handle, which it then uses to simulate the information leak done by the original $\mathcal{F}_{\text{OR}}$ functionality.

In case $\mathcal{S}_3$ receives an onion from the first node P in the final visible subpath $\langle P, \ldots, P_{last}\rangle$ of a circuit (i.e. the subpath containing the exit node), and the exit node is compromised, $\mathcal{S}_3$ decrypts the onion using the session keys (that it learned from the compromised parties) used in the circuit. AT the same time $\mathcal{S}_3$ make sure to correctly update the states of the onion algorithms it learns internally. The resulting message is then forwarded to the adversary. Note that we do not have to care about the states of the onion encryption and decryption algorithms of the nodes following P: if the adversary chooses to not forward an intercepted message and therefore the states of the onion algorithms of the parties in the network get asynchronous, this is just a result of the synchronicity property of secure onion algorithms.

In order to accommodate for the onion algorithm computation in each party, we again accelerate the parties and add idle commands in their code, making sure that the correctly computed onion are leaked to $\mathcal{S}_3$ at the same time as the handle information would have been leaked to $\mathcal{S}_2$ in **Game 2**.

Thus the time stamps of the messages sent through the network remain the same. As $\mathcal{S}_3$ makes sure that Env only receives messages which it would have received in **Game 2**, both games are indistinguishable.

Game 4: In $\mathcal{F}_{\text{OR}4}$, symmetric encryption keys are no longer drawn at random, but generated using a $1W - AKE$. Due to the security properties of $1W - AKE$-schemes, the generated keys are indistinguishable from randomly chosen ones.

In order to accommodate for the additional delay caused by the key agreement protocol, we accelerate each machine and add idle commands, so that at the end of the circuit establishment, both in **Game 3** and **Game 4**, every party is in the same point of time.

As the key exchange is handled inside the circuit establishment messages, $\mathcal{S}_4$ does not need to drop or create any messages, but just modify them to look like the circuit establishment messages from **Game 4**. This is done by simply dropping the key-establishment part of each message.

Thus, as the messages received by Env in **Game 4** are the same as in **Game 5**, and these messages all have the same timestamps, **Game 4** and **Game 5** are both indistinguishable.

Game 5: In $\mathcal{F}_{\text{OR}5}$ we replace the key registration phase in $\mathcal{F}_{\text{OR}}$ with the one from Π_{OR} and the code for sending messages over the secure channel functionality $\mathcal{F}_{\text{SCS}}$. This includes introducing two children scheduler nodes in $\mathcal{S}_5$ which internally simulate $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ and $\mathcal{F}_{\text{SCS}}$ nodes.

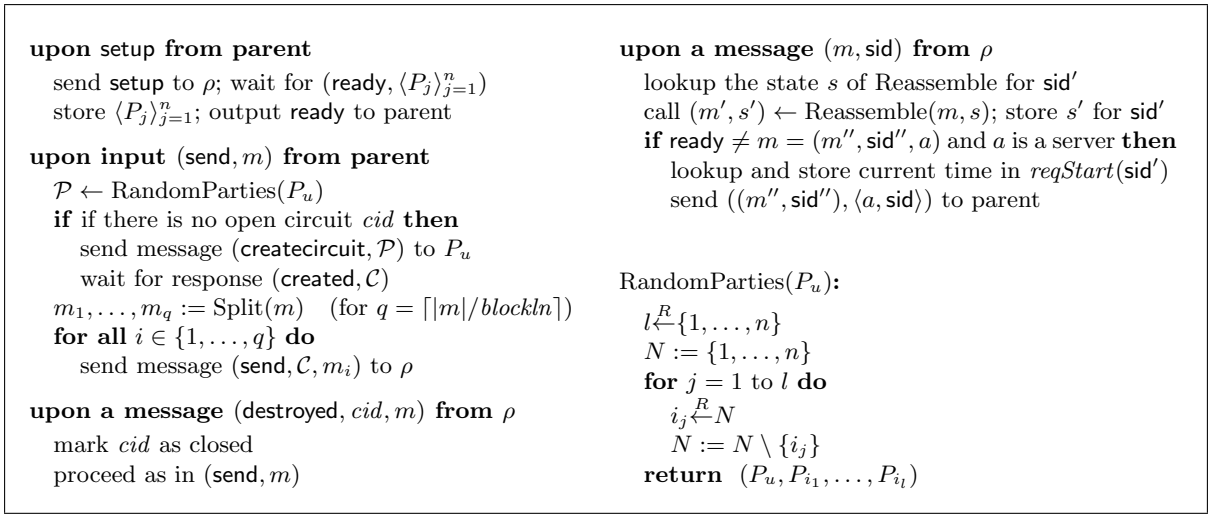

Figure 20: Wrapper $\Pi_{\text{WOR}}\rho$ for client P_u

Game 6: We pull the $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ and $\mathcal{F}_{\text{SCS}}$ parties out of $\mathcal{S}_5$ (resulting in $\mathcal{S}_6$) and put them into the network, into the protocol trees of the parties in $\mathcal{F}_{\text{OR}5}$. We thus obtain Π_{OR} the protocol executed in a $\mathcal{F}_{\text{SCS}}, \mathcal{F}_{\text{REG}}^{\mathcal{N}}$-hybrid environment.

As we already simulated all messages correctly in above game, the number of messages and messages themselves remain the same. We do have to care about the timestamps of the messages, as these parties are now timeful and we get additional delay as more parties are involved in transmitting a message. We compensate for this by suitably accelerating the machines. This is possible, as both ideal functionalities behave deterministically and create a predictable amount of overhead, depending on by which message they are activated.

Strictly speaking, **Game 6** does not exactly contain the code provided by Π_{OR}, but is augmented by additional idle commands. To get rid of these, we can count back all idle commands added in the intermediate games and add these (sufficiently stretched to account for the accelerations) to the delay-vector of the initial ideal functionality $\mathcal{F}_{\text{OR}}$. This is indeed possible, as we only have rational time in our model, and can therefore get integer number of steps for the delay vector by suitably accelerating (or decelerating) the initial game. □

5.4 A User Interface: the Wrapper Π_{wor}.

We allow for the sake of modularity to let the environment, i.e., the parent node, to command the circuit and to choose the path. In many cases, however, this additional complexity becomes inconvenient. In this section, we present a wrapper Π_{WOR} (see Figure 20) that performs the circuit construction, and splits messages and re-assembles the messages blocks. We present a wrapper that uses a uniform path selection; however, by adjusting the distribution RandomParties, any path selection can be used.

We stress that such a $\mathcal{F}_{\text{OR}}$ together with such a wrapper Π_{WOR} (i.e., $\Pi_{\text{WOR}\mathcal{F}_{\text{OR}}}$) give rise to an anonymous channel functionality, as to the suggested by Canetti [9]. However, such an anonymous channel functioanlity would have as much leakage as and give the attacker as much influence capabilities as $\mathcal{F}_{\text{OR}}$; thus, we refrained from presenting it in this work.

Re-assembling and splitting in Π_{wor}. We assume a stateful routine Reassemble(m, s) which expects as input a message block m (and a state s) and outputs together with a new state s' either a dummy message ready, if a complete message could not be reassembled yet, or a re-assembled message $m' \neq \mathsf{ready}$, if m and the state s allowed re-assembling a complete message. In the description of the protocol, we lookup the state of Reassemble and store it in the variable s. If this lookup fails, we assign to the variable s the empty state, i.e., the empty string. Dual to the re-assembling routing, we assume a splitting routine $(m_1, \ldots, m_{\lceil |m|/blockln \rceil}) \leftarrow \text{Split}(m)$ which splits the message into message blocks m_i of length *blockln* and pads the last block if necessary.

Both: upon (setup) **from the parent**
send setup to ρ; wait for ready; send ready to parent

Client: upon $(\mathsf{req}, address)$ **from the parent**
send $(\mathsf{send}, address)$ to ρ

Client: upon (m, cid) **from** ρ
output (m, cid) to the parent

Exit node: upon $(\mathsf{received}, (m, \mathsf{sid}'))$ **from** ρ
if $m = (m'', \mathsf{sid}'', a) \wedge bun_{\mathsf{sid}''} = \bot \wedge a$ is a server **then**
lookup and store current time in $reqStart(\mathsf{sid}')$
store (m, sid') in $bun_{\mathsf{sid}''}$
while $\exists$ $request(loc, a)$ in $bun_{\mathsf{sid}''}$ **do**
remove $request(loc, a)$ from $bun_{\mathsf{sid}''}$
send $(loc, \langle a, \mathsf{sid}\rangle)$ over the network
store the response res in $bun_{\mathsf{sid}''}$
lookup the smallest $n' \geq |bun_{\mathsf{sid}''}|$ in psBuckets
pad $bun_{\mathsf{sid}''}$ to a size of n' and store it in m'
send (send, m') to ρ at time $t_{\text{buf}} + reqStart(\mathsf{sid}')$

Figure 21: The protocol WFC_ρ for party pid, where sid is its session ID

upon $(\mathsf{visit}, (a_i)_{i=1}^k)$ **from the parent**
send setup to ρ; wait for ready
if all $a_i \in \mathsf{pageBuck}(k)$ **then**
draw one a_j from $(a_i)_{i=1}^k$ at random
store $g := (a_j, k)$; send (req, a_j) to ρ

upon (guess, a) **from the parent**
if $g = (a, k)$ **then** output $(\mathsf{guess}, k, 1)$
else $(\mathsf{guess}, k, 0)$

Figure 22: k-anonymity challenger: KCh_ρ

6 Countermeasure against Website Fingerprinting

We leverage our time-sensitive framework and our Tor abstraction, to prove a countermeasure against website fingerprinting secure. The countermeasure achieves k-anonymity for web pages without dynamic requests, such as Ajax. The countermeasure protocol, called WFC, is plugged on top of the Tor protocol such that the exit nodes perform all the web page requests and respond the entire web page at once and additionally wait until a time buffer t_{buf} has passed. In order to improve performance, the countermeasure uses buckets for common web page sizes and pads the web pages up to the next larger bucket. We assume a list psBuckets of buckets for web page sizes, i.e., a list of bucket sizes. Solely for the analysis, we further assume the dual data structure pageBuck that upon a bucket size k outputs the list of names of webpages that would fall into that bucket. Our countermeasure protocol WFC is depicted in Figure 21.

Webpages. For our purposes it suffices to represent webpages as lists of elements and requests. Elements are arbitrary bitstrings m that are marked as elements; we denote them as $element(m)$. Requests are consist of a pair of party ID a and a location loc on that party's webserver (an arbitrary bitstring); we denote them as $request(loc, a)$. We stress that we prove our results for web pages that do not dynamically load content, upon user inputs, e.g., by using JavaScript techniques such as Ajax.

Web servers. Our result is parametric in the list L of webpages that are served by the web servers. We require that for every webpage (w in L) there is one server. A server might offer several webpages. We characterize a web server by the list $L' \subseteq L$ of webpages that it offers. The web server protocol $S_{L'}$ works as follows: upon a request r, the web server looks up whether r is the name of a webpage $w \in L'$, and if so $S_{L'}$ responds with w.

For the definition of k-anonymity, we use a data structure $\mathsf{pageBuck}_L$, which is dual to the list psBuckets of bucket sizes for webpages. Upon input k (the bucket size), $\mathsf{pageBuck}_L$ outputs the list of names that would fall into that bucket. $\mathsf{pageBuck}_L$ depends on L, but for the sake of convenience we omit the L and merely write pageBuck.

k-anonymity challenger. We use our framework to formulate the k-anonymity game. We let the adversary consist of the environment and the network attacker and define around the WFC protocol the k-anonymity challenger (see Figure 22), which defines the requests that the adversary can query.

Clients, entry nodes, exit nodes, and entry links. For the formulation of our result, we assume a partitioning the Tor network into **clients nodes**, called onion proxies, **entry nodes**, i.e. nodes that are connected to a client node, and **exit nodes**, i.e. nodes that communicate with web servers outside of the Tor network. With such a partitioning an **entry link** is an edge between a client node and an entry node. For the sake of convenience, we use the wrapper protocol Π_{WOR} (see Figure 20), which construct circuits and splits message to message blocks before it sends these message blocks to Π_{OR} (or $\mathcal{F}_{\text{OR}}$, respectively).

The k-anonymity follows from the composition theorem (Theorem 10), the realization theorem (Theorem 16), the definition of $\mathcal{F}_{\text{OR}}$, and from the bucketing property and the waiting procedure.

Lemma 18 (WFC provides k-anonymity for the recipient)**.** *Let* EXEC' *be defined as* EXEC *except that* EXEC' *outputs the bit* b *from the first output of the form* (guess, k, b) *by* KCh *for* Env*.*

For any ppt environment Env *that and any ppt attacker* $\mathcal{A}$ *that at most compromises one client's entry node or the link to that entry node and for sufficiently large* t_{buf} *and* η *and a negligible function* μ

$$\Pr[\mathsf{EXEC}'_\eta(\langle \text{KCh}_{\text{WFC}\Pi_{\text{WOR}}\Pi_{\text{OR}}}, \text{S}\rangle, \mathcal{A}, \text{Env}) = 1] \leq 1/k + \mu(\eta)$$

Proof. By the realization theorem (Theorem 16) and the composability theorem (Theorem 10), $\text{KCh}_{\text{WFC}\Pi_{\text{WOR}}\Pi_{\text{OR}}}$ securely realizes $\text{KCh}_{\text{WFC}\Pi_{\text{WOR}}\mathcal{F}_{\text{OR}}}$; hence a network attacker against $\text{KCh}_{\text{WFC}\Pi_{\text{WOR}}\mathcal{F}_{\text{OR}}}$ (and hence also against $\text{KCh}_{\text{WFC}\Pi_{\text{WOR}}\Pi_{\text{OR}}}$) that only compromises the entry node of one client or the link to the entry node of one client only learns the *cid*s and freshly drawn handles, which are both independent of the recipient. Hence, the pattern of the response is the same for all names that the environment input.

Since we assumed that the web servers wait until a fixed time t before they response, the response time of the web servers is always the same. We conclude that the adversary cannot learn more than the length of the requested web page. Since KCh has output $(\mathsf{guess}, k, 1)$, the length of all possibly requested web pages is padded to $\mathsf{pageBuck}(k)$. Hence, the length and the time pattern of all web pages is the same. □

7 Conclusion and Future Work

In this work, we presented TUC, a universal composability framework that includes a comprehensive notion of time, which is suitable for and tailored to the demands of analyzing AC protocols. Our framework provides all properties that are expected of a universally composability framework: a universal composability result, a joint state theorem, and the completeness of the dummy adversary. As a case study, we showed that a abstraction of the onion routing protocol provided by Backes et al. [2] can be suitably extended to account for timing and that it is realized in TUC by a similarly extended onion routing protocol. We then leveraged this abstraction and our framework to formulate a countermeasure against website fingerprinting attacks and proved this countermeasure secure.

An interesting direction for future work is the evaluation of more elaborate countermeasures against known time-sensitive attacks. Since our framework comprehensively models timing attacks, every verification of our abstraction, or a possible extension thereof after implementing countermeasures, also yields security guarantees for the actual OR protocol.

We would also like to see an extension of the basic framework, e.g. by means of suitable ideal functionalities, which incorporates latency and bandwidth limits on communication links between parties. Such an extension could for example be used for the analysis of denial-of-service resistance mechanisms.

There is a line of work on automated verification techniques for timed automata. It would be interesting to show that in certain cases timed automata are a sound abstraction for protocols that are formulated in TUC. Such a result would allow to obtain strong guarantees, i.e., against computational attackers that can perform time-measurements, from established automated verification tools [12, 28, 6, 34, 31, 32].

Acknowledgements. We would like to thank Dominique Unruh for initial discussions and on a simulation-based composability framework with time-sensitive attackers and valuable feedback on the preliminary version of the work.

References

[1] Michael Backes, Goran Doychev, and Boris Köpf. Preventing Side-Channel Leaks in Web Traffic: A Formal Approach. In *Proceedings of the 20th Network and Distributed Systems Security Symposium (NDSS)*, 2013.

[2] Michael Backes, Ian Goldberg, Aniket Kate, and Esfandiar Mohammadi. Provably Secure and Practical Onion Routing. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium (CSF)*, pages 369–385, 2012.

[3] Michael Backes, Aniket Kate, Praveen Manoharan, Sebastian Meiser, and Esfandiar Mohammadi. AnoA: A Framework for Analyzing Anonymous Communication Protocols. In *Proceedings of the 26th IEEE Computer Security Foundations Symposium (CSF)*, pages 163–178, 2013.

[4] Michael Backes, Aniket Kate, and Esfandiar Mohammadi. Ace: An Efficient Key-Exchange Protocol for Onion Routing. In *Proceedings of the 11th ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 55–64, 2012.

[5] Michael Backes, Birgit Pfitzmann, and Michael Waidner. The Reactive Simulatability (RSIM) Framework for Asynchronous Systems. *Information and Computation*, 205:1685–1720, 2007.

[6] Liana Bozga, Cristian Ene, and Yassine Lakhnech. A Symbolic Decision Procedure for Cryptographic Protocols with Time Stamps. *The Journal of Logic and Algebraic Programming*, 65:1–35, 2005.

[7] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching From a Distance: Website Fingerprinting Attacks and Defenses. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 605–616, 2012.

[8] Jan Camenisch and Anna Lysyanskaya. A Formal Treatment of Onion Routing. In *Advances in Cryptology - CRYPTO 2005*, pages 169–187, 2005.

[9] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, 2001.

[10] Ran Canetti and Tal Rabin. Universal Composition with Joint State. In *Advances in Cryptology - CRYPTO 2003*, pages 265–281, 2003.

[11] Sambuddho Chakravarty, Angelos Stavrou, and Angelos D. Keromytis. Traffic Analysis against Low-Latency Anonymity Networks Using Available Bandwidth Estimation. In *Proceedings of the 15th European Symposium on Research in Computer Security (ESORICS)*, pages 249–267, 2010.

[12] Ricardo Corin, Sandro Etalle, Pieter H. Hartel, and Angelika Mader. Timed Model Checking of Security Protocols. In *Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering*, pages 23–32, 2004.

[13] Roger Dingledine. Tor and Circumvention: Lessons Learned. In *Advances in Cryptology - CRYPTO 2011*, pages 485–486, 2011.

[14] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium (USENIX)*, pages 303–320, 2004.

[15] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S & P)*, pages 332–346, 2012.

[16] Nathan S. Evans, Roger Dingledine, and Christian Grothoff. A Practical Congestion Attack on Tor Using Long Paths. In *Proceedings of the 18th USENIX Security Symposium (USENIX)*, pages 33–50, 2009.

[17] Joan Feigenbaum, Aaron Johnson, and Paul F. Syverson. A Model of Onion Routing with Provable Anonymity. In *Proceedings of the 11th International Conference on Financial Cryptography and Data Security (FC)*, pages 57–71, 2007.

[18] Joan Feigenbaum, Aaron Johnson, and Paul F. Syverson. Probabilistic Analysis of Onion Routing in a Black-Box Model. *ACM Transactions on Information and Systems Security*, 15(3):14, 2012.

[19] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1, 1999. RFC Editor.

[20] Yossi Gilad and Amir Herzberg. Spying in the Dark: TCP and Tor Traffic Analysis. In *Proceedings of the 12th Privacy Enhancing Technologies Symposiun (PETS)*, pages 100–119, 2012.

[21] Ian Goldberg, Douglas Stebila, and Berkant Ustaoglu. Anonymity and One-Way Authentication in Key Exchange Protocols. *Designs, Codes and Cryptography*, 67(2):245–269, 2013.

[22] Oded Goldreich, Silvio M. Micali, and Avi Wigderson. How to Play ANY Mental Game. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229, 1987.

[23] David M. Goldschlag, Michael G. Reed, and Paul F. Syverson. Onion routing. *Communications of the ACM*, 42(2):39–41, 1999.

[24] Xun Gong, Negar Kiyavash, and Nikita Borisov. Fingerprinting Websites using Remote Traffic Analysis. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 684–686. ACM, 2010.

[25] Dennis Hofheinz and Victor Shoup. GNUC: A New Universal Composability Framework. Journal of Cryptology, to appear, 2014.

[26] Dennis Hofheinz, Dominique Unruh, and Jrn Mller-Quade. Polynomial runtime and composability. *Journal of Cryptology*, 26(3):375–441, 2013.

[27] Nicholas Hopper, Eugene Y. Vasserman, and Eric Chan-Tin. How much anonymity does network latency leak? *ACM Transactions on Information and Systems Security*, 13(2):13, 2010.

[28] Gizela Jakubowska and Wojciech Penczek. Is Your Security Protocol on Time? In *Proceedings of the 2007 International Symposium on Fundamentals of Software Engineering (FSEN)*, pages 65–80, 2007.

[29] Weijia Jia, Fung Po Tso, Zhen Ling, Xinwen Fu, Dong Xuan, and Wei Yu. Blind Detection of Spread Spectrum Flow Watermarks. *Security and Communication Networks*, pages 257–274, 2013.

[30] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 104–113, 1996.

[31] Miroslaw Kurkowski and Wojciech Penczek. Timed Automata Based Model Checking of Timed Security Protocols. *Fundamenta Informaticae - Concurrency Specification and Programming*, 93(1-3):245–259, 2009.

[32] Miroslaw Kurkowski, Wojciech Penczek, and Andrzej Zbrzezny. SAT-Based Verification of Security Protocols Via Translation to Networks of Automata. In *Model Checking and Artificial Intelligence*, volume 4428, pages 146–165. 2007.

[33] Ralf Küsters. Simulation-Based Security with Inexhaustible Interactive Turing Machines. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 309–320, 2006.

[34] Ruggero Lanotte, Andrea Maggiolo-Schettini, and Angelo Troina. Time and Probability-Based Information Flow Analysis. *IEEE Transactions on Software Engineering*, 36(5):719–734, 2010.

[35] Zhen Ling, Junzhou Luo, Yang Zhang, Ming Yang, Xinwen Fu, and Wei Yu. A Novel Network Delay based Side-Channel Attack: Modeling and Defense. In *Proceedings of the 31st Annual IEEE International Conference on Computer Communications (INFOCOM)*, pages 2390–2398, 2012.

[36] Xiapu Luo, Peng Zhou, Edmond W. W. Chan, Wenke Lee, Rocky K. C. Chang, and Roberto Perdisci. HTTPOS: Sealing Information Leaks with Browser-side Obfuscation of Encrypted Flows. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS)*, 2011.

[37] Steven J. Murdoch and Piotr Zieliński. Sampled Traffic Analysis by Internet-Exchange-Level Adversaries. In *Proceedings of the 7th Workshop on Privacy Enhancing Technologies (PET)*, pages 167–183, 2007.

[38] Gavin O'Gorman and Stephen Blott. Improving Stream Correlation Attacks on Anonymous Networks. In *Proceedings of the 24th ACM Symposium on Applied Computing*, pages 2024–2028, 2009.

[39] Lasse Øverlier and Paul F. Syverson. Locating Hidden Servers. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S & P)*, pages 100–114, 2006.

[40] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website Fingerprinting in Onion Routing based Anonymization Networks. In *Proceedings of the 10th ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 103–114, 2011.

[41] Birgit Pfitzmann and Michael Waidner. Composition and Integrity Preservation of Secure Reactive Systems. In *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS)*, pages 245–254, 2000.

[42] Manoj Prabhakaran and Mike Rosulek. Homomorphic Encryption with CCA Security. In *Automata, Languages and Programming*, volume 5126, pages 667–678. 2008.

[43] Paul F. Syverson, Gene Tsudik, Michael G. Reed, and Carl E. Landwehr. Towards an Analysis of Onion Routing Security. In *Designing Privacy Enhancing Technologies - International Workshop on Design Issues in Anonymity and Unobservability*, pages 96–114, 2000.

[44] Tor Metrics Portal. `https://metrics.torproject.org/`. Accessed May 2013.

[45] The Tor Project. `https://www.torproject.org/`. Accessed May 2013.

[46] Dominique Unruh. *Protokollkomposition und Komplexität*. PhD thesis, Universität Karlsruhe (TH), 2007. In German.

[47] Xinyuan Wang, Douglas S. Reeves, and Shyhtsun Felix Wu. Inter-Packet Delay Based Correlation for Tracing Encrypted Connections through Stepping Stones. In *Proceedings of the 17th European Symposium on Research in Computer Security (ESORICS)*, pages 244–263, 2002.

[48] Charles V. Wright, Lucas Ballard, Scott E. Coull, Fabian Monrose, and Gerald M. Masson. Spot Me if You Can: Uncovering Spoken Phrases in Encrypted VoIP Conversations. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (S & P)*, pages 35–49, 2008.

[49] Charles V. Wright, Scott E. Coull, and Fabian Monrose. Traffic Morphing: An Efficient Defense Against Statistical Traffic Analysis. In *Proceedings of the 16th Network and Distributed Security Symposium (NDSS)*, pages 237–250, 2009.