TUC: Time-sensitive and Modular Analysis of Anonymous Communication

Michael Backes   Praveen Manoharan   Esfandiar Mohammadi

CISPA, Saarland University, Germany

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

February 12, 2014

Abstract

The anonymous communication protocol Tor constitutes the most widely deployed technology for providing anonymity for user communication over the Internet. Several frameworks have been proposed that show strong anonymity guarantees for such protocols; none of these frameworks, however, are capable of modeling the class of traffic-related timing attacks against Tor, such as traffic correlation and website fingerprinting.

In this work, we present TUC: the first framework that allows for rigorously proving strong anonymity guarantees in the presence of time-sensitive adversaries that mount traffic-related timing attacks. TUC incorporates a comprehensive notion of time in an asynchronous communication model with sequential activation, while offering strong compositionality properties for security proofs. We apply TUC to evaluate a novel countermeasure for Tor against website fingerprinting attacks. Our analysis relies on a formalization of the onion routing protocol that underlies Tor and proves rigorous anonymity guarantees in the presence of traffic-related timing attacks.

Contents

1 Introduction

Anonymous communication protocols, as provided by the Tor network [53], are an increasingly popular way for users to improve their privacy by hiding their location, i.e., their IP address. The Tor network is currently used by hundreds of thousands of users around the world [52].

In order to precisely understand the anonymity guarantees provided by Tor, several rigorous analyses have been conducted [2, 9, 21, 51, 3, 22], which show strong anonymity guarantees for the onion routing protocol used by Tor; however, all of these analyses abstract from network-level timing attacks, such as traffic correlation or website fingerprinting, which arguably form the most important class of attacks against Tor's anonymity guarantees [14, 19, 43, 31, 23, 30, 41, 44, 46, 47, 55, 32]. One of the main obstacles in including such time-sensitive attacks into a rigorous analysis is the lack of a communication model that enables a composable security 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. [25] and put forward by [6, 12, 16, 28, 39, 42], which enable a composable security analysis of complex cryptographic protocols.

Contribution. In this work, we present TUC: the first framework that allows for rigorously proving strong anonymity guarantees in the presence of time-sensitive adversaries that mount traffic-related timing attacks. TUC incorporates a comprehensive notion of time in an asynchronous communication model with sequential activation, while offering strong compositionality properties for security proofs. In particular, TUC is based on a modified version of GNUC [28], which is one of the recent pieces of work [12, 39] that address many of the problems faced by earlier designs for simulation-based security frameworks [11].

We discuss the modifications to the communication model of GNUC in order to adequately account for time, and we show solutions for problems that occur when handling time-sensitive interaction between different parties over the network. In particular, we discuss that previous frameworks inherently are not suited for modeling time-sensitive asynchronous communication because they allow unrestricted activation orders: it might, e.g., happen that a message that was sent in the past (over a direct connection) arrives after a time-out mechanism already closed a port, only because the sending party was not activated early enough. We propose a remedy by only allowing consistency enforcing activation orders, which enforce that all parties receive all messages at the correct time. It turns out that all consistency enforcing activation orders are equivalent. As a result, we fix the activation order and thereby, in contrast to previous work, neither the environment nor the adversary has to learn any unrealistic information about activation requests. We show that valued properties, such as the joint state theorem and universal composability, hold in our time-sensitive framework as well.

Finally, we apply TUC to the onion routing protocol that underlies Tor, and we show how traffic-related timing attacks, such as inter-packet delay, traffic watermarking, and website fingerprinting attacks, can be mounted by an adversary in TUC. Next, we propose a countermeasure against website fingerprinting attacks and utilize TUC to prove k-recipient anonymity guarantees for this countermeasure.

Outline. Section 2 discusses related work. Section 3 introduces the time-sensitive TUC framework, and presents the activation order independence of TUC. Section 4 then introduces the notion of secure realization into this time sensitive communication model and shows that classic results of composable security are preserved in the time sensitive setting. In Section 6, we discuss how known traffic-related timing attacks on Tor can be represented in TUC. Moreover, we provide a countermeasure against website fingerprinting attacks and prove it secure in TUC.

2 Related Work

Tor [53] is one of the most widely used anonymous communication protocols to date [52] and is based on the (first generation) onion routing protocol by Goldschlag et al. [26]. There has been significant work in analyzing the anonymity guarantees provided by Tor [9, 21, 51, 22, 20, 4]. The major shortcoming of previous work is that it does not consider timing features of network traffic, which are used in timing-based traffic analyses. Considering the amount of proposed attacks [46, 23, 14, 55, 44, 8, 48, 27, 18] in the literature that use these timing features, it is clear that a rigorous framework that encompasses time-sensitive adversaries is required.

Some protocols, such as the onion routing protocol, are inherently insecure against global adversaries, but provide guarantees against partially global adversaries, which might only control servers or the user's links (like ISPs), and are useful in practice. Such systems cannot always be properly analyzed in time-insensitive frameworks [6, 12, 16, 28, 39, 42] because in these frameworks partially global adversaries are too weak: they cannot measure time-sensitive features, such as measure inter-packet delay or throughput per time interval, and they thus can also not measure effects of some active attacks, such as traffic watermarking or slowing down certain parties by mounting denial-of-service attacks. Since TUC enables the adversary to measure time-sensitive features, this family of attacks can be mounted by an adversary in TUC; thus, TUC is better suited for analyzing such weaker adversary scenarios.

This work contributes to the successful line of work on simulation-based universal composability frameworks [6, 12, 16, 28, 39, 42]. These frameworks allow for a composable 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. We chose to base our TUC framework on the GNUC framework by Hofheinz and Shoup [28]. While GNUC is not as general as other frameworks due to its strict poly-time notion and its tree-like structure of party-structure, it has the advantage that a composed ideal poly-time protocol implies a composed real poly-time protocol due to its strict polynomial-time notion [28, Section 11.8], and simplifies the proof of the composition theorem and thereby also our extension due to its simple party-structure. We are, however, confident that the main mechanisms for introducing our comprehensive notion of time, including time-sensitive adversaries, can also be applied to other frameworks, such as the RSIM[6], IITM [39], and UC [12] framework.

Previous work on synchronous communication granting protocol parties the capability to measure time or to proceed round-wise in order to enable proofs about properties, such as guaranteed termination or input termination [34, 1, 16, 10, 35, 45, 49]. Such approaches, however, do not grant the adversary the capability to measure the time at which a message arrives.

Modeling timing attacks in synchronous frameworks might be possible, assuming very fast rounds and thus highly synchronized clocks, (in the order of milliseconds), but such an approach has two severe technical limitations: first, highly synchronized clocks can seldom be assumed in practice, in particular not for commodity hardware; second, such an attempt would technically only result in guarantees for protocols with highly synchronized clocks but not for protocols with loosely synchronized or unsynchronized clocks, while traffic-related timing attacks solely depends on the adversary's clock and not on the protocol parties' clocks. TUC grants the adversary access to a precise clock, independent of the parties' clocks.

Networks of timed automata are well studied. However, they are seldom used for cryptographic purposes since timed automata are not as expressive as Turing machines. Thus, networks of timed automata are not sufficiently expressive for the analysis of cryptographic protocols. In particular, an adversary represented by a timed automata would be too weak.

There has been work on the time sensitive analysis of Dolev-Yao style abstracted cryptographic protocols using timed automata [7, 15, 33, 38, 37, 40]. However, this line of work analyzes Dolev-Yao style abstractions and therefore does not offer the full generality of networks of Turing machines.

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 [28] and the framework by Unruh [54], but fundamentally extends these frameworks by incorporating a notion of time while preserving the highly desired properties such as universal composability

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

(a) Tree Structure of Parties   (b) Network Adversary and Communication
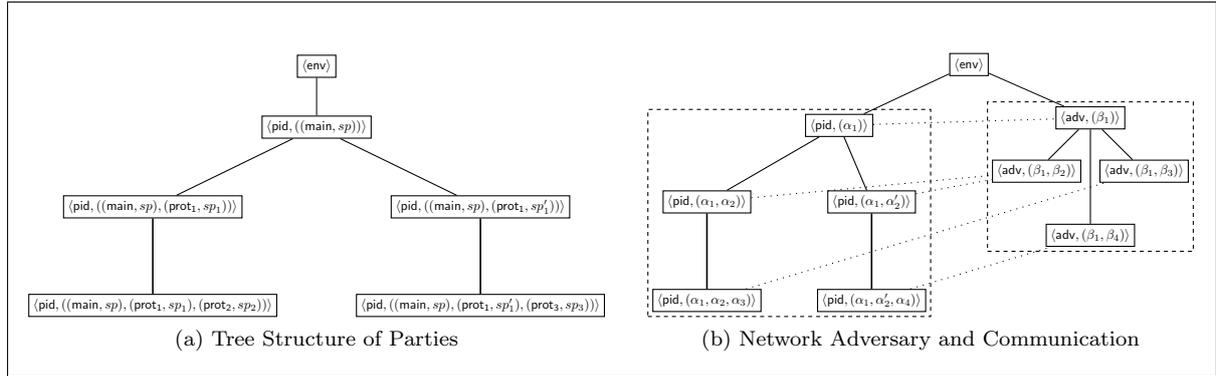
Figure 1: The Network Model in TUC

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 adversary might consist of several machines that work in parallel. A natural way of modeling this capability is to represent these environment and the adversary 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 adversary 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.

3.1 Execution

The whole network is run inside single a machine we call 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. We define the output of EXEC as the output of the environment ENV after observing the communication between the involved parties. We capture this output by introducing a random variable.

Definition 1. *The execution EXEC is a probabilistic, poly-time Turing Machine which receives the security parameter η and outputs a value in $\{0,1\}$. $\mathsf{EXEC}_\eta(\Pi, \mathcal{A}, \mathrm{ENV})$ denotes the output of ENV after EXEC ran the network of machines running protocols in Π together with the network adversary $\mathcal{A}$ and the environment ENV. EXEC stops whenever ENV halts and outputs a bit.*

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 8.

3.1.1 Machines & Session Identifiers

In order to adequately represent complex protocols in our model, we adopt the notion of protocol machines from GNUC [28].[1] Each party P that participates in network communication is represented by a tree of machines, each of which provides the sub-protocols used by P. This tree structure simplifies the substitution operation, which is used in the composition theorem (presented in Section 3.3.1). Our definition of a party is in line with what is presented as a structured system of interactive machines presented in [28, Section 3].

Each machine M in the network is identified by a unique *machine ID* id_M. This machine ID consists of a party identifier pid_M and a session identifier sid_M: $\mathsf{id}_M = \langle \mathsf{pid}_M, \mathsf{sid}_M \rangle$. The session identifier

---

[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.

$\mathsf{sid} = (\alpha_1, \ldots, \alpha_k)$ consists of the *parent session identifier* $(\alpha_1, \ldots, \alpha_{k-1})$, the protocol name $\mathsf{protNAME}$, some session parameters sp, and the role: $\alpha_k = (\mathsf{protNAME}, sp, \mathsf{role})$. The protocol name is used to determine the code executed by the respective machine, as detailed in Section 3.3.

Definition 2. *The machine ID id_M of a machine M is a tuple $\mathsf{id}_M = \langle \mathsf{pid}_M, \mathsf{sid}_M \rangle$, where pid is the* party identifier *and* $\mathsf{sid} = (\alpha_1, \ldots, \alpha_k)$ *the* session identifier *of M. The last component of $\alpha_k = (\mathsf{protNAME}, sp, \mathsf{role})$ is the* basename *of M, where $\mathsf{protNAME}$ specifies* protocol name *executed by M, sp contains* session parameters, *and* role *specifies the role in the protocol, e.g., client or server. The machine ID is unique to each machine.*

Given such machine IDs id_M, we define a *party* as a collection of machines $\{M_1, \ldots, M_l\}$ such that each machine M_i has the same party identifier pid_{M_i} and the session identifiers of all machines induce a tree, if the parent session identifier of each machine is the session identifier of another machine (except for the root).

Definition 3. *A party P is a collection of machines $\{M_1, \ldots, M_l\}$ with following properties:*

$P1:$ $\forall i, j \in \{1, \ldots, j\} : \mathsf{pid}_{M_i} = \mathsf{pid}_{M_j}$

$P2:$ $\exists! M \in \{M_1, \ldots, M_l\} : sid_M = (\alpha)$ *(called* root *of P)*

$P3:$ $\forall M_j \in \{M_1, \ldots, M_l\} \setminus \{M\}, \exists M_i \in \{M_1, \ldots, M_l\} : \mathsf{sid}_{M_j}$ *is a one-step extension of* sid_{M_i}, *i.e.* $\mathsf{sid}_{M_i} = (\alpha_1, \ldots, \alpha_{k-1})$ *and* $\mathsf{sid}_{M_j} = (\alpha_1, \ldots, \alpha_k)$

By Definition 2 and 3, the set of machines inside a party P put up a a tree, given we understand each session ID as a node in a graph and have an edge between session IDs that are one-step extensions of each other.

Corollary 1. *A party P consists of a tree of machines.*

Accordingly we will occasionally denote machines in such a machine-tree as *nodes*, and, given a pair of machines M_i and M_j satisfying property $P3$, we call M_i *parent* of M_j, and M_j *child* of M_i.

As we will discuss in Section 3.1.4, we restrict communication between machines to communication between parent-children pairs, and machines that are so-called *peers*: a peer is a machine with the same protocol name $\mathsf{protNAME}$ and the same session parameter sp for a basename $(\mathsf{protNAME}, sp, \mathsf{role})$.

Definition 4. *Two machines M and M' with the basenames $(\mathsf{protNAME}_M, sp_M, \mathsf{role}_M)$ and $(\mathsf{protNAME}_{M'}, sp_{M'}, \mathsf{role}_{M'})$ are* peers *if $\mathsf{pid}_M \neq \mathsf{pid}_{M'}$ and they both have the same protocol name $\mathsf{protNAME}_M = \mathsf{protNAME}_{M'}$ and the same session parameter $sp_M = sp_{M'}$.*

This formalization follows the intuition that a party P represents the protocol stack executed on a real world machine, and network communication is done between sub-processes running the same protocol-code.[2]

We adopt all of the constraints listed in [28, Section 4,5] that ensure that each party indeed consists of a tree of machines and that each machine can only communicate with its parent, its children and its peers.

Inside a party, a machine in the machine tree can create new machines as children by 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 3.3 – of the machine tree and creates the new machine.

3.1.2 Environment and Adversary

Influences to network communication outside of the regular parties are traditionally captured in two special parties: 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.

Definition 5. *The* network adversary $\mathcal{A}$ *is the unique party with party identifier $\mathsf{pid}_{\mathcal{A}} = \mathsf{adv}$. The environment* ENV *consists of only one machine with machine ID $\mathsf{id}_{\mathrm{ENV}} = \langle \mathsf{env} \rangle$ and is parent of all root machines of parties in the network.*

---

[2]More general protocol models are conceivable; however, we inherit this restriction from GNUC.

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 machine $M \in \mathcal{M}$ gives control to the network execution, to current global time T_M for M is updated according to

$$T_M := T_M + \frac{n}{c_M},$$

where n is the number of steps done by M in its last activation and c_M is its speed.

upon input (time) **from** $M \in \mathcal{M}$	**before every input** (cmd, t) **from** $M \in \{\mathcal{A}, \mathbf{Env}\}$
1: retrieve T_M	1: **if** $t > 0$ **then**
2: compute local time $t_M := f_M(T_M)$	2: set $T_M := T_M + t$
3: activate M with input t_M on the time tape	3: proceed with cmd
	4: **else** activate M with error

Figure 2: Timing and Initialization in EXEC with machine set $\mathcal{M}$, where f_M is the $M's$ local time function and $f_M = id$ for $M \in \{\text{ENV}, \mathcal{A}\}$

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 receives intercepted messages from the network originating from machines with the respective basename. The distributed design of the adversary allows us to formulate the construction for the composition theorem in Section 4.2.2 in a much simpler way. The tree-structure of the adversary is depicted in Figure 1b.

3.1.3 Timing

We finally extend the basic communication model presented in the previous sections to include time. We achieve this by attaching a timer to each machine in the network and by utilizing the execution EXEC to maintain these timers. In order to allow un-synchronized clocks between network parties, we introduce local-time functions, which transform the timer's value to the local time experienced by each machines. We achieve time-consistency for messages exchanged between machines by introducing time-stamps for these messages. The execution EXEC utilizes these time-stamps for a timely delivery of messages to the recipient.

   We introduce time into our model by assigning a timer to every machine in the system.

Definition 6. *The* timer $T_M \in \mathbb{Q}$ *of a machine M is a rational-valued variable associated with M that is maintained by the execution.*

The timer T_M is initialized to 0 at the beginning of the execution. T_M records the current global time of M and is updated every time M returns control to the execution. How much T_M is updated depends on the speed of M. Each machine has a different speed. Except for the environment and the adversary, the speed of each machine M is characterized by a *speed coefficient* c_M, which specifies how many computation steps M does per time unit. Hence, for the timer T_M of M we have

$$T_M := T_M + \frac{n}{c_M}$$

where n is the number of steps M did in its last activation.

   The poly-time notion we use in our communication model (see Section 3.1.9) necessitates that a machine makes a polynomially bounded number of steps per time unit: a machine with an exponential speed coefficient would not be able to meaningfully progress in time as each machine in our network model is restricted to at most a polynomial number of computation steps per activation. Hence, we require the speed coefficients be polynomials.

Definition 7. *The* speed coefficient *of a machine M specifies the number of computation steps that M can perform per unit time. The speed coefficient is a polynomial $c_M \in \mathbb{N}[X]$. Whenever M returns control to the execution, T_M is updated by $T_M := T_M + \frac{n}{c_M(\eta)}$ where n is the number of steps M did in its last activation and η is the security parameter.*

   Local time functions are needed to model unsynchronized local clocks, only loosely synchronized clocks, or too fast or too slow clocks. Each machine M can requests its local time by sending a (time)

request to the execution. EXEC then computes the local time of M by applying M's local time function f_M to its current global time T_M. The execution and the simulating machine in the internal simulation lemma (Lemma 2) need to be able to efficiently compute and invert the local time; thus, we require that the local time function be efficiently computable and invertible function.

Definition 8. *The* local time function $f_M : \mathbb{Q} \to \mathbb{Q}$ *of M is a strictly monotonically increasing, efficiently computable and efficiently invertible function that transforms the value of T_M to M's local time $f_M(T_M)$.*

We require the local time function to be invertible 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 7. We make a worst case assumption and define the local time function of the environment ENV and the network adversary $\mathcal{A}$ to be the identity function(see Figure 2).

Speed coefficients and local time functions are fixed once the machine is spawned. Formally, the speed coefficient and the local time function depend on the protocol, the session parameter, the role, and the party ID. In order to assign speed coefficients to dynamically created machines, we require the protocol Π to determine for each basename not only the code to be executed but also a distribution over the speed coefficients and a distribution over local time functions (see Definition 21). The execution draws the speed coefficients from these distributions whenever a new machines is created during runtime.

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 amount of computation steps per unit time.[3]

Definition 9. *A machine M is* timeless *if it does not have a speed coefficient and M itself tells the execution the time-difference by which its timer increases next time it returns control to the execution.*

Note that by the notion of poly-time we introduce in Section 3.1.9 this still restricts both to at most a polynomial number of computation steps (in the security parameter η) per activation.

3.1.4 Communication Model

We differentiate between inner party communication between parent and children nodes inside a party and network communication between peers using a notion of *ports*: we distinguish between *environment*, *subroutine*, and *network port*. Figure 1b illustrates this with thick lines for inner party communication and dashed lines for network communication.

Definition 10. *A* port p *of a machine M is a set of one input tape p_{in} and one output tape p_{out} that is used by the execution to pass information to M. Each machine has one* environment port $\mathsf{sid}(M).\text{ENV}$, *a* network port $\mathsf{sid}(M)_{\mathsf{net}}$ *and a set of* subroutine ports $\mathcal{S}_M$.

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 as well. 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.

M uses it environment port $\mathsf{sid}(M).\text{ENV}$ to communicate with its parent, or if M is a root node, with the environment. The set of subroutine ports $\mathcal{S}_M$ contains a unique port for each child of M. 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 3.

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 [28, Section 4]. The methods used for message passing inside EXEC are presented in Figure 3.

---

[3]For the completeness of the dummy adversary, the adversary needs to be able to forward message in a way that is unobservable for the protocol even though every message-forwarding costs time. For that situation, we make use of the timelessness of the dummy adversary and show that if the dummy adversary proceeds in exponentially small steps, message-forwards remains unobservable for the protocol (see proof of Lemma 3).

Figure 3: Communication methods in EXEC with machine set $\mathcal{M}$ and protocol Π

For the case that a machine waits for incoming messages, we introduce a listen-command: (listen, T). As soon as this command is sent, the execution EXEC will not activate these machines until either (i) they receive a message, or (ii), they can no longer receive a message before time T. The machines also have the possibility to set $T = \infty$ the machines then will not be activated unless they receive a message. Once the machine is activated, their timer is also updated, either to the time-stamp of the received message, or T if the machine is activated without a message.

Timing in communication. In contrast to other sequential activation models, messages in TUC are not directly delivered to the recipient because there might be another message from a yet not activated machine that has to arrive earlier. More formally, if M sends a message to M' and $T_M > T_{M'}$, then the message obviously should not reach M' until $T_{M'} \geq T_M$. The execution remedies this problem by utilizing time stamps.

Definition 11. *The* time stamp *of a message m sent by a machine M is the updated value of the timer T_M at the point when M sends the message.*

The execution attaches this time-stamp to each message before it is sent to the recipient. On the recipient's end we use time-ordered queues, called *input queues*, which organize all messages that still need to be received, and release them once the recipient has progressed far enough in time. It is crucial that we deliver all messages at once; otherwise it is not clear how to internally simulate several machines (see the proof of Lemma 2).

Definition 12. *The* input queue Q_M *of a machine M is a priority queue which receives all messages directed to M as input and uses their time-stamp as the keys which are sorted. On request with a time-stamp T, Q_M returns all messages with time-stamp $T_m \leq T$.*

When delivering m, EXEC puts the tuple (m, T_m, p) into the input queue $Q_{M'}$ of M'. Here p denotes the port through which M' receives the message m. Whenever M is in the listen state, i.e. listen for new messages, the execution retrieves all messages from Q_M with time-stamps $T_m \leq T_M$ and forwards

9

```
activate_listen(M)                                    upon output (listen, T) from M ∈ M
 1: if ∀M' ∈ M \ {M} : T̃_{M'} ≥ T̃_M then              1: set T̃_M := T
 2:    if M is timeless then                           2: activate_listen(M)
 3:       set T_M = T̃_M
 4:    else                                           upon activation by M ∈ M without output
 5:       set T_M = ⌈T̃_M · c_M⌉ / c_M                 1: activate ACT with activation request
 6:    if Q_M is not empty then
 7:       Pull messages (m_1, T_{m_1}, p_1), …, (m_l, T_{m_l}, p_l)   upon input (activate, M) from ACT
           with time-stamps T_{m_i} ≤ T_M from Q_M     1: if M in listen state then
 8:       activate M with m_1, …, m_l on ports         2:    call activate_listen(M)
           p_1, …, p_l                                 3: else
 9:    else                                            4:    activate M
10:       activate M without input
11: activate ACT with a activation request
```

Figure 4: Activation order methods in EXEC with machine set $\mathcal{M}$ and activation strategy ACT

them to M. In case there are several messages in $Q_{M'}$ that could be retrieved at time $T_{M'}$, $Q_{M'}$ simply returns all of them. All methods involved in message passing are presented in Figure 3.

3.1.5 Consistency Enforcing Scheduling

Other simulation-based frameworks use a sequential activation model: machines in the network directly activate each other by sending messages and the environment or the adversary decides which machine is activated in case no messages. We call these decisions the *activation strategy*. Keeping to this unrestricted sequential activation model, however, causes several problems as soon as time is introduced: messages from the past arrive at nodes which are already in the future or the activation order (which is usually represented by the adversary or the environment) can push machines arbitrarily far into the future. The example below shows how this can lead into problems with a timeout mechanism, assuming the adversary decides the activation order.

Example 1: Inconsistencies with unrestricted sequential activation strategies. Consider machines M and M' which go into a timeout state if they do not receive a message up to 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. ◇

We avoid such inconsistencies as follows: we introduce a special (listen, T) state for the machines in the network, in which they have to be in order to receive messages. Furthermore, we deviate slightly from the traditional sequential activation model by discarding activation commands that do not satisfy the following consistency enforcing property.

Definition 13 (Consistency Enforcing). *An activation order of machines is* consistency enforcing, *if* $\forall M' \neq M : T̃_M \leq T̃'_M$ *whenever M is activated from the listening state. Here $T̃_M$ is the* virtual time *of the machine M defined as follows:*

- $T̃_M = \min\{T, T_m\}$, *where T_m is the smallest time-stamp of a message in Q_M (or ∞ if no such message exists), if M is in the (listen, T) state*

- $T̃_M = T_M$, *if M is not in the (listen, T) state*

Consistency enforcing activation orders resolve 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 up to 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.

Upon activation with input (write, D, d, M, T_M):
1: let $D_{[T_i, \infty]}$ be the last version of D
2: set $D_{[T_i, T_M]} := D_{[T_i, \infty]}$
3: remove $D_{[T_i, \infty]}$
4: set $D_{[T_M, \infty]} := d$
5: return confirmation to M

Upon activation:
1: **if** there is an input (lookup, D, M, T_M) **then**
2:     put the request (lookup, D, M, T_M) into the queue Q_L
3: let T^* be the smallest timer value in $\mathcal{M}_{\mathsf{MEM}}$
4: **while** $\exists$ lookup requests Q_L with time stamp $T \leq T^*$ **do**
5:     remove request (lookup, $D', M', T_{M'}$) with smallest time stamp $T_{M'}$ from Q_L
6:     return data set $D_{[T_i, T_{i+1}]}$, $T_M \in [T_i, T_{i+1})$ to requesting party M

Figure 5: The shared memory MEM

Corollary 2. *Under consistency enforcing activation orders, whenever a machine M is activated from the* listen *state at time T, M receives all messages m with time-stamp $T_m \leq T$ and any messages not yet received were sent at a time $T' > T$.*

3.1.6 Activation strategies

We need an *activation strategy* to determine the machine to be activated next whenever a machine turns inactive after switching into the listen state. Under consistency enforcing activation orders, however, we cannot take the same approach as previous frameworks, in which the adversary decides the activation strategy: since the adversary is a time-sensitive component of the network and thus also affected by consistency enforcing activation order, the network can end up in a deadlock situation where all machines can either not be activated, or are stuck in the listen state. To avoid this problem we introduce the activation strategy as a sub-machine of the execution EXEC.

Definition 14. *The* activation strategy ACT *is a probabilistic, poly-time TM, which, given the state of the network as input, determines the next machine to be activated.*

ACT does not have a timer and implements the activation order based on the current state of the network. EXEC enforces the consistency enforcing property be checking the required conditions whenever ACT wants to activate a machine. In Section 3.2.1, we show that all consistency enforcing activation orders are equivalent. The activation methods are depicted in Figure 4.

3.1.7 Shared Memory

As in other simulation-based framework, our goal is to analyze complex protocols by simplifying them to ideal functionalities with 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.

Definition 15. *A* shared memory MEM *is a machine without a timer which, given the current state of the network, implements time-sensitive data exchange outside of message passing. The set of machines with access to* MEM *is denoted with* $\mathcal{M}_{\mathsf{MEM}}$.

    MEM *maintains every version $D_{[T_1, T_2]}$ of each data-set D with respect to time (organizing them through time-intervals $[T_1, T_2]$ between changes) and on request at time T returns the version $D_{[T_i, T_{i+1}]}$ of data set D with $T \in [T_1, T_{i+1})$*

Note that, similar to the activation strategy presented in the last section, shared memory does not have a timer, is therefore outside of time. Technically this means that both activation strategy as well as shared memory are part of the execution, which is the only part of our network model that is not time-sensitive. We however separate them from EXEC as sub-machines for a more modular definition.

Similar to the activation orders, the concept shared memory causes consistency issues: 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 shared 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 activation orders.

Definition 16. *Shared memory access is* consistency enforcing *if a lookup requests* $(\mathsf{lookup}, D, M, T_M)$ *is only processed if* $\forall M' \in \mathcal{M}_{\mathsf{MEM}} : T'_M \geq T_M$.

If this condition is not true, the shared memory puts the request together with its time stamp into a input 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. Figure 5 gives a possible pseudo-code implementation of a shared-memory unit in the time-sensitive network model.

As a consequence of consistency enforcing memory access we get Corollary 3 which ensures consistency of shared memory entries read by machines in the network.

Corollary 3. *If a data set D is read by a party M from a consistency enforcing shared memory MEM at time T, then any changes to D will only happen at a point in time $T' \geq T$.*

3.1.8 Compromisation

An important part in the analysis of protocols is accurately modeling adversarial capabilities, which includes restricting the adversary's access to the network, as well as differentiating between active and passive adversaries.

Link corruption. 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: Tor, e.g., is not secure against global adversaries [55, 47, 44, 14], it is not even designed to be secure against global adversaries. Previous work on the analysis of Tor shows how partial compromisation of the network can be modeled by introducing special network functionalities $\mathcal{F}_{\mathrm{NET}}$, which are used as a link between parties [2]. $\mathcal{F}_{\mathrm{NET}}$ only leaks a message to the network adversary if the communication link they represent is compromised.

To simplify the analysis, especially with regard 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 should be compromised. Afterwards, any communication on the compromised link is forwarded to the adversary $\mathcal{A}$ (see Figure 6).

We assume that inner-party communication, i.e. communication between children and parent nodes inside a party P, cannot be intercepted without compromising the party: a party models a system that resides at one physical location.

In order to keep EXEC modular, we introduce a network topology sub-machine NET, which handles all requests regarding compromised links: compromisation requests from ENV are forwarded to NET, which internally maintains the corruption status of the network, and on request from EXEC, determines whether a messages can be directly forwarded to the recipient, or is intercepted by the network adversary. As we discuss for future work in Section 8, NET can also be extended to include network latency and other parameters of the network.

```
upon (compromise, id₁) from parent of M_id₁        upon input (compromise, id₁, id₂) from Env
 1: if M_id₁ ∉ C_M then                             1: NET(compromise, id₁, id₂)
 2:    replace M_id₁ code to cd_comp                2: activate ENV with (compromised, id₁, id₂) as
 3:    C_M := C_M ∪ {M_id₁}                             input.
 4:    send (compromise) to M_id₁
 5: else
 6:    return error to ENV
```

Figure 6: Corruption Commands in EXEC for Adaptive Compromisation

```
upon (compromise, id₁) from parent of M_id₁
 1: if M_id₁ ∉ C_M ∧ M did not receive a message yet then
 2:    replace M_id₁'s code to cd_comp
 3:    C_M := C_M ∪ {M_id₁}
 4:    send (compromise) to M_id₁
 5: else
 6:    return error to ENV
```

Figure 7: Machine Corruption in EXEC for Static Compromisation

Party corruption. The network machines themselves can also be completely compromised by the environment ENV. Upon receiving a compromisation message compromise, EXEC replaces the code executed by the receiving machine M with the following code of a compromised machine cd_{comp} and 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. 6). cd_{comp} is defined as follows: whenever M receives a message, it is forwarded to the adversary, who in turn instructs M.

Since the adversary is modeled as a network party, we cover passive as well as active adversaries: a passive adversary would simply forward all messages he intercepts, while an active adversary can send additional messages through the network as well as change intercepted messages.

The analysis of AC protocols usually differentiates between two important classes of Compromisation.

Definition 17. *An execution allows for* static compromisation *if machines and communication links can only be compromised at initialization. It allow for* adaptive compromisation *if the environment can compromise even during run-time.*

While the presentation of EXEC in Figure 8 works for the adaptive case, we need to make some changes for the static case: 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 they 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. 7).

3.1.9 Runtime Bounds

Correctly addressing polynomial runtime bounds for networks of machines has been a point of major debate in the literature [29]. We adopt the solution put forward in [28, 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 [28, Section 6].

The IITM-model proposes a simpler notion of poly-time machines which is compatible with our communication model as well. As discussed in [28, Section 11.8] however, this notion does not imply poly-boundedness for the composition of poly-bounded machines, which the definition in GNUC does.

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. This ensures that on any activation, the activated machine has an input. We define polynomial time based on this input.

Run-time of machines in the network is defined based on the length of the messages they exchange, i.e. the *flow* between machines. To this end we denote with $f_{ep}(\eta)$ the flow from the environment Env to protocol machines Π, with $f_{ea}(\eta)$ the flow from Env to the adversary $\mathcal{A}$, and with $f_{ap}(\eta)$ the flow from $\mathcal{A}$ to Π.

We only consider environments which on each activation are polynomially bounded in their input and which additionally are well-behaved.

Definition 18. *An environment* Env *is well-behaved if there is a polynomial p such that for every adversary and for all security parameters η,*

$$f_e(\eta) = f_{ep}(\eta) + f_{ea}(\eta) \leq p(\eta)$$

With this, we can define probabilistic, poly-time protocols.

Definition 19. *Let $T_\eta^\Pi[\Pi, \text{Env}, \mathcal{A}]$ denote the accumulated number of steps done by all machines running Π. A protocol Π is* probabilistic, poly time *(PPT) if there exists a polynomial p such that for all every well-behaved environments* Env, *$T_\eta^\Pi[\Pi, \text{Env}, \mathcal{A}]$ can be bounded by*

$$Pr[T_\eta^\Pi[\Pi, \text{Env}, \mathcal{A}] > p(f_{ep}(\eta) + f_{ea}(\eta))] \leq negl(\eta)$$

In order to get an overall poly-time execution, we also only allow bounded adversaries.

Definition 20. *An adversary $\mathcal{A}$ is* bounded *for Π if $\mathcal{A}$ is time-bounded for Π, i.e. there exists a polynomial p such that for all well-behaved environments* Env *we have that*

$$Pr[T_\eta^\mathcal{A}[\Pi, \text{Env}, \mathcal{A}] > p(f_{ep}(\eta) + f_{ea}(\eta))] \leq negl(\eta)$$

and if $\mathcal{A}$ is flow-bounded for Π, i.e. there exists a polynomial p' such that for all well behaved environments

$$Pr[f_{ap}(\eta) > p'(f_{ea}(\eta))] \leq negl(\eta)$$

With these restrictions we get an execution EXEC which overall uses a polynomial number of steps in the security parameter η.

Lemma 1. *For all well-behaved environments* Env, *all PPT protocols Π, all bounded adversaries $\mathcal{A}$ for Π and all inputs $x \in \{0,1\}^{p(\eta)}$ the execution $\text{EXEC}_\eta(\Pi, \mathcal{A}, \text{Env})$ is probabilistic polynomial time in η with overwhelming probability.*

The lemma follows, on the one hand, from Π and $\mathcal{A}$ being polynomially time-bounded in their incoming flow (with overwhelming probability), $\mathcal{A}$ being flow bounded (with overwhelming probability), and Env being well-behaved. On the other hand we have that EXEC only requires polynomial time to update the timers of each machine, compute their local time-functions, compute the input queues of each machine and check the condition for consistency enforcing activation orders. EXEC therefore overall only uses polynomial time.

Note that we also inherit the notion of *invitations* from GNUC. This technical artifact is sometimes required for the construction of the simulators in proofs of secure realization (see 4), which often cannot wait until they receive a message from the environment before they interact with protocol machines. For brevity we do not directly include them into our construction, but stress that their inclusion do not cause any problems with runtime bounds: invited messages do not count towards the flow we bound. But since only polynomially many invitations are generated, the overhead of invited messages can be polynomially bounded as well.

3.1.10 Discussion

Modeling asynchronous communication despite consistency enforcing scheduling. In other simulation-based framework, such as UC, GNUC, RSIM, IITM, the environment (or sometimes the network adversary) decides which machines are activated next. Quantifying over all possible activation strategies in particular includes those scenarios in which a message transmission is arbitrarily delayed. Canetti argues that such an activation order is chosen for modeling asynchronous communication [12, page 28].

In spite of consistency enforcing activation orders, TUC can be used to model asynchronous communication by protocols that do not use their local clock.[4] Arbitrary networks delays for compromised links can be modeled in TUC since the network adversary can arbitrarily delay a message.

Beside the advantage that asynchronous communication can be modeled by arbitrary activation strategies, another effect of (traditional) arbitrary activation strategies is that it can model, by not activating a machine, crashed nodes, but only if these machines would not have received any message (otherwise this message would activate them). Since this mechanism anyway only covers machines that would not have received any messages, we believe that such crashes should rather be explicitly modeled, i.e., in the same way as corruptions, and not via an activation strategy.

Encoding our approach in previous frameworks. It might be possible to wrap each machine M in the network in a local wrapper W that performs the same actions as the execution EXEC in TUC. W would count the number of steps that M performs and divides these steps by the speed of that party to calculate M's current time. This wrapper W would for each outgoing message from M add a timestamp and for each incoming remove the timestamp before forwarding it to the recipient. Moreover, W would order every incoming messages in a time-ordered input queue and only deliver those message to M for which M already proceeded far enough in time.

Such a network of locally wrapped machines $W(M_1), \ldots, W(M_n)$, however, does not ensure the consistency enforcing property for activation orders, i.e., allows inconsistent activation orders (see Example 1). Since consistency enforcing is a global property the local wrappers W would have to synchronize their timer information to find the next, eligible party and activate this party (by sending some dummy message). Although it might be possible to show that such an encoding is equivalent to our approach, we believe that it is more elegant and easier to understand to incorporate time-sensitive adversaries as done in TUC.

3.2 Properties of EXEC

We present two properties which will be helpful for proving the security properties of TUC.

3.2.1 Simplified activation strategy

It turns out that under consistency enforcing all activation strategies are equivalent to the following activation strategy SAS, as long as no deadlocks occur: the next machine to be activated is selected based on each machine's timer T by randomly selecting a machine with the lowest timer value.

Theorem 1. *Let* $\mathsf{EXEC}'_{k,S,t}(\Pi, \mathcal{A}, \mathrm{ENV})$ *denote the machine that executes* $\mathsf{EXEC}_k(\Pi, \mathcal{A}, \mathrm{ENV})$ *with the activation strategy* S. *Moreover* $\mathsf{EXEC}'_{k,S,t}(\Pi, \mathcal{A}, \mathrm{ENV})$ *records the internal states of all machines in the execution after each step together with the global time in which that machine was during that state. After the execution finished,* $\mathsf{EXEC}'_{k,S,t}(\Pi, \mathcal{A}, \mathrm{ENV})$ *outputs the sequence of all states of all machines (including $\mathcal{A}$ and* ENV*) up to the time t. Let* $\mathrm{reach}(S, t)$ *denote the following property: the probability that for all $\mathcal{A}$ and all* ENV *and all $x \in \{0,1\}^*$ in* $\mathsf{EXEC}'_{k,S,t}(\Pi, \mathcal{A}, \mathrm{ENV})$ *all machines reach a time $> t$ is overwhelming in k.*

Let A be a (potentially timeless) machine and let S_1, S_2 be any two activation strategies, i.e., any two machines that, given all the information that the execution EXEC *has, (adaptively) determine which machine shall be activated next. For all sets of machines $P := \{P_1, \ldots, P_n\}$ ($n \in \mathbb{N}$), S_1 is indistinguishable from S_2, i.e., for all points in time $t \in \mathbb{Q}$ and for all distinguisher machines D there is a negligible function such that:*

$$\mathrm{reach}(S_1, t) \ and \ \mathrm{reach}(S_2, t) \implies$$
$$\Big| \Pr[b \leftarrow \langle D \mid \mathsf{EXEC}'_{\eta, S_1, t}(\Pi, \mathcal{A}, \mathrm{ENV}) \rangle : b = 1]$$
$$- \Pr[b \leftarrow \langle D \mid \mathsf{EXEC}'_{\eta, S_2, t}(\Pi, \mathcal{A}, \mathrm{ENV}) \rangle : b = 1] \Big|$$
$$\leq \mu(\eta)$$

Proof. First, we prove the following statement.

---

[4]Even the slightly stronger setting in which each party uses its local clock, but the clocks are completely unsynchronized can be modeled by unsynchronized local time functions (see Section 3.1.3).

Claim 1: For two activation strategies S_1 and S_2, $\mathsf{EXEC}'_{\eta, S_1, t}(\Pi, \mathcal{A}, \textsc{Env})$ is indistinguishable from $\mathsf{EXEC}'_{\eta, S_2, t}(\Pi, \mathcal{A}, \textsc{Env})$ if and only if the sequence of results of all listen-commands, i.e., the input message with which the respective machine is activated, is indistinguishable in both scenarios.

Proof of Claim 1. By the definition of EXEC all the information that A can observe is independent of the activation order except of the results of the pull-queries to the input queue Q_A. Hence, it is indistinguishable whether the strategy S_1 or S_2 has been used. The reverse direction holds because every machine can use the result from the listen -command to distinguish the two settings. ◇

Let $\mathsf{EXEC}'_{\eta, S_1, t, i}(\Pi, \mathcal{A}, \textsc{Env})$ be the execution (with the activation strategy S_1) that stops after the ith activation of a machine that is in the listen state. Let $\mathsf{EXEC}'_{\eta, S_2, t, i}(\Pi, \mathcal{A}, \textsc{Env})$ be defined analogously.

We perform a proof by induction over the activation of machines that are in the listen state.

Induction basis ($i = 0$): By Claim 1 the two activation strategies are indistinguishable.

Induction step ($i > 0$): We know that for all $i - 1$th listen activations the two activation strategies S_1 and S_2 are indistinguishable. Thus, the results of all $i - 1$ listen-commands are indistinguishable for both activation strategies S_1 and S_2. Thus, by Claim 1, it remains to show that also seeing the result of the ith listen-command is indistinguishable.

Let M be the ith machine that is activated in the listen state. If no machine sent a message to M since the $i - 1$th listen-command, the statement follows by induction hypothesis. Assume that at least one message has been send to M since the $i - 1$th listen activation. Let T be the global time of M when it issue the listen-command. By consistency enforcing property of the activation order, we know that all machines are at least in time $T' \geq T$ when M is activated in its listen state for any activation strategy S that lets all machines reach the (global) time T. By the definition of EXEC, we know that after M is activated (upon a listen-command) no machine can send a message M that will get a timestamp $\leq T$. Hence, the input of the listen activation, i.e., the inputs from the input queue Q_M, are independent from the activation strategy. Hence, by Claim 1, the statement follows. □

Theorem 1 directly implies that the activation strategy SAS is indistinguishable from any other activation strategy as long as no deadlock occurs.

Corollary 4 (SAS is equivalent to all deadlock-free activation strategies). *The activation strategy* SAS *is indistinguishable from any other, deadlock-free activation strategy(in the sense of Lemma 1).*

3.2.2 Internal Simulation of Multiple Machines

Here we show that a finite number n of timed machines $M_1, \ldots, M_n$ in our network can be simulated by a single machine M^* which shows same behavior as the n separate machines. While the internal simulation is clearly not a problem if M^* is timeless (even if some of the simulated machines are timeless), this is not clear for the case where M^* is a timed machine with a timer that automatically progresses in time whenever M^* does a computation step.

The following lemma will allow us to simplify large parts of the theorems we show regarding the security definition we introduce in section 4.

Lemma 2. *For any set of n timed machines $M_1, \ldots, M_n$ in the network there exists a single timed machines M^* which shows the same behavior as $M_1, \ldots, M_n$ for consistency enforcing activation orders.*

Proof. We construct and show that M^* sends the same messages as $M_1, \ldots, M_n$, together with the same time-stamps, into the network.

Construction: We set the speed coefficient of M^* to $c_{M^*} = n \cdot c_{M'} + O(n)$, where $M' \in \{M_1, \ldots, M_n\}$ is the machine with the highest speed coefficient. The local time function of M^* is the identity.[5] This allows M^* to do at least one computation with every internally simulated machine without progressing in time further than M' (since each simulated computation step also causes one computation step in M^*).

M^* inherits all ports of the machines it simulates. Thus all messages intended for the internal machines are first received by M^*, who then forwards them appropriately.

In particular this implies that M^* has more than one network port: M^* has n network ports, each of which are identified with the identifiers of $M_1, \ldots, M_n$. This is necessary since M^* needs to be able to differentiate between the recipients of messages coming into $M_1, \ldots, M_n$ and the senders of messages

---

[5]This is important for recomputing the execution EXEC.

going out of $M_1, \ldots, M_n$. Since EXEC automatically identifies network ports based in machine ID, messages targeted at machines inside M^* are automatically forwarded to M^*.

In principle, M^* simply emulates the network execution for the machines it simulates, managing message passing from the machines to the real execution EXEC and the activation order. As shown in Lemma 1, M^* can adopt the strategy of activating the machine with the lowest timer value.

In particular, M^* now works as follows: on regular activation, M^* simulates each machine $M_1, \ldots, M_n$ for one computation step at a time, each time activating a machine M' with the lowest timer $T_{M'}$ (randomly choosing one if more than one machine is at time $T_{M'}$). M^* stops the simulation if all machines with the lowest timer value are either in the listening state or returned control to the execution. M^* sends each message sent out by the simulated machines using the delayed sending command (see Figure 3), using the appropriate time-stamps.

M^* performs idle steps until its own timer matches the lowest timer value T'. Then, M^* turns into the listening state of the network port of machine M_i if there is a machine M_i at T' that is in the listening state (randomly choosing one if there is more than one), or simply returning control to EXEC otherwise.

If activated from the listening state, M^* automatically receives all messages on its ports with time-stamps smaller than T_{M^*}. These messages are forwarded to the message queues of the machines inside M^*, with T_{M^*} as a time-stamp, and each machine is in turn activated for one computation step each. M^* then continues with the usual activation order as described above.

Upon a command (listen, T) from a party, M^* computes the virtual time $\tilde{T}_M$ for each machine M and handles the (listen, T) command as EXEC. M^* can compute for a machine M the virtual time $\tilde{T}_M$ since it knows the speed coefficient of M and the content of the input queue of M.

We show by induction over the activations of M^* that M^* shows the same behavior towards the rest of the network as $M_1, \ldots, M_n$:

Activation 1: On initial activation, all timers are the initial value, and no machine is in the listening state. M^* will in turn simulate each internal machine until all machines with the lowest timer T' either sent a message or went into the listening state.

Due to our selection of the speed coefficient c_{M^*} of M^*, we have that at this point $T_{M^*} < T'$, with enough time for M^* to forward all messages sent by the simulated machines using the delayed sending command, and turning into the listen state at time T'. Since all messages until T' were sent with the right time stamp, and the simulated machines are activated in the same order as they would have been activated from the execution (due to consistency enforcing scheduling), the behavior of M^* is the same as the behavior of each single machine $M_1, \ldots, M_n$.

Activation $i \rightarrow i + 1$: M^* is either gets activated regularly or from a listening state. The former case is the same as for the initial activation above, hence we only consider the activation from the listening state.

Being activated from a listening state means there are machines with timer value T_{M^*} simulated in M^*, which also are in the listening state. Due to consistency enforcing scheduling, each of these machines receive all messages they would have received in the regular network as well (since all other machines in the network have progressed past T_{M^*}).

Messages forwarded to machines which are not in the listening state at time T_{M^*} correctly receive these messages once they go into the listening state. The potentially changed time-stamp of these messages inside M^* does not influence the behavior of each machine as they do not receive the time-stamp of the message (it is only used by the network execution, which here is simulated by M^*).

Hence all machines receive the same messages as in the regular network, at the same points in time, and thus will also produce the same output messages, at the same points in time, as well. □

We stress that Lemma 2 does not enforce the tree-restrictions of machines inside a party defined in Section 3.1.1. One should still mind these restrictions if composition is to be used on together with above construction.

As a corollary, we can show that a timeless machine can also internally simulate other timeless machines.

Corollary 5. *For any set of n machines $M_1, \ldots, M_n$ in the network there exists a single timeless machines M^* which shows the same behavior as $M_1, \ldots, M_n$ for consistency enforcing activation orders.*

Proof. The simulator is exactly constructed as in the proof of Lemma 2 except that the activation is slightly modified. All timed machines are only activated once in one point in time, but timeless machines are activated (in a round-robin manner) as often as they stay in that point in time. Moreover, if a simulated timeless machine M_i sends a message at time t and no other machine wants to perform a computation, M^* performs a (listen, t) command, where t is the minimal time of the next timed machine that could perform a step and a message that shall be sent or. $\square$

3.3 Protocols

A protocol is a runtime library that assigns to each basename (i.e., protocol name, session parameter, and role) 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 $Dist(X) \subset X \to [0, 1]$ the set of distributions over the natural numbers (without 0). Moreover, we denote with $Mon_{A \to B}$ the set of strictly monotic functions from A to B and with D the set of machine IDs.

Definition 21. *For a set P of party IDs and a set of basenames D, a protocol is a runtime library $\pi : P \times D \to \{0,1\}^* \times Dist(\mathbb{N}) \times Dist(Mon_{\mathbb{Q} \to \mathbb{Q}})$, which assigns every party pid $\in P$ and basename $d \in D$ the code $c \in \{0,1\}^*$ run by every machine with basename d, an efficiently computable speed-distribution $\mathcal{S} \in Dist(\mathbb{N}[X])$ from which the execution EXEC can draw the speed coefficient for newly created machines, and an efficiently computable local-time function distribution $\mathcal{F} \in Dist(Mon_{\mathbb{Q} \to \mathbb{Q}})$ over strictly monotic functions.*
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 machine ID $d \in D$ can call as subroutines. By the requirements listed in in [28, 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.

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

3.3.1 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 [28, Section 5].

Definition 22. *The sub-protocol $\Pi' = \Pi|x$ of Π is the restriction of Π to D', the set of all basenames reachable from the basename 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 basename x.

Definition 23. *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 9 gives an example for such a substitution. On the protocol level, composition comes down to replacing the code provided for all basenames in a sub-tree of the *acyclic call graph* of basenames.

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 $M \in \mathcal{M}$ gives control to the network execution, to current global time T_M for M is updated according to $T_M := T_M + \frac{n}{c_M}$, where n is the number of steps done by M in its last activation and c_M is its speed coefficient.

upon (m, id) **on port** net **from** $M \in \mathcal{M}$ **at time** T_M
1: **if** id is a peer of $\mathsf{id}(M)$ **then**
2: $(r, m, T_m) \leftarrow \mathsf{NET}(\mathsf{id}(M), \mathsf{id}, m, T_M)$
3: **if** $r = \mathcal{A}$ **then**
4:   put (m, TmM, net) into $Q_{\mathcal{A}_i}$ of the corresponding sub-adversary $\mathcal{A}_i$.
5:   activate $\mathcal{A}_i$.
6: **else**
7:   put (m, T_m, net) into Q_{id}
8:   **if** M_{id} in listen state **then**
9:    **if** $T_M \leq \tilde{T}_{M_{\mathsf{id}}}$ **then**
10:     set $\tilde{T}_{M_{\mathsf{id}}} := T_M$
11:    **activate_listen**(M_{id})
12:   **else**
13:    activate M_{id}
14: **else**
15: return error to M

upon m **on port** $p \neq net$ **from** $M \in \mathcal{M}$ **at time** T_M
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_M, p) into $Q_{M_{\mathsf{id}}}$
4: **if** M_{id} in listen state **then**
5:   **if** $T_M \leq \tilde{T}_{M_{\mathsf{id}}}$ **then**
6:    set $\tilde{T}_{M_{\mathsf{id}}} := T_M$
7:   **activate_listen**(M_{id})
8: **else**
9:   activate M_{id}
10: **else**
11: **if** $p = \mathsf{pid}(M).\mathsf{sid}(M).\mathsf{sid}' \wedge$
     sid' proper extension of $\mathsf{sid}(M)$ **then**
12:   let $(cd, \mathcal{S}, \mathcal{LT}) = \Pi(\mathsf{pid}(M), \mathsf{basename}(\mathsf{sid}'))$
13:   sample speed c' from $\mathcal{S}$
14:   sample local time function f from $\mathcal{LT}$
15:   create a new machine M' with code cd,
16:   $\mathsf{sid}(M') := \mathsf{sid}'$
17:   $c(M') := c'$
18:   set up translation of M''s environment port to p
19:   set $T_{M'} = T_M$
20:   put (m, T_M, p) into $Q_{M'}$
21:   activate M'
22: **else**
23:   return error to M

upon input $(\mathsf{compromise}, \mathsf{id}_1, \mathsf{id}_2)$ **from Env**
1: $\mathsf{NET}(\mathsf{compromise}, \mathsf{id}_1, \mathsf{id}_2)$
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}
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 (delay, m, t) **on port** q **from** $M \in \mathcal{M} \setminus \{\mathbf{Env}, \mathcal{A}\}$
1: **if** $t \geq f_M(T_M)$ **then**
2: compute global time $T = f_M^{-1}(t)$
3: execute appropriate send message for m with time stamp T for port q
4: **else**
5: return error to M

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

before every input (cmd, t) **from** $M \in \{\mathcal{A}, \mathbf{Env}\}$
1: **if** $t > 0$ **then**
2: set $T_M := T_M + t$
3: proceed with cmd
4: **else** activate M with error

upon output (listen, T) **from** $M \in \mathcal{M}$
1: set $\tilde{T}_M := T$
2: **activate_listen**(M)

upon activation by $M \in \mathcal{M}$ **without output**
1: activate ACT with scheduling request

upon input $(\mathsf{activate}, M)$ **from ACT**
1: **if** M in listen state **then**
2: call **activate_listen**(M)
3: **else**
4: activate M

activate_listen(M)
1: **if** $\forall M' \in \mathcal{M} \setminus \{M\} : \tilde{T}_{M'} \geq \tilde{T}_M$ **then**
2: **if** M is timeless **then**
3:   set $T_M = \tilde{T}_M$
4: **else**
5:   let $k^* = \min\{k \in \mathbb{N} \mid \frac{k}{c_M} \geq \tilde{T}_M\}$
6:   set $T_M = \frac{k}{c_M}$
7: **if** Q_M is not empty **then**
8:   Pull messages $(m_1, T_{m_1}, p_1), \ldots, (m_l, T_{m_l}, p_l)$ with time-stamps $T_{m_i} \leq T_M$ from Q_M
9:   activate M with $m_1, \ldots, m_l$ on ports $p_1, \ldots, p_l$
10: **else**
11:   activate M without input
12: activate ACT with a scheduling request

Figure 8: 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.

3.4 Ideal Functionalities

Typically, the notion of secure realization is used to prove that a protocol Π is as secure as a simpler protocol π that has some additional capabilities, such as a shared memory for all machines running π. Protocols that have such additional capabilities are called *ideal functionalities*.

(a) Real Protocol   (b) Substitution with Ideal Functionality
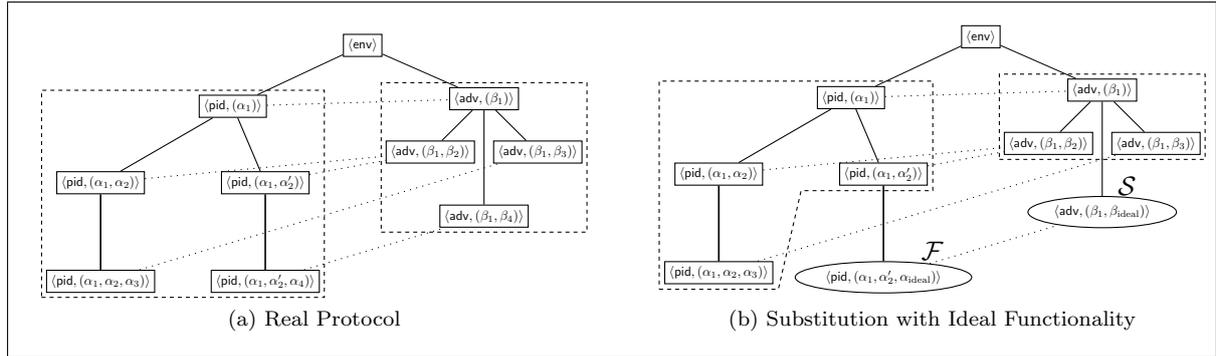
Figure 9: Substitution operation and the construction for the universal composability theorem – a sub-protocol is substituted for an ideal functionality $\mathcal{F}$ and its sub-adversary by the simulator $\mathcal{S}$ used in the universal composability proof

An ideal functionality is a protocol, i.e., every party contains a copy of the ideal functionality in its protocol tree, and all of these copies share a common state (see Section 3.1.7). Since such a copy of the 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 [28].

In contrast to previous work, we furthermore allow ideal functionalities 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.

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.

Previous work [12, 28, 39] models ideal functionalities as a single, separate machine that has a direct connection to the rest of the protocol via a so-called dummy nodes that solely forwards messages between the ideal functionality and the parent protocol.

In a time-sensitive setting, however, an ideal functionality has to abstract several machines, each of which has their own timer. It is therefore much more natural to consider distributed ideal functionalities than having a central ideal functionality: in the centralized setting the ideal functionality would have to manage the timers of each machine it replaced (each of which was in different parties in the network), as well as manage the additional delay the dummy nodes create. In the distributed setting, on the other hand, each instance of the ideal functionality is a separate machine with its own timer, allowing for a much simpler construction.

In the following, we therefore use distributed ideal functionalities, in particular for our abstraction of the onion routing protocol used in Tor (see Section 6).

3.4.1 Central vs. Distributed Ideal Functionalities

A *central ideal functionality* is a machine without parents that is a peer of so-called *dummy nodes*. Similar to previous work, a dummy node is linked to a central ideal functionality $\mathcal{F}$ (see below). Upon receiving a messages from its parent node, a dummy node forwards this message to the machine $\mathcal{F}$. Analogously, upon receiving a message from the machine $\mathcal{F}$, 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.

For the sake of convenience, we treat dummy nodes as re-wirings, i.e., reroutings. Formally, dummy nodes live at all points in time at once, we say they are *omni-time* machines. Upon receiving a message at time t they also forward the message at time t, without any delay. It would be possible use normal machines as dummy nodes, but then the central ideal functionalities would have to compensate the time the dummy nodes need for forwarding messages.

In contrast to a central 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 *distributed ideal functionality*.

We define for every central ideal functionality a corresponding distributed ideal functionality. Each replica uses the same code of the central ideal functionality but we replace each memory access with an access to the shared memory. Analogously, we define for every distributed ideal functionality a central

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.[6]

Corollary 6. *Let $\mathcal{F}_r$ be a distributed ideal functionality without children and $\mathcal{F}_c$ be the corresponding central ideal functionality. Then, $\mathcal{F}_r \geq_t \mathcal{F}_c$ and $\mathcal{F}_c \geq_t \mathcal{F}_r$.*

Proof. This lemma directly follows the internal simulation lemma (Lemma 2) and from Corollary 3, 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. □

A note on adding omni-time dummy nodes. Extending our framework with such dummy nodes that are omni-time machines, still preserves all previously proven statements. The crucial difference is in the internal simulation lemma (Lemma 2 and Corollary 5). The internal simulation is perform almost verbatim with the difference that dummy nodes are not internally executed but rather understood as re-wiring instructions: given a machine M that has a dummy nodes child that in turn in connected to an ideal functionality $\mathcal{F}$, in the internal simulation the port to the dummy node child is directly connected to the corresponding port in $\mathcal{F}$. With this construction almost by the same arguments Lemma 2 and Corollary 5 hold, which in turn implies the completeness of the dummy adversary (Lemma 3) and the composition theorem (Theorem 2).

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 secure realization adopted in TUC and show that important properties of secure realization such as the completeness of the dummy adversary and universal composability hold.

4.1 Security Definition

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. [7] 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 [11].

Definition 24 (Indistinguishability). *Two ensembles $(\mathsf{EXEC}_\eta(\Pi, \mathcal{A}, \mathrm{ENV}))_{\eta \in \mathbb{N}}$ and $(\mathsf{EXEC}_\eta(\Pi', \mathcal{A}', \mathrm{ENV}'))_{\eta \in \mathbb{N}}$ are indistinguishable, denoted*

$$\mathsf{EXEC}(\Pi, \mathcal{A}, \mathrm{ENV}) \approx \mathsf{EXEC}(\Pi', \mathcal{A}', \mathrm{ENV}'),$$

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

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

Using this definition, we can now formalize secure realization.

Definition 25. *A protocol π securely realizes another protocol ρ, written $\pi \geq_t \rho$, if for all PPT adversaries $\mathcal{A}$ there is a PPT simulator $\mathcal{S}$ such that for all PPT environments ENV*

$$\mathsf{EXEC}(\pi, \mathcal{A}, \mathrm{ENV}) \approx \mathsf{EXEC}(\rho, \mathcal{S}, \mathrm{ENV})$$

---

[6]Formally, we actually require that the states of the Turing machine of the central ideal functionality are the cartesian product S^k of the states S of the Turing machine of the distributed 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 central ideal functionality.

[7]Recall that the speed of a protocol party (along with the local time function) is determined by the protocol, more specifically the speed (and the local time function) distribution.

As the notion of secure realization is based on the notion of indistinguishability, we get as a direct consequence the transitivity and reflexivity of $\geq_t$.

Corollary 7.

$$\Pi \geq_t \Pi \ and \ \Pi_1 \geq_t \Pi_2 \wedge \Pi_2 \geq_t \Pi_3 \implies \Pi_1 \geq_t \Pi_3$$

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

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

Reflexivity can be shown similarly. $\qquad\square$

4.2   Properties of Secure Realization

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. In particular, we can view these properties as sanity checks, which attest that, with the notion of time we introduced earlier, we still get a well-formed model that allows for proving security guarantees for complex protocols.

The most important design decisions with regard to shoving these properties include making $\textsc{Env}$ and $\mathcal{A}$ timeless (see Section 3)as well as having machines run with polynomially bounded speed coefficients (see Section 3.1.3). The proofs for the following results are slightly adjusted compared to their counterparts in classic composability frameworks, such as presented in [28], in order to account for timing.

4.2.1   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 it is enough to only consider the dummy adversary $\mathcal{A}_d$, which just forwards all messages (with timing information) between environment and network parties. Furthermore, whenever $\mathcal{A}_d$ is activated without a message, it turns into the $(\mathsf{listen}, \infty)$ state and waits until it receives a message.

While the central construction for the proof are the same as in classic proofs, where the original adversary $\mathcal{A}$ is coupled with the dummy adversary $\mathcal{A}_d$ before they are split, we need to be careful with the additional delay δ the dummy adversary $\mathcal{A}_d$ induces whenever it forwards a message. We deal with this delay δ by setting it to be exponentially small, and using the fact that polynomially bounded speed coefficients of regular protocol machines also induce an at least polynomially large gap between two activations of protocol machines: This creates a tunnel in which $\mathcal{A}_d$ can delay messages without affecting the rest of the network. The rest of the Lemma then follows from Corollary 5.

Before we can show the completeness of the dummy adversary however, we require the following insight, which shows that in a network of machines with polynomial speed coefficient (i.e. make at most a polynomial number of steps per second), there is always an at least polynomial gap time-gap between two activations of machines.

Corollary 8. *Given two polynomials $A, B \in \mathbb{N}[X]$, there exists a polynomial $p \in \mathbb{N}[X]$ such that, $\forall k, l \in \mathbb{N}. \exists n_0. \forall \eta \in \mathbb{N}, \eta > n_0$. if*

$$\frac{k}{A(\eta)} - \frac{l}{B(\eta)} > 0$$

then

$$\frac{k}{A(\eta)} - \frac{l}{B(\eta)} \geq \frac{1}{p(\eta)}$$

Proof.

$$\frac{k}{A(\eta)} - \frac{l}{B(\eta)} \geq \frac{1}{p(\eta)}$$
$$\Leftrightarrow p(\eta) \geq \frac{A(\eta)B(\eta)}{kB(\eta) - lA(\eta)}$$

Since both $A(\eta)$ and $B(\eta)$ are natural valued, we know that for all $\eta > n_0$ we have $1 \leq kB(\eta) - lA(\eta)$. Thus, for all $\eta > n_0$

$$\frac{A(\eta)B(\eta)}{1} \geq \frac{A(\eta)B(\eta)}{kB(\eta) - lA(\eta)}$$

Choosing $p(\eta) := A(\eta)B(\eta)$ proves the claim. $\qquad\square$

Given Corollary 8 we can now proof the completeness of the dummy adversary.

Construction of the dummy adversary. The dummy adversary merely forwards messages from the protocol to the adversary and vice versa.

$\mathcal{A}_d$: upon a message m from a protocol Π

    request time from EXEC; wait for result $t_{\mathcal{A}_d}$
    send m to ENV at time $t + 1/2^\eta$
    listen until activated

$\mathcal{A}_d$: upon a message (m, P, t) from the environment ENV

    send m to P at time t
    listen until activated

Lemma 3 (Completeness of the dummy adversary). *If there exists an adversary $\mathcal{S}$ for a protocol Π such that for all environments* ENV,

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

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

Proof. Take any adversary $\mathcal{A}$ for Π. We substitute $\mathcal{A}$ with the dummy adversary $\mathcal{A}_d$, and put $\mathcal{A}$ together with ENV into the new environment ENV', which emulates the execution for ENV and $\mathcal{A}$.

The internal simulation of ENV and $\mathcal{A}$ works as in the proof of the internal simulation lemma (Lemma 2). Since ENV' is timeless, it has to decide how to proceed in time. The timer of ENV' always matches the minimum of the timers of ENV and $\mathcal{A}$, and ENV' is in the listen state if the machine with the lowest timer is in the listen state as well. Whenever ENV' is activated without a message, ENV internally activates the machine with the lowest timer (independent of state) and then continues to activate the machine with the lowest timer until this machine is in the listen state at time T. During this process, ENV' adds the factor δ to $T_{\mathcal{A}}$ whenever it evaluates which timer is lower, where $\delta = 1/2^\eta$ is the time the dummy adversary progresses in time on each activation, e.g., whenever ENV' decides in which . This off-set is necessary in order to correctly synchronize the dummy adversary and $\mathcal{A}$. As a consequence, whenever ENV' returns control to EXEC, we have that $T_{\mathrm{ENV}'} = min\{T_{\mathrm{ENV}}, T_{\mathcal{A}} + \delta\}$.

Whenever $\mathcal{A}$ in ENV' sends a message m to a protocol party P at time t, ENV' instructs the dummy adversary $\mathcal{A}_d$ to send the message at time $t' := \max\{T'_{\mathrm{ENV}} + \delta, t - \delta\}$ (with the message (m, P, t')).

It remains to show that this scenario is indistinguishable for the original network adversary $\mathcal{A}$, the original environment ENV, and the protocol Π.

Claim 1. *Let $\mathsf{EXEC}_{\mathcal{A},\eta}$ be defined as EXEC_η except that it outputs the sequence of internal states (i.e., the contents of all tapes of all machines) of $\mathcal{A}$ during the execution. Then, $\mathsf{EXEC}_{\mathcal{A},\eta}(\Pi, \mathcal{A}, \mathrm{ENV})$ is indistinguishable from $\mathsf{EXEC}_{\mathcal{A},\eta}(\Pi, \mathcal{A}_d, \mathrm{ENV}')$.*

Proof. Recall that the dummy adversary is activated at time T when a message arrives from the protocol Π. Thus, the environment ENV' receives the message at time $T + \delta$. Since ENV' keeps in its internal simulation a δ-gap to the timer of $\mathcal{A}$ and then directly forwards the messages to $\mathcal{A}$, $\mathcal{A}$ thinks it receives the messages at time T. $\qquad\square$

Claim 2. *Let $\mathsf{EXEC}_{\mathrm{ENV},\eta}$ be defined as EXEC_η except that it outputs the sequence of internal states (i.e., the contents of all tapes) of ENV during the execution. Then, $\mathsf{EXEC}_{\mathrm{ENV},\eta}(\Pi, \mathcal{A}, \mathrm{ENV})$ is indistinguishable from $\mathsf{EXEC}_{\mathrm{ENV},\eta}(\Pi, \mathcal{A}_d, \mathrm{ENV}')$.*

Proof. Messages from subroutine ports of regular machines are forwarded to ENV with the current time of ENV' as (the internally simulated) time-stamp. Since $T_{\mathrm{ENV}} \geq T_{\mathrm{ENV}'}$, these messages timely arrive at ENV. $\qquad\square$

Claim 3. *Let* $\mathsf{EXEC}_{\Pi,\eta}$ *be defined as* EXEC_{η} *except that it outputs the sequence of internal states (i.e., the contents of all tapes of all machines) of* Π *during the execution. Then,* $\mathsf{EXEC}_{\Pi,\eta}(\Pi, \mathcal{A}, \mathrm{ENV})$ *is indistinguishable from* $\mathsf{EXEC}_{\Pi,\eta}(\Pi, \mathcal{A}_d, \mathrm{ENV}')$.

Proof. It suffices to show the following two properties: (i) messages from the environment are indistinguishable; (ii) messages from the network are indistinguishable.

By the same arguments as in the proof of Lemma 2, messages from the environment are indistinguishable since ENV' perfectly simulates ENV.

Next, we show that all messages from the dummy adversary $\mathcal{A}_d$ arrive at a time T' that is indistinguishable from the time T at which the network adversary $\mathcal{A}$ sends the messages. Recall that the time gap of that the dummy adversary for sending a message is $\delta = 1/2^\eta$.

We show by induction the ith message $\tilde{m}$ received by the dummy adversary $\mathcal{A}_d$ that $\mathcal{A}_d$ forwards a message from the adversary $\mathcal{A}$ either

1.) at the time $T_{\mathcal{A}} = T_{\mathcal{A}_d}$ or

2.) at a time $T_{\mathcal{A}_d}$ such that $T_\pi + k/2^\eta \geq T_{\mathcal{A}_d} \geq T_{\mathcal{A}} = T_\pi + \delta'$ where δ' is some time induced by $\mathcal{A}$, k is polynomially bounded, and T_π is a protocol time, i.e., a time in which some protocol party can be.

In the base case, $\mathcal{A}_d$ is by construction in time 0 and listens until it receives a message. Observe that the dummy adversary only receives messages from the adversary $\mathcal{A}$ and not from ENV. Thus, the environment ENV' sends the message from the $\mathcal{A}$ at time $\max\{\delta, T_{\mathcal{A}} - \delta\}$. Since the speed coefficient of regular machines are polynomials (see Section 3.1.3), we know by Corollary 8 that the gap between two protocol times is at least $1/p(\eta)$, for some polynomial p. Hence, any protocol machine that is in a listen state will receive messages up to a time $\geq 1/p(\eta) > \max\{\delta, T_{\mathcal{A}} - \delta\}$.

In the case that the dummy adversary $\mathcal{A}_d$ already received previous messages, we distinguish two cases. First, the last message m that $\mathcal{A}_d$ received came from the protocol Π. Second, the last message m that $\mathcal{A}_d$ received came from $\mathcal{A}$ (via ENV').

In the first case, let $\tilde{T}_{\mathcal{A}_d}$ be the time at which $\mathcal{A}_d$ received the message m from the protocol. By construction, $\mathcal{A}_d$ immediately forwarded the message, hence sent it to $\mathcal{A}$ (via ENV') at time $T_\pi + \delta = T_\pi + 1/2^\eta$. By construction of ENV', $\mathcal{A}_d$ receives the response $\tilde{m}$ of $\mathcal{A}$ at time $\tilde{T}_{\mathcal{A}_d} + \delta$ and sends it at time $\tilde{T}_{\mathcal{A}_d} + 2 \cdot \delta$.

Assume we have by induction hypothesis that for the last message m' (at some time before m) from $\mathcal{A}$ we have $T_\pi + k/2^\eta \geq T'_{\mathcal{A}_d} \geq T'_{\mathcal{A}} = T_\pi + \delta'$ where T_π is some protocol time and δ' is some gap that is induced by $\mathcal{A}$. Hence, $\mathcal{A}_d$ was able to receive the message m from the protocol at a time $\tilde{T}_{\mathcal{A}_d} = T_{\mathcal{A}_d}$ such that $T_\pi + k/2^\eta \geq T'_{\mathcal{A}_d} \geq T_{\mathcal{A}} = T_\pi + \delta'$. Thus, we get that $\mathcal{A}_d$ forwards the response $\tilde{m}$ of $\mathcal{A}$ at time $T_\pi + k + 2 \cdot \delta = T_\pi + k + 2/2^\eta$.

If we have by induction hypothesis that $T'_{\mathcal{A}} = T'_{\mathcal{A}_d}$, then the message is sent at time $T'_{\mathcal{A}_d} + 2 \cdot \delta = T_{\mathcal{A}_d} + 2 \cdot \delta$, which satisfies the statement.

In the second case, let $\tilde{T}_{\mathcal{A}_d}$ be the time at which $\mathcal{A}_d$ received m. Let m' be the $i - 1$th message and $T'_{\mathcal{A}_d}$ be the time-stamp of $\mathcal{A}_d$ for sending m' and $T'_{\mathcal{A}}$ be the time-stamp of $\mathcal{A}$ for sending m'. If we have by induction hypothesis that $T'_{\mathcal{A}} = T'_{\mathcal{A}_d}$ for the time-stamp of m', we again distinguish two cases: first, $\mathcal{A}$ sent the message $\tilde{m}$ at a time T such that $T \geq \tilde{T}_{\mathcal{A}} + \delta$; second, $\mathcal{A}$ sent the message $\tilde{m}$ at a time T such that $T < \tilde{T}_{\mathcal{A}} + \delta$. In the first case, $\mathcal{A}_d$ sends $\tilde{m}$ at time T and we have $T_{\mathcal{A}} = T_{\mathcal{A}_d}$. In the second case, $\mathcal{A}_d$ sends $\tilde{m}$ at time T and we have $T_{\mathcal{A}} = T_{\mathcal{A}_d}$.

If we have by induction hypothesis that for m' we have $T_\pi + k/2^\eta \geq T'_{\mathcal{A}_d} \geq T'_{\mathcal{A}} = T_\pi + \delta'$ where T_π is some protocol time, then by the same argumentation as before we get two cases: (i) if $\mathcal{A}$ sent the message $\tilde{m}$ at a time T such that $T \geq \tilde{T}_{\mathcal{A}} + k \cdot \delta = \tilde{T}_{\mathcal{A}} + k + 1/2^\eta$, then $T_{\mathcal{A}} = T_{\mathcal{A}_d}$; if $\mathcal{A}$ sent the message $\tilde{m}$ at a time T such that $T < \tilde{T}_{\mathcal{A}} + \delta$, then $T_\pi + k + 1/2^\eta \geq T'_{\mathcal{A}_d} \geq T'_{\mathcal{A}} = T_\pi + \delta'$ holds. $\diamond$

By the statement above, we know that either $\mathcal{A}_d$ sends the messages at the same time as $\mathcal{A}$, or $T_\pi + k/2^\eta \geq T_{\mathcal{A}_d} \geq T_{\mathcal{A}} = T_\pi + \delta'$ for some polynomially bounded k. In the second case, we use Corollary 8 implies that for any pair of protocol times T_p and T'_p the probability that $|T_\pi - T'_\pi| > 1/p(\eta)$ for any polynomial is negligible in η. Hence, upon a listen command, any machine in the protocol Π receives the same messages in $\mathsf{EXEC}_{\Pi,\eta}(\Pi, \mathcal{A}, \mathrm{ENV})$ and $\mathsf{EXEC}_{\Pi,\eta}(\Pi, \mathcal{A}_d, \mathrm{ENV}')$. $\square$

As ENV' generates the same output as ENV and does not generate any additional delays, both situations with ENV' and ENV generate the same behavior.

Finally, we are in a situation with the environment ENV', the dummy adversary $\mathcal{A}_d$ and the protocol Π. By assumption we can replace Π with Π' and $\mathcal{A}_d$ with $\mathcal{S}$ and remain indistinguishable.

Reverting $\textsc{Env}'$ to $\textsc{Env}$ and $\mathcal{A}'$ and combining $\mathcal{S}$ with $\mathcal{A}'$ to a new simulator $\mathcal{S}'$ then gives us the theorem. Note that in this step we can again remove all methods in $\textsc{Env}'$ we used to remove the time gap between the dummy adversary $\mathcal{A}_d$ and the regular adversary $\mathcal{A}$ since both $\mathcal{S}$ and $\mathcal{A}$ now reside in the same machine. $\mathcal{S}'$ therefore only has to translate all messages from $\mathcal{S}$ to the environment to network messages to $\mathcal{A}$. □

4.2.2 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 9.

Theorem 2. *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* $\Pi'_1 \geq_t \Pi'$. *Then*

$$\Pi[\Pi' \setminus \Pi'_1] \geq_t \Pi.$$

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.
Game 1: Original network consisting of parties running protocol Π, environment $\textsc{Env}$ and the dummy adversary $\mathcal{A}_d$.
Game 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 intercepted messages from Π' go directly to $\mathcal{A}'_d$.

As the network is otherwise the same as in Game 1, Game 2 and Game 1 are both indistinguishable.
Game 3: We define a new environment $\textsc{Env}'$ which internally simulates $\textsc{Env}$, 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 $\mathcal{A}_d \setminus \mathcal{A}'_d$.

By Lemma: 2, $\textsc{Env}'$ can simulate the interaction between $\textsc{Env}$, $\Pi \setminus \Pi'$ and $\mathcal{A}_d \setminus \mathcal{A}'_d$ as in the regular network, without introducing additional delays.
Game 4: In Game 3 we have the situation where parties running protocol Π' communicate with the corresponding dummy adversary $\mathcal{A}'_d$ and the environment $\textsc{Env}'$. 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 Game 3.
Game 5: We now split $\textsc{Env}'$ 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 $\textsc{Env}'$ simulated, Game 5 is indistinguishable from Game 4.

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

4.2.3 Joint State Theorem

A Joint State Theorem [13, 28] 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 communication model. While the proof is along the lines of the proof in [28, Section 9], we have to carefully consider the timing information the adversary has access to in our communication model (as for the composition theorem).

Multi-session functionality. In order to analyze multi-session protocols with a joint-state, we 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 [28]. Consistency enforcing scheduling (Section 3.1.5)
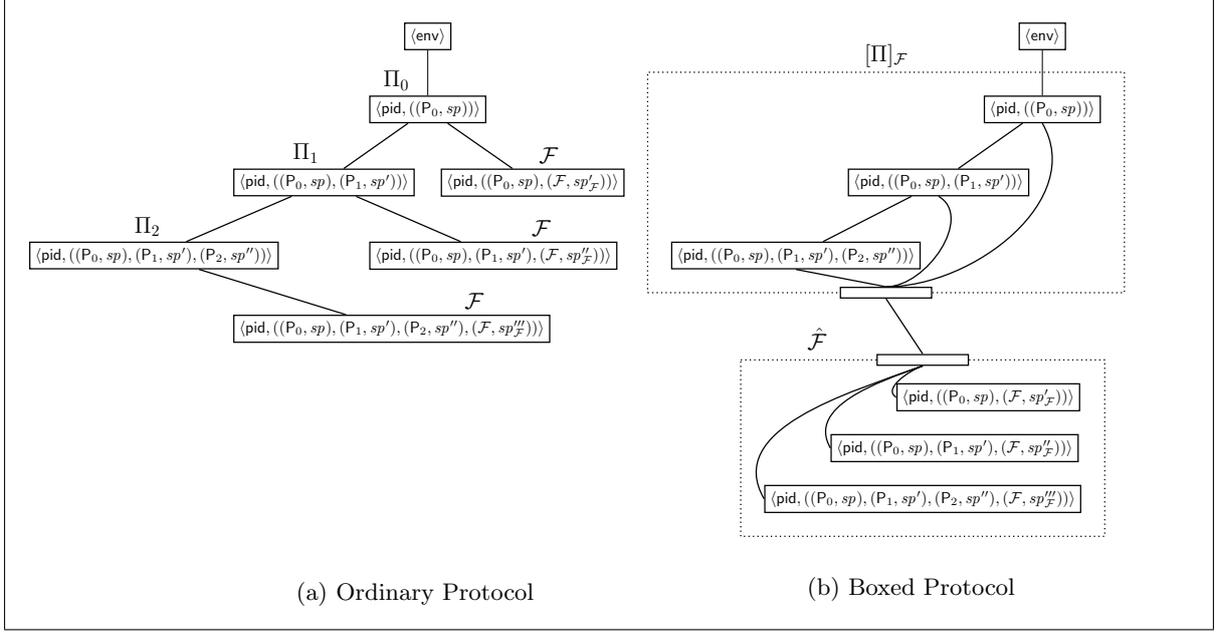
Figure 10: Boxed Protocol $[\Pi]_{\mathcal{F}}$ and Multi-Session Functionality $\hat{\mathcal{F}}$

allows us to also do this in our time-sensitive communication model as well (see Lemma 2). Thus we can adopt a similar construction, only amending the speed coefficient of the multi-session machine to also account for session ID translation.

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, above construction is not compatible with our construction for a multi-session machine developed in Section 3.2.2, which only allows for a single environmental port. Furthermore, the single caller rule that we adopt from GNUC [28] is no longer 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 [28]: 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: 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 $sid_2 = sid$. 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 TM M 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 10 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.

With this construction we can now formulate the Joint State Composition Theorem.

Theorem 3. *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 need to construct a simulator $\mathcal{S}$ such that following holds:

$$\mathsf{EXEC}(\Pi, \mathcal{S}, \mathrm{ENV}) \approx \mathsf{EXEC}([\Pi]_{\mathcal{F}}, \mathcal{A}_d, \mathrm{ENV})$$

As shown in Lemma 2, both machines, the multi-sessions functionality $\hat{\mathcal{F}}$ and the boxed protocol $[\Pi]_{\mathcal{F}}$ simulate their internal machines as in the regular network. We only have to add additional computation time for the translation to and from virtual sessions IDs between $[\Pi]_{\mathcal{F}}$ and $\hat{\mathcal{F}}$. Since this translation is possible in polynomial time we can achieve this by simply accelerating the simulating machines by the appropriate factor.

The only exception that remains is $\mathcal{S}$ having to handle irregular virtual session IDs which could be sent to $\hat{\mathcal{F}}$ through a corrupted $[\Pi]_{\mathcal{F}}$: Π-impossible virtual session IDs have a basename which specify an instance of $\mathcal{F}$, but have a session ID prefix which is invalid with regard to Π. $\hat{\mathcal{F}}$ cannot distinguish such session IDs from regular ones, hence $\mathcal{S}$ needs to catch them and return an error to ENV. $\mathcal{S}$ cannot do so directly however, but needs to simulate a run with a message from ENV to the network containing such a Π-impossible sessions ID in order to find the right time to send the response to ENV. But as $\mathcal{S}$ is timeless and the simulation of the network is possible in poly-time, $\mathcal{S}$ can do so without a problem. $\square$

This concludes the introduction of secure realization in TUC and the presentation of its properties. The next sections show how to formalize the onion routing protocol that underlies Tor [53] in TUC and how to prove this formalization secure in TUC.

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 [53]. 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 [52].

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 $\mathcal{F}_{\mathrm{OR}}$. Finally, Section 5.3 shows that Π_{OR} securely realizes $\mathcal{F}_{\mathrm{OR}}$, i.e., we show that the abstraction $\mathcal{F}_{\mathrm{OR}}$ is sound in the sense that every attack against Π_{OR} can also be mounted against $\mathcal{F}_{\mathrm{OR}}$.

Considering a time-sensitive adversary imposes new challenges on the analysis of complex, cryptographic communication protocols. As in previous work [2], we 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, however, 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) [24].[8] 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.[9]

---

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

[9]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.

upon (setup) from the parent node:

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

upon (createcircuit, $\mathcal{P} = \langle P, \langle P_j \rangle_{j=1}^\ell \rangle$) from the parent node:

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

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

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

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

1: **if** $(P, cid, \text{relay}, (\text{data}, m)) = lookup(h)$
   and P is the OP for cid **then**
2:    **if** $prev(cid) = \bot$ **then**
3:       **if** $getkey(cid) = (k_j)_{j=1}^{\ell'}$ **then**
4:          $O \leftarrow UnwrOn(O, (k_j)_{j=1}^{\ell'})$
5:          **if** $O = (\text{data}, m)$ **then**
6:             output (received, cid, m)

Figure 11: Π_{OR}: Client Protocol for Machine 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 11, 13 and 12. Π_{OR} closely follows the Tor specification [17] 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 *client protocol*.

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.[10] 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., $next(cid_i)$ returns (P_{i+1}, cid_{i+1}) and $next(cid_{i+1})$ returns $\bot$.

---

[10] Tor currently uses the TAP protocol and is going to switch to the more efficient and secure `ntor` protocol. [24]

28

```
ExtendCircuit(𝒫 = ⟨P_j⟩_{j=1}^ℓ, 𝒞 = ⟨P ⇔^{cid₁,k₁} P₁ ⇔^{k₂}
··· P_{ℓ'}⟩):
  1: determine the next node P_{ℓ'+1} from 𝒫 and 𝒞
  2: if P_{ℓ'+1} = ⊥ then
  3:    output (created, ⟨P ⇔^{cid₁} P₁ ⟺ ··· P_{ℓ'}⟩)
  4: else
  5:    X ← Initiate(pk_{P_{ℓ'+1}}, P_{ℓ'+1})
  6:    if P_{ℓ'+1} = P₁ then
  7:       cid₁←{0,1}^κ
  8:       send a cell (cid₁, create, X) to P₁ over 𝓕_scs
  9:    else
 10:       O ← WrOn({extend, P_{ℓ'+1}, X}, (k_j)_{j=1}^{ℓ'})
 11:       send a cell (cid₁, relay, O) to P₁ over 𝓕_scs

DestroyCircuit(𝒞, cid):
  1: if next(cid) = (P_next, cid_next) then
  2:    send a cell (cid_next, destroy) to P_next over 𝓕_scs
  3: else if prev(cid) = (P_prev, cid_prev) then
  4:    send a cell (cid_prev, destroy) to P_prev over 𝓕_scs
  5: discard 𝒞 and all streams
```

Figure 12: Subroutines of Π_OR for Party P

In the next section we go into detail about the different messages exchanged in Π_OR.

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 $\mathcal{F}_{REG}^{\mathcal{N}}$, which is defined as in [11] with the exception that $\mathcal{F}_{REG}^{\mathcal{N}}$ rejects all parties not in $\mathcal{N}$ and only distributes the public keys after all parties in $\mathcal{N}$ have registered with $\mathcal{F}_{REG}^{\mathcal{N}}$. As soon as all parties have registered, each of them receives the message $(registered, \langle P_j, pk_j \rangle_{j=1}^n)$, which contains a list of all valid OR nodes, together with their public keys.

Circuit creation. Upon an input createcircuit (see Figure 11), 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 12). 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).

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 $\mathcal{F}_{scs}$ as proposed by Canetti [11].[11]

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.

---

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

upon receiving an input (exit, sid, m) from the parent
1: obtain $\mathcal{C} = \langle P' \stackrel{cid,k}{\Longleftrightarrow} P \rangle$ for sid
2: $O \leftarrow WrOn(m,k)$
3: send a cell (cid relay, O) to P' over $\mathcal{F}_{\text{SCS}}$

upon receiving a cell (cid, create, X) from P' over $\mathcal{F}_{\text{SCS}}$:
1: $\langle Y, k_{\text{new}} \rangle \leftarrow Respond(pk_P, sk_P, X)$
2: store $\mathcal{C} \leftarrow \langle P' \stackrel{cid,k_{\text{new}}}{\Longleftrightarrow} P \rangle$
3: send a cell (cid, created, Y, t) to P' over $\mathcal{F}_{\text{SCS}}$

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

upon receiving a cell (cid, destroy) from P' over $\mathcal{F}_{\text{SCS}}$:
1: call $DestroyCircuit(\mathcal{C}, cid)$

upon receiving a cell (cid, relay, O) from P' over $\mathcal{F}_{\text{SCS}}$:
1: **if** $prev(cid) = \bot$ **then**
2: **if** $getkey(cid) = (k_j)_{j=1}^{\ell'}$ **then**
3: $O \leftarrow UnwrOn(O, (k_j)_{j=1}^{\ell'})$
4: **if** $(\text{type}, m) \neq O$ **then** abort

5: **else if** $prev(cid) = (P'', cid', k')$ **then**
6: /* a backward onion */
7: $O \leftarrow WrOn(O, k')$; type ← default
8: **switch** (type)
9: **case** extend:
10: $(P_{next}, X) \leftarrow m$; $cid_{next} \leftarrow \{0,1\}^{\kappa}$
11: update $\mathcal{C} \leftarrow \langle P' \stackrel{cid,k}{\Longleftrightarrow} P \stackrel{cid_{next}}{\Longleftrightarrow} P_{next} \rangle$
12: send a cell (cid_{next}, create, X) to P_{next} over $\mathcal{F}_{\text{SCS}}$

13: **case** extended:
14: get $\langle Y, t \rangle$ from m
15: let P_{ex} be the first party in $\mathcal{P}$ without a key in $\mathcal{C}$
16: $k_{\text{ex}} \leftarrow ComputeKey(pk_{\text{ex}}, Y, t)$
17: update $\mathcal{C}$ with (k_{ex}); call $ExtendCircuit(\mathcal{P}, \mathcal{C})$
18: **case** data:
19: **if** P is the OP for cid **then** output (received, cid, m)
20: **else if** $m = (P'', m')$
21: // P is the exit node for cid
22: generate or lookup the unique sid for cid
23: output (exit, (P'', (sid, m'))) to parent
24: **case** corrupted: /*corrupted onion*/
25: call $DestroyCircuit(\mathcal{C}, cid)$
26: **case** default: /*encrypted forward/backward onion*/
27: **if** $prev(cid) = \bot$ **then** $(P'', cid') = next(cid)$

28: send a cell (cid', relay, O) to P'' over $\mathcal{F}_{\text{SCS}}$

Figure 13: Π_{OR} for Machine P: Network Messages for an Onion Router.

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 contraset to previous work, an exit node does not send the message over the network but rather outputs the message to the environment with a exit string as prefix to mark the message as an exit message. This exit messages are important for being able to apply the countermeasure.

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 12). 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.

A destroy cell is also in that situation sent through the circuit if 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 secure realization 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

```
upon input (setup):                              upon receiving a handle (P', P, h) from the net-
  1: draw a fresh handle h; set registered_flag ← true   work
                                                    1: if (P, cid, relay, (data, m)) = lookup(h)
  2: store lookup(h) ← (dir, registered, N)              and P is the OP for cid then
  3: lookup the current time t; let t' ← v[1]      2:    lookup the current t; let t' ← v[6]
  4: send (h, register, P) to the network at time t + t'   3:    if prev(cid) = ⊥ then
                                                    4:       output (received, cid, m) at time t + t'
  5: wait for a msg (dir, registered, N) via a handle
  6: lookup the current time t                     upon input (send, C = ⟨P ⟺^{cid_1} P_1 ⟺ ··· P_ℓ⟩, m)
  7: let t' ← v[2]                                  1: if current time − Start(cid_1) ≤ ttl_C then
  8: output (ready, (P_j)_{j=1}^n) = (ready, N) at time t+t'   2:    let t ← v[4]
                                                    3:    SendMessage(P_1, cid_1, relay, (data, m), t)
  upon input (createcircuit, P = ⟨P, P_1, ..., P_ℓ⟩)   4: else
  1: set Start to the current time                 5:    DestroyCircuit(C, cid_1)
  2: store P and C ← ⟨P⟩                            6:    lookup the current time t
  3: let t ← v[3]                                   7:    let t' ← v[5]
  4: ExtendCircuit(P, C, t)                         8:    output (destroyed, C, m) at time t + t'
```

Figure 14: The ideal functionality $\mathcal{F}_{OR}^{\mathcal{N}}$ (short $\mathcal{F}_{OR}$) for Machine P: Client

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 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 3 to compensate for these differences.

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

The ideal functionality $\mathcal{F}_{OR}$, as presented in Figure 14, 15, and 16, is close to the OR protocol Π_{OR} presented in Section 5.1. Due to the similarities of Π_{OR} and $\mathcal{F}_{OR}$, we concentrate on highlighting the differences between them.

The major difference between Π_{OR} and $\mathcal{F}_{OR}$ is that $\mathcal{F}_{OR}$ 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 $\mathcal{F}_{OR}$.

In fact, $\mathcal{F}_{OR}$ 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 $\mathcal{F}_{OR}$, which allows all parties running $\mathcal{F}_{OR}$ 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 $⟨P, P_{next}, h⟩$ over the network.

$\mathcal{F}_{OR}$ also does not require $\mathcal{F}_{REG}^{\mathcal{N}}$ 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.

Compromising parties. A party running $\mathcal{F}_{OR}$ 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 secure realization for $\mathcal{F}_{OR}$, 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 realization proof in [2] does not

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

1: set *compromised* $\leftarrow$ *true*; delete local information

upon receiving $(\_, P, h, [\mathsf{corrupt}, T(\cdot)])$ from the network:

1: $msg \leftarrow lookup(h)$
2: **if** corrupt $= true$ **then**
3:   $msg \leftarrow T(msg)$
4:   set $corrupted(msg) \leftarrow true$
5: lookup the current time t; let $t' \leftarrow v[11]$
6: proceed with msg at time $t + t'$

upon receiving an input $(\mathsf{response}, \mathsf{sid}, m)$ from the parent

1: obtain $\mathcal{C} = \langle P' \stackrel{cid}{\Longleftrightarrow} P \rangle$ for sid
2: lookup the current time t; let $t' \leftarrow v[15]$
3: $SendMessage(P', cid, \mathsf{relay}, (\mathsf{data}, m), t + t')$

upon receiving a handle $(\_, P, h)$ from the network:

1: lookup the current time t; let $t' \leftarrow v[6]$
2: proceed with $msg \leftarrow lookup(h)$ at time $t + t'$

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

1: store $\mathcal{C} \leftarrow \langle P' \stackrel{cid}{\Longleftrightarrow} P \rangle$
2: let $t \leftarrow v[7]$
3: $SendMessage(P', cid, \mathsf{created}, t)$

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

1: **if** $prev(cid) = (P'', cid')$ **then**
2:   let $t \leftarrow v[8]$
3:   $SendMessage(P'', cid', \mathsf{relay}, \mathsf{extended}, t)$
4: **else if** $prev(cid) = \bot$ **then**
5:   let $t \leftarrow v[9]$
6:   $ExtendCircuit(\mathcal{P}, \mathcal{C}, t)$

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

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

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

1: **if** $prev(cid) = \bot$ **then** $(type, m) \leftarrow O$
2: **else** $(P'', cid') \leftarrow prev(cid)$; $type \leftarrow \mathsf{default}$
3: **switch** $(type)$
4: **case** extend:
5:   $P_{next} \leftarrow m$; $cid_{next} \leftarrow \{0, 1\}^\kappa$
6:   update $\mathcal{C} \leftarrow \langle P' \stackrel{cid}{\Longleftrightarrow} P \stackrel{cid_{next}}{\Longleftrightarrow} P_{next} \rangle$
7:   let $t \leftarrow v[10]$
8:   $SendMessage(P_{next}, cid_{next}, \mathsf{create}, t)$
9: **case** extended:
10:   update $\mathcal{C}$ with P_{ex}
11:   $ExtendCircuit(\mathcal{P}, \mathcal{C})$
12: **case** data:
13:   **if** $(P = OP)$ **then** output $(\mathsf{received}, cid, m)$
14:   **else if** $m = (P'', m')$
15:     generate or lookup the unique sid for cid
16:     output $(\mathsf{exit}, (P'', (\mathsf{sid}, m')))$ to parent
17: **case** corrupt: /*corrupted onion*/
18:   let $t \leftarrow v[12]$
19:   $DestroyCircuit(\mathcal{C}, cid, t)$
20: **case** default: /*encrypted forward/backward onion*/
21:   **if** $prev(cid) = \bot$ **then** $(P'', cid') = next(cid)$

22:   let $t \leftarrow v[13]$
23:   $SendMessage(P'', cid', \mathsf{relay}, O, t)$

Figure 15: The ideal functionality $\mathcal{F}_{\mathrm{OR}}^{\mathcal{N}}$ (short $\mathcal{F}_{\mathrm{OR}}$) for Machine P: Network messages for an onion router

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 16. Parties running $\mathcal{F}_{\mathrm{OR}}$ share their *compromised*-status over the shared memory and based on this leak the required information to the adversary.

Messages through a handle. Figure 15 considers messages m that are retrieved *through a handle*. As described above, $\mathcal{F}_{\mathrm{OR}}$ 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 $\mathcal{F}_{\mathrm{OR}}$ 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.

```
ExtendCircuit(𝒫 = (P_j)_{j=1}^ℓ, 𝒞 = ⟨P ⟺^{cid_1} P_1 ⟺        SendMessage(P_next, cid_next, cmd[, relay-type][, data]
··· P_ℓ'⟩[, time]):                                          [, time]):
  1: determine the next node P_{ℓ'+1} from 𝒫 and 𝒞            1: draw a fresh handle h
  2: if P_{ℓ'+1} = ⊥ then                                     2: set    lookup(h)    ←    (P_next, cid, cmd[, relay-
  3:    output (created, 𝒞)                                         type][, data])
  4: else                                                     3: if compromised = true then
  5: if P_{ℓ'+1} = P_1 then                                   4:    let P_last be the last node in the complete con-
  6:    cid_1 ← {0,1}^κ                                             tiguous visible subpath path starting P_next
  7:    SendMessage(P_1, cid_1, create[, time])               5:    if (P_last = OP) or P_last is the exit node and
  8: else                                                           data ≠ ⊥ then
  9:    SendMessage(P_1, cid_1, relay, (extend, P_{ℓ'+1})[, time]) 6:    msg' ← lookup(h')
                                                              7:       lookup the current time t; send the entire
DestroyCircuit(𝒞, cid[, time]):                                      message ⟨P, P_next, ..., P_last, cid_next, cmd, data⟩
  1: if next(cid) = (P_next, cid_next) then                          to 𝒜 at time t[+time]
  2:    SendMessage(P_next, cid_next, destroy[, time])        8:    else send ⟨P, P_next, ..., P_last, cid_next, cmd, h⟩
  3: else if prev(cid) = (P_prev, cid_prev) then                     to 𝒜
  4:    SendMessage(P_prev, cid_prev, destroy[, time])        9: else send ⟨P, P_next, h⟩ to the network
  5: discard 𝒞 and all streams
```

Figure 16: Subroutines of $\mathcal{F}_{OR}$ for Party P

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

In order to correctly capture the notion of time in our abstraction, we modify some aspects of $\mathcal{F}_{OR}$ 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 $\mathcal{F}_{OR}$ 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 secure realization for our abstraction, we therefore need to adjust the pace in which parties running $\mathcal{F}_{OR}$ advance in time.

We achieve this by introducing a *delay vector* $v = (d_1, \ldots, d_{15})$ with which we parameterize $\mathcal{F}_{OR}$. Each entry of v is a delay-distribution, which is inserted at specific points in the code of $\mathcal{F}_{OR}$ (see Figure 14). $\mathcal{F}_{OR}$ 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 $\mathcal{F}_{OR}$ 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 $\mathcal{F}_{OR}$. We explain this in more detail in Section 5.3.

In Section 3.1.3 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 $\mathcal{F}_{OR}$ 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.

We stress that $\mathcal{F}_{OR}$ basically resembles the protocol except for the cryptographic operations. Instead of ciphertexts and group elements, the $\mathcal{F}_{OR}$ merely sends freshly drawn handles over the network. The

predecessor of this $\mathcal{F}_{\mathrm{OR}}$ 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 $\mathcal{F}_{\mathrm{OR}}$ is indeed an accurate abstraction of the onion routing protocol Π_{OR}: we show that Π_{OR} securely realizes $\mathcal{F}_{\mathrm{OR}}$ in TUC, which was already shown by Backes et al. [2] for the standard UC-framework. This gives us the secure realization for the abstraction.

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 [24]. 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.

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 26 (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.

```
(setup, ℓ')                                    (unwrap, O, cid)
   if initiated = false then                       look up the key k for cid
      for i = 1 to ℓ' do                            O' ← UnwrOn(O, k)
         k_i←{0,1}^η; cid_i←{0,1}^η                 send O'
      initiated ← true; store ℓ'
      send cid                                  (respond, m)
                                                   O ← WrOn(m, k_{ℓ'})
(compromise, i)                                    send O
   initiated ← false; erase the circuit
   compromised(i) ← true; run setup;          (wrap, O, cid)
   for j with compromised(j) = true do            look up the key k for cid
      send (cid_j, k_j) for all                    O' ← WrOn(O, k)
                                                    send O'
(send, m)
   O ← WrOn(m, ⟨k_i⟩_{i=1}^{ℓ'})              (destruct, O)
   send O                                         m ← UnwrOn(O, ⟨k_i⟩_{i=1}^{ℓ'})
                                                   send m
```

Figure 17: The Honest Onion Secrecy Challenger $\mathsf{OS\text{-}Ch}^0$: $\mathsf{OS\text{-}Ch}^0$ only answers for honest parties

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 with the output of the adversary 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 27 (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 adversary. Let the last onion $O_{ℓ'}$ of an onion O_1 be defined as follows:*

$$
\begin{aligned}
&\mathsf{Last}(O_1)\text{:}\\
&\quad \boldsymbol{for}\ i = 1\ to\ ℓ' - 1\ \boldsymbol{do}\\
&\qquad O_{i+1} ← UnwrOn(O_i)
\end{aligned}
$$

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 adversaries $\mathcal{A}$ the following is negligible in the security parameter η*

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

End-to-end integrity. The third property that we require is *end-to-end integrity*, i.e., the adversary 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 adversary, 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, $(a, s) ← A^B$ denotes in the following definition the pair of the outputs of A and B.

Definition 28 (Synchronicity). *For a machine $\mathcal{A}$, let $\Omega_{l,\mathcal{A}}$ and $\Omega_{r,\mathcal{A}}$ be defined as follows:*

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

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

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

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

Figure 18: The Faking Onion Secrecy Challenger $\mathsf{OS\text{-}Ch}^1$: $\mathsf{OS\text{-}Ch}^1$ only answers for honest parties. st^i_f, st^i_b is the current forward, respectively backward, state of party i. $((O_i)^j_{i=u-1}, s') \leftarrow WrOn^{j-u+1}(m, \langle k_i \rangle^j_{i=u}, st)$ is defined as $O_{u-1} \leftarrow m$; $\textbf{for } i = u \textbf{ to } j \textbf{ do } (O_i, s') \leftarrow WrOn(O_{i-1}, k_{j+u-i}, st)$

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 [50].

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 adversary does not control the final node.

For $\mathsf{OS\text{-}Ch}^1$, we define the maximal paths that the adversary knows from the circuit. A visible subpath of a circuit $(P_i, k_i, cid_i)^\ell_{i=1}$ from an honest onion proxy is a minimal subsequence of corrupted parties $(P_i)^s_{i=u}$ of $(P_i)^\ell_{i=1}$ 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 $(P_i)^s_{i=u}$. We store visible subpaths by the first $cid = cid_u$.

Figure 17 and 18 presents $\mathsf{OS\text{-}Ch}^0$, and $\mathsf{OS\text{-}Ch}^1$, respectively. [12]

Definition 29 (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

---

[12] We stress that in Figure 18 the onion O_u denotes the onion from party P_j to party P_{j+1}.

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 [36] 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

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

we have that

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

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 $\mathcal{F}_{\mathrm{OR}}$ 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).[13]

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 Secure Realization

The proof of secure realization that we present here is very close to the proof presented by Backes et al. [2] for the realization of $\mathcal{F}_{\mathrm{OR}}$ 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 $\mathcal{F}_{\mathrm{OR}}$ compared to the scenario in which Π_{OR} is used.

Theorem 4. Π_{OR} securely realizes $\mathcal{F}_{\mathrm{OR}}$ in the $\mathcal{F}_{\mathrm{SCS}}, \mathcal{F}_{\mathrm{REG}}^{\mathcal{N}}-$ 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 $\mathcal{F}_{\mathrm{SCS}}, \mathcal{F}_{\mathrm{REG}}^{\mathcal{N}}$ - 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 $\mathcal{F}_{\mathrm{SCS}}$ and $\mathcal{F}_{\mathrm{REG}}^{\mathcal{N}}$ 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 $\mathcal{F}_{\mathrm{SCS}}$ and $\mathcal{F}_{\mathrm{REG}}^{\mathcal{N}}$. That is, we move all the children nodes

---

[13]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.

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. Hence, **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 remains correct, we first double the speed coefficient 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: In Game_4, we adopt the visible subpath computations from [2]. The onions do not contain the real messages anymore but only the constant zero bitstring. Π_4 maintains a shared datastructure q in which the real messages are stored.

Recall that 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 $(P_i)_{i=u}^{s}$. In particular, the onion proxy is also a guard. Every circuit can be split into a sequence of visible subpaths and guards. Π_4 stores for every circuit $(P_i, k_i, cid_i)_{i=1}^{\ell}$ such a splitting into visible subpaths and guards. These splittings are updated upon each **compromise** command.

Upon receiving a **send** input or a response from a network, Π_4 stores an input message m in a shared datastructure q as follows. For a guards P, let cid_P be the circuit id for which P knows the key. Let s the state of the wrapping algorithms of the sender before computing the onion. Then, we store $q(cid_P, s) \leftarrow m$ for each P.

The adversary might be able to corrupt onions such that the contained plaintext is changed. Π_4, however, does not rely on the content of the onions anymore but rather looks up the message in the shared memory. Therefore, Π_4 needs a way to derive the changes to the plaintext due to possible modifications of the ciphertexts. At this point our predictable malleability applies, and we use the algorithm D from the onion secrecy definition for computing the changes in the plaintext. However, for computing the changes in the plaintext, we need to store the onions that the receiving guard has to expect. Hence, Π_4 maintains a shared datastructure *onions* indexed by the *cid* of the receiving guard that stores the expected onions.

Π_4 initially draws some distinguished random key k_S, which is later used for a distinguished last wrapping-layer of the constant zero bitstring. Whenever in Π_3 a guard P that is neither the exit node nor the onion proxy would unwrap an onion O with key k and circuit id cid, P looks up $O' = pending(cid)$. Then, it runs $T \leftarrow S(O, O')$ and replaces the real message $m \leftarrow q(cid, st)$ in the shared memory with $T(m)$, where st is the state of the onion algorithms in the forward direction. Then, P unwraps O with the fake flag, i.e., $(O'', st') \leftarrow UnwrOn(O, k_S, \mathsf{fake}, st)$ instead of $UnwrOn(O, k, st)$. We set the fake flag, because the unwrapping has to skip the integrity check; otherwise a corrupted onion would already in the middle of the circuit be stopped in Π_4. However, instead of forwarding O'', P constructs a new onion either for the adversary or for the next guard as follows. P looks up the adjacent visible subpath $(P_i)_{i=u}^{s}$ in forward direction. If $s = \ell$, then P constructs the onion for the adversary. P reads the real message $m \leftarrow q(cid, st)$ from the shared memory and sends a forward onion O_j for the subcircuit $(P_i, k_i, cid_i)_{i=u}^{\ell}$ that contains the message m and is constructed as follows:

> $O_{u-1} \leftarrow m$
> **for** $i = u$ **to** ℓ **do** $(O_i, st') \leftarrow WrOn(O_{i-1}, k_{j+u-i}, st)$

Only then, P updates the forward state $st \leftarrow st'$. Thereafter, P stores $q(cid_{P_{j+1}}, st') \leftarrow O_u$, where $cid_{P_{j+1}}$ is the circuit id of the guard P_{j+1}.

If $s < \ell$, P sends a forward onion for the subcircuit $(P_i, k_i, cid_i)_{i=u}^{s+1}$ that contains $0^{|m|}$ instead of m, where we replace for the last layer k_{s+1} by the distinguished key k_S. Again only then, P updates the forward state $st \leftarrow st'$. Analogously, guards that are onion proxies, i.e., construct an onion in forward direction, also only construct an onion for the adversary or the next guard.

Similar to the forward direction, guards that receive an onion O in backward direction do not wrap it further as in Π_3 but first unwrap O with the fake flag and the distinguished key k_S, i.e., $O' \leftarrow UnwrOn(O, k_S, \mathsf{fake})$. Instead of wrapping O as in Π_3, the guard constructs an onion for the adjacent subpath in backward direction as follows. Since P is a guard for the circuit, also the onion proxy is honest, thus $u > 1$. P looks up the adjacent visible subpath $(P_i)_{i=u}^{s}$ in backward direction. Let $m \leftarrow q(cid, st)$ be the real message stored in the shared memory, cid be the circuit id for which P knows the key and s be the state of the onion algorithms in the backward direction. Then, P sends an onion $(O, st') \leftarrow UnwrOn(0^{|m|}, \langle k_i \rangle_{i=u-1}^{s}, st)$, where $k_{u-1} := k_S$. Thereafter, update the backward state $st \leftarrow st'$.

It might happen that the adversary compromised a node in the middle of the circuit and the exit node. Then the adversary sends a random message to an honest node P. In this case, P would honestly unwrap the message. Since the adversary controls the exit node the broken integrity is not realized. But from that point on the guard P is out of sync, i.e., P has a different unwrapping state than the predecessor guards. Consequently, by the synchronicity of the onion algorithms all future messages that are sent from the onion proxy will be garbage. For guards that are out of sync, we only send randomly chosen messages of appropriate length.

Then, by a hybrid argument it follows that any adversary distinguishing Game_3 from Game_4 can be used for breaking onion secrecy or synchronicity, where the hybrids are indexed by the circuits of honest onion proxies in the order in which the circuits are initiated. Hence, Game_3 and Game_4 are indistinguishable.

While this visible subpath computation 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 $\mathcal{F}_{\mathrm{OR}}$ for doing the visible-subpath computations. That is, every input from the environment is directly forwarded to $\mathcal{F}_{\mathrm{OR}}$, 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 $\textsc{Env}$.

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 $\mathcal{F}_{\mathrm{OR}}$ 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].

upon setup from parent
1: send setup to ρ; wait for (ready, $\langle P_j \rangle_{j=1}^n$)
2: store $\langle P_j \rangle_{j=1}^n$; output ready to parent

upon a message (received, (m, cid)) **from** ρ
1: lookup the state s of Reassemble for sid$'$
2: call $(m', s') \leftarrow$ Reassemble(m, s); store s' for sid$'$

3: **if** ready $\neq m'$ **then**
4: send (received, m') to parent

upon a message (exit, (m, sid)) **from** ρ
1: lookup the state s of Reassemble for sid$'$
2: call $(m', s') \leftarrow$ Reassemble(m, s); store s' for sid$'$

3: **if** ready $\neq m' = (m'', \text{sid}'', a)$ and a is a server **then**
4: send (exit, $(m'', \text{sid}'', a), \text{sid}$) to parent

upon input (response, (sid, m)) **from parent**
1: $m_1, \ldots, m_q := \text{Split}(m)$ (for $q = \lceil |m|/blockln \rceil$)

2: **for all** $i \in \{1, \ldots, q\}$ **do**

3: send message (response, sid, m_i) to ρ

upon input (send, m) **from parent**
1: $\mathcal{P} \leftarrow$ PathSelection(P_u)
2: **if** if there is no open circuit cid **then**
3: send message (createcircuit, $\mathcal{P}$) to P_u
4: wait for response (created, $\mathcal{C}$)
5: $m_1, \ldots, m_q := \text{Split}(m)$ (for $q = \lceil |m|/blockln \rceil$)

6: **for all** $i \in \{1, \ldots, q\}$ **do**
7: send message (send, $\mathcal{C}, m_i$) to ρ

upon a message (destroyed, cid, m) **from** ρ
1: mark cid as closed
2: proceed as in (send, m)

PathSelection(P_u):

1: $l \xleftarrow{R} \{1, \ldots, n\}$
2: $N := \{1, \ldots, n\}$
3: **for** $j = 1$ to l **do**
4: $i_j \xleftarrow{R} N$
5: $N := N \setminus \{i_j\}$
6: **return** $(P_u, P_{i_1}, \ldots, P_{i_l})$

Figure 19: Wrapper $\Pi_{\text{WOR}\rho}$ for client P_u and a sub-protocol ρ

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.

$\square$

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 19) 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 note 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, that is similar in spirit to the anonymous channel functionality suggested by Canetti [11]. However, such an anonymous channel functioanlity would have as much leakage as and give the adversary 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 $m' \neq$ 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.

6 Timing Attacks in TUC

While previous frameworks only allowed modeling time-independent traffic features such as packets-counts or direction-changes, the communication model in TUC allows us to capture common timing features of traffic such as inter–packet delay, throughput and round–trip–times. Subsequently, we briefly discuss how these features are represented in TUC and how they can be exploited by the network adversary. Our analysis is inspired by the adaptive extension of the AnoA framework [4]. Due to space constraints, we use a simplified version of the anonymity notions presented therein.

6.1 The Set-Up

We consider the class of environments that consist of two sub-machines, an environment adversary $\mathcal{A}_{\mathrm{ENV}}$ and the challenger Ch. The environment adversary $\mathcal{A}_{\mathrm{ENV}}$ is connected to the network adversary and the challenger is connected to the users and defines the security game. In the following, we will use the term adversary to denote the collaboration of the environment adversary and the network adversary. By the completeness of the dummy adversary (Lemma 3) it suffices to consider the network dummy adversary and the environment adversary.

Each party consists of a the Π_{WOR} protocol with the onion routing protocol as a child. By Theorem 4 , we directly consider the ideal functionality $\mathcal{F}_{\mathrm{OR}}$ instead of the onion routing protocol Π_{OR} ($\mathcal{F}_{\mathrm{OR}}$ is depicted in Figure 14).

This set-up is exemplified in Figure 20. It depicts three instances of $\mathcal{F}_{\mathrm{OR}}$ together with the wrapper Π_{WOR}, the shared memory MEM used by $\mathcal{F}_{\mathrm{OR}}$, a challenger Ch, the network adversary and the environment adversary $\mathcal{A}_{\mathrm{ENV}}$.

Sender anonymity challenger SACh**.** For illustrating attacks, we present a guessing-based sender anonymity game via a sender anonymity challenger SACh (which instantiates Ch in Figure 20). In this sender anonymity game the environment adversary $\mathcal{A}_{\mathrm{ENV}}$ has to determine the sender of a specific message-stream in the presence of noise, i.e., other message-streams. $\mathcal{A}_{\mathrm{ENV}}$ has to link at least one session to the correct sender address. Recall that $\mathcal{A}_{\mathrm{ENV}}$ can observe all compromised network links $\mathcal{L}$ though the dummy adversary.

We model a scenario in which users are randomly assigned to addresses, and (in the case of the sender anonymity game) the adversary has to guess which user sits at which address. An address in our model is represented by a party, and a user is represented by a *user model*. For each server $S \in \mathcal{S} := \{S_1, \ldots, S_l\}$, there is a user model UM_S that reactively creates messages for a (potentially interactive) communication with a server S. The user model in particular also decides, when a specific message is sent.

Technically, a user model UM_S is a randomized PPT machine that upon a server message r outputs a sequence $((msg_1, t_1), \ldots, (msg_a, t_a))$, consisting of a client message msg_i and the time t_i at which msg_i shall be sent. Initially, it expects a distinguished message fresh to start a new session.

The sender anonymity challenger SACh allows the adversary to initially register a user model for every server. Then, SACh randomly assigns parties $P_1, \ldots, P_l$ (i.e., addresses) to servers $S_1, \ldots, S_l$ (i.e., to user models) and internally runs the user models UM_S. SACh forwards each message msg_i from UM_S as a send-command from P to S at time t_i and forwards any response r from S to P to the user model UM_S as input.

Moreover, for the analysis of sender anonymity, we can assume that all servers are compromised. Hence, SACh forwards all messages from the servers to the environment adversary $\mathcal{A}_{\mathrm{ENV}}$. Finally, upon an input $(\mathsf{guess}, (P', S'))$, SACh checks whether the user u is assigned to the user model UM_S for the server S. If so, SACh outputs 1; otherwise it outputs 0.

Onion routers O_i**.** In addition to the regular users, i.e., protocol parties that are controlled through user models, we also assume protocol parties $O_1, \ldots, O_v$ that run the same protocol code but only serve as onion routers. We assume that users and onion routers are distinct, i.e., $\forall i, j. O_i \neq P_j$.

6.2 Mounting Attacks that use Timing Features

In TUC timing-based traffic features can be measured by the adversary. Example 4 details how timing features can be measured in TUC. Subsequently, we discuss how timing based traffic analysis attacks from the literature can be mounted using these features [14, 19, 43, 31, 23, 30, 41, 44, 46, 47, 55, 32].
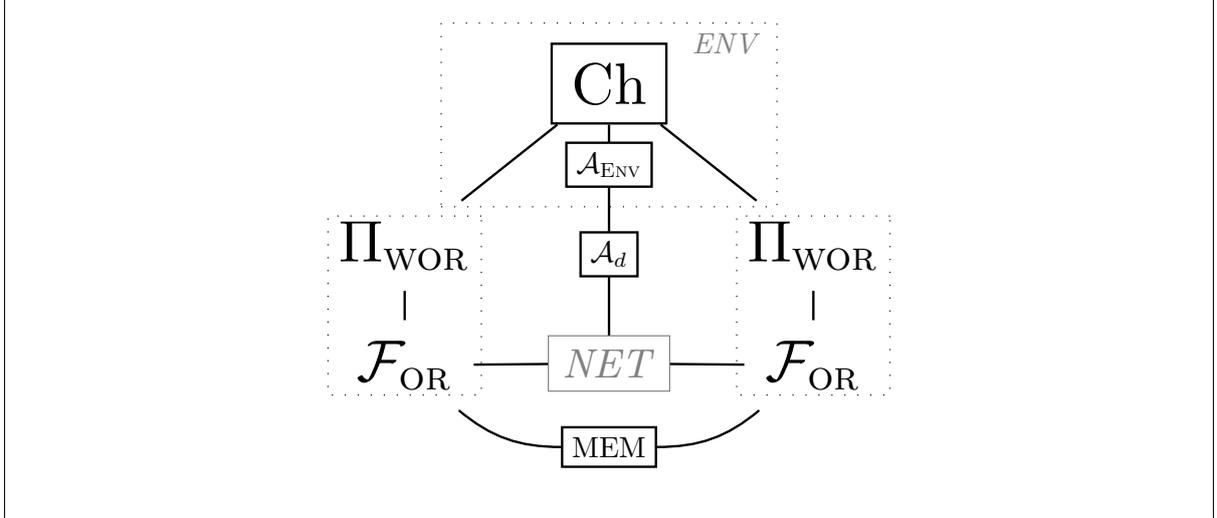
Figure 20: The Attack Scenario

Example 4: Observing timing features in TUC. We examine how the time at which a message is sent from a party P correlates to P's speed coefficient c_P. For simplicity, we assume that party P just created a new circuit before receiving the message from SACh. We further assume that both machines inside P have the same speed coefficient c_P.

1: SACh sends (send, m) to party P at time T
2: Π_{WOR} in P receives (send, m) at $T' = \frac{k}{c_P} \geq T$, where T' is the first time after T where Π_{WOR} is in the listen state
3: Let n be the number of steps needed to split m into $q := \lceil |m| / blockln \rceil$ packets $m_1, \ldots, m_q$.
   Π_{WOR} sends m_i at time $T_i = T' + \frac{n+i}{c_P}$ to $\mathcal{F}_{\mathrm{OR}}$
4: Let n' be the time that $\mathcal{F}_{\mathrm{OR}}$ needs to send a message and $v[4]$ the delay for being synchronized with Π_{OR}.
   $\mathcal{F}_{\mathrm{OR}}$ forwards m_i at time $T' + v[4] + \frac{n+1+i\cdot n'}{c_P} =: T' + g(c_P, i)$, since messages $m_2, \ldots, m_q$ arrive at $\mathcal{F}_{\mathrm{OR}}$ before n'/c_P time has passed.

The *traffic pattern* a user model UM_S generates is the stream of message blocks that are generated if, for the sequence of messages $((msg_1, t_1), \ldots, (msg_a, t_a))$, the message msg_i is sent at the instructed time t_i. Thus, the corresponding traffic pattern is preserved when $\Pi_{\mathrm{WOR}\,\mathcal{F}_{\mathrm{OR}}}$ sends its message blocks.

If the network links through which the messages, or in the case of $\mathcal{F}_{\mathrm{OR}}$ the message handles, are sent are compromised, the network adversary $\mathcal{A}$ learns the time at which the messages cross the network. From this $\mathcal{A}$ can determine different timing features of the traffic, e.g, he can determine the difference $g(c_P, i+1) - g(c_P, i)$ between the two messages m_i and m_{i+1}, or $\mathcal{A}$ can learn at which time how much throughput, i.e., how many message blocks per time, passed through the link. Thus, it learns the traffic patterns of the message stream generated by the user model.

Similarly to the function g, which estimates the delay created by creating a stream of cells from a message m_i, we can also determine a function h for the delay created by relaying a message through a onion router. This function h solely depends on the speed of the onion router and i.

Consider a circuit with the onion routers O_1, O_2, O_3. The time at which a message m_i is sent from an exit link O over the network to the server is $g(c_P, i) + h(c_{O_1}, i) + h(c_{O_2}, i) + h(c_{O_3}, i) + d + w$ for some network delay d that is caused by other messages that the onion routers have to concurrently process and, optionally, for some watermarking delay w that the adversary deliberately introduces for recognizing traffic (as studied in previous work [19, 43, 31, 32]).

Let $|\mathcal{L}|$ be the number of compromised links and M be the number of total links between the protocol parties. Then, the probability that the entry link from P to the entry node, i.e., to the first onion router, is compromised is $|\mathcal{L}|/M$. This is therefore the probability with which $\mathcal{A}$ observes the links that belong to the same connection and can then try to correlate the traffic. The success of this correlation however depends on the methods used by $\mathcal{A}$.

For a passive network adversary, which does not introduce any watermarking delay, we get the following: if the traffic patterns of all user models are sufficiently well distinguishable and the network

delay is sufficiently small (i.e., there is not much traffic on the onion routing network), then the traffic pattern of a user model is recognizable, i.e., the traffic can be correlated, for an adversary in TUC. For an active adversary, which does introduce watermarking delays, it is possible to compensate the network delay, and such an adversary can even recognize a stream if the user models produce exactly the same traffic patterns. Thus, an active adversary can recognize a user model as soon as it controls the entry, i.e., with more than $|\mathcal{L}|/M$ probability.[14] $\diamond$

Inter-packet delay. Inter-packet delay is the time difference in the time stamps of two consecutive packets sent through a connection. Example 4 illustrates how the time distance $t_{i+1} - t_i$ between two messages $((m_i, t_i), (m_{i+1}, t_{i+1}))$ from the user model is preserved in the sequence of message blocks that $\Pi_{\text{WOR}\mathcal{F}_{\text{OR}}}$ produces. Moreover, it illustrates how a delay $g(c_P, i+1) - g(c_P, i)$ between two message blocks that belong to the same message is produced and that this delay depends on the speed coefficient of the party. These delays reflect the inter-packet delays used in traffic analysis attacks from the literature. We are aware that we abstract from the delays that, in the real world, are produced by low-level network protocols, such as TCP and IP and from machine specific hardware delays, e.g., induced by a machine's network card. In principle, however, TUC and $\mathcal{F}_{\text{OR}}$ allow for fine-grained modeling of these timing features by introducing the respective protocols as sub-protocols of the onion routing protocol.

As discussed in Example 4, if the network delay is small and the traffic patterns of the user models are distinguishable, the inter-packet delays in the traffic pattern of a user model can be correlated even for a passive adversary.

Traffic watermarking attack. In a traffic watermarking attack, the adversary deliberately causes a delay pattern, called a watermark, for a packet stream, e.g., at the entry link, and measures at the other links, e.g., at exit links, whether it recognizes such a watermarked message stream. We illustrated in Example 4 how such a traffic watermarking attack could be modeled in TUC.

Similar to recognizing watermarks and inter-packet delays, the adversary can measure other timing features such as throughput and round-trip-times in the network. For round-trip times, the speed coefficients of the onion routers potentially give a unique fingerprint if they are sufficiently different. For simplicity, we omit network latency on links. Network latency can however be modeled by refining the network topology NET in the execution EXEC.

The attack described above only acts passively as it collects messages sent through the network and evaluates them. But, especially for traffic correlation, active methods such as *traffic watermarking* are very popular in the literature [19, 43, 31, 32]. Here the adversary slightly modifies the traffic he intercepts, creating watermarks that are easier to spot if the same traffic is observed at some other point in the network. These kinds of attacks are also possible in our model, as we do allow active adversaries (we are only restricted to static corruption).

Modeling website-fingerprinting attacks. A website fingerprinting attack assumes that the adversary up-front possesses a list of fingerprints for each server, which characterizes connections to these servers based on traffic features, e.g., direction changes in traffic, throughput, round-trip-times and inter-packet-delays. $\mathcal{A}$ then only needs to listen to the entry links from users to the onion routing network and collect the messages (together with time-stamps) that go through this entry link. After collecting sufficiently many messages, he can then match the fingerprints he possesses to the traffic he intercepted. In the literature [8, 48, 27] several successful website-fingerprinting attacks are known.

While previous frameworks already allowed fingerprinting websites based on time-insensitive traffic features, such as overall size of traffic and direction changes in traffic, an adversary in TUC can utilize timing features of traffic for website-fingerprinting as well. In turn, proving the absence of attacks in TUC excludes the entire family of attacks that use time-sensitive traffic features, such as throughput in time and inter-packet delay. In the next section we propose a countermeasure against the class of website fingerprinting attacks against the onion routing protocol and prove this countermeasure to be secure.

---

[14] For simplicity, we give a very coarse approximation of the probability that the adversary control the entry link. The adversary could additionally compromise onion routers and thereby increase its chance to control the entry point of a circuit.

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

Client: upon (req, $address$) **from the parent**
1: send (send, $address$) to ρ

Client: upon m **from** ρ
1: output m to the parent

Exit node: upon (exit, $((m'', \mathsf{sid}'', a), \mathsf{sid}'))$ **from** ρ
1: **if** $bun_{\mathsf{sid}''} = \bot \wedge a$ is a server **then**
2:     lookup and store current time in $reqStart(\mathsf{sid}')$
3:     store $((m'', \mathsf{sid}', a), \mathsf{sid}')$ in $bun_{\mathsf{sid}''}$
4: **while** $\exists\, request(loc, a)$ in $bun_{\mathsf{sid}''}$ **do**
5:     remove $request(loc, a)$ from $bun_{\mathsf{sid}''}$
6:     send $(loc, (a, \mathsf{sid}'))$ over the network
7:     store the response res in $bun_{\mathsf{sid}''}$
8:     lookup the smallest bucket size $n' \geq |bun_{\mathsf{sid}''}|$ in pageBuck
9:     pad $bun_{\mathsf{sid}''}$ to a size of n' and store it in m'
10:     let $t \leftarrow t_{\mathrm{buf}} + reqStart(\mathsf{sid}')$
11:     send (response, (sid', m')) to ρ at time t

Figure 21: The protocol WFC$_\rho$ for party pid, where sid is its session ID

7   Countermeasure against Website Fingerprinting

We leverage our time-sensitive framework and our Tor abstraction to propose a countermeasure against website fingerprinting and prove it secure. The countermeasure achieves k-recipient anonymity for web pages without dynamic requests, such as Ajax.

The countermeasure protocol, called WFC, is plugged on top of the Tor protocol. At an exit nodes, WFC performs all web page requests until the web page is fully loaded and returns the entire web page at once to the user. In order to remove size features of web pages, the response packet-stream are padded to a common denominator for all web pages. WFC additionally waits until a time buffer t_{buf} has passed in order to remove traffic-related timing features.

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 (instead of padding to a common size of all web pages). The data structure pageBuck contains a bucket for each target web page size and upon input of a size n, pageBuck(n) returns the list of web pages that have size n or are padded to size n. The countermeasure protocol WFC is depicted in Figure 21.

We next describe the *k*-recipient anonymity challenger.

The k-recipient anonymity challenger RACh   We consider following notion of recipient anonymity: an adversary that control all entry links of a party P, e.g., an ISP-level adversary, should not be able to determine the web pages that the party P visits. The set-up for recipient anonymity is exactly as for the sender anonymity game (See Section 6.1) except that the challenger Ch is replaced by the following k-recipient anonymity challenger RACh. This challenger RACh initially allows the environment adversary to define the page-buckets used in the game, but requires that each bucket in pageBuck contains exactly k web pages.

In contrast to the sender anonymity game, RACh does not allow the environment adversary to control the servers. Instead, we assume that the adversary solely controls all links connected to the parties representing users (i.e., controls entry links to the onion routing network).

In order to strengthen the adversary, we allow it to choose the time at which a user sends a web page request to a server. Similar to the sender anonymity game the environment adversary has to guess the correct server/request pair.

In the following, we describe the code of the web servers.

Web pages.   For our purposes it suffices to represent web pages as lists of *elements*, which are associated with *locations* on the web server and are returned upon requests for these locations. Elements are arbitrary bit-strings m that are marked as elements; we denote them as $element(m)$.

A page request consists of a pair of party ID a and a location loc (represented by a bit-string), which we denote as $request(loc, a)$. The pair (loc, a) can be understood as the url that is requested from the user, where a denotes the domain and loc denotes the path to a specific web page on the domain a. Note

```
upon (register-webpages, pageBuck) from 𝒜_Env          1: draw a random j ∈ {1, ..., k}
  1: for all sizes n do                                 2: let (a_j, loc_j, pg_j) ← pageBuck(n)
  2:    if |pageBuck(n)| = k then                       3: (a_ch, loc_ch) ← (a_j, loc_j)
  3:       for all (a, loc, pg) ∈ pageBuck(n) do
  4:          if (a, loc) is unregistered and loc ≠     upon (send, (a, loc)) from 𝒜_Env
             challenge then                               1: in the first call: send setup to ρ; wait for ready
  5:             send (register, loc, pg) to Π_server at  2: if loc = challenge then (a, loc) ← (a_ch, loc_ch)
                party a                                   3: send (a, loc) to P
  6:    else
  7:       return error                                 upon (guess, (a, loc)) from the 𝒜_Env
                                                          1: if (a, loc) = (a_ch, loc_ch) then output (guess, 1)
upon (challenge, n) from 𝒜_Env                            2: else (guess, 0)
```

Figure 22: k-recipient anonymity challenger: RACh_k

that the countermeasure WFC only provides recipient anonymity guarantees for web pages that do not dynamically load content upon user inputs, e.g., by using JavaScript techniques such as Ajax.

Server protocol Π_{server}. Web servers are modeled as a protocol Π_{server}, which can be thought of an abstraction of server software, e.g., Apache HTTP Server.[15]

The server protocol can register several locations on the machine it runs on. Formally, upon a message of the form $(\mathsf{register}, loc, pg)$ from its parent, Π_{server} registers a web page pg at the location loc. Upon a network message $(loc, (a, \mathsf{sid}'))$, the server then responds with the web page pg.

The k-recipient anonymity for WFC follows from the composition theorem (Theorem 2), the realization theorem (Theorem 4), the definition of $\mathcal{F}_{\mathrm{OR}}$ and from the fact that all web-pages in the same bucket are padded to the same size and that WFC removes timing features from the traffic by introducing artificial delays.

Lemma 4. *Let* EXEC' *be defined as* EXEC *except that* EXEC' *outputs the bit* b *from the first output of the form* (guess, b) *by* RACh. *Let* $\langle \mathrm{RACh}, \mathcal{A}_{\mathrm{ENV}} \rangle$ *denote the machine that contains* RACh *and* $\mathcal{A}_{\mathrm{ENV}}$, *as described in Section 6.1. Let* $\langle \mathrm{WFC}_{\Pi_{\mathrm{WOR}}\Pi_{\mathrm{OR}}}, \Pi_{\mathrm{server}} \rangle$ *denote the combined protocol with the countermeasure along with the wrapper and the onion routing protocol and the server protocol.*

For any PPT *environment adversary* $\mathcal{A}_{\mathrm{ENV}}$ *and a dummy network adversary* $\mathcal{A}_d$ *that only compromises entry links or entry nodes, for sufficiently large* t_{buf} *and* η *and a negligible function* μ *we have*

$$\Pr[\mathsf{EXEC}'_{\eta}(\langle \mathrm{WFC}_{\Pi_{\mathrm{WOR}}\Pi_{\mathrm{OR}}}, \Pi_{\mathrm{server}} \rangle, \mathcal{A}_d, \langle \mathrm{RACh}, \mathcal{A}_{\mathrm{ENV}} \rangle) = 1] \leq 1/k + \mu(\eta)$$

Proof. By the realization theorem (Theorem 4) and the composability theorem (Theorem 2), $\mathrm{WFC}_{\Pi_{\mathrm{WOR}}\Pi_{\mathrm{OR}}}$ securely realizes $\mathrm{WFC}_{\Pi_{\mathrm{WOR}}\mathcal{F}_{\mathrm{OR}}}$; hence a network adversary against $\mathrm{WFC}_{\Pi_{\mathrm{WOR}}\mathcal{F}_{\mathrm{OR}}}$ (and hence also against $\mathrm{WFC}_{\Pi_{\mathrm{WOR}}\Pi_{\mathrm{OR}}}$) that only compromises the entry node of one client or the link to the entry node of one client only learns the *cids* 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 RACh has output $(\mathsf{guess}, 1)$, 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 in $\mathsf{pageBuck}(k)$ is the same. □

8 Conclusion and Future Work

In this work, we presented TUC, a formal framework for the analysis of complex mutil-party protocols 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 allow for strong compositionality: a universal composability result, and the completeness of the dummy adversary. We apply this framework to the widely deployed Tor network and showed that a previous abstraction of the onion routing protocol [2] can be suitably extended to account for timing and that it is realized in TUC by a similarly

---

[15]Technically, Π_{server} is the "server" role in the WFC protocol; otherwise they cannot be peers.

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, in particular traffic correlation attacks. Since our framework comprehensively models timing attacks, every verification of the abstraction for onion routingyields security guarantees for the actual OR protocol.

For future work there are scenarios in which it is crucial to characterize the network topology in a more detailed way. Possible extensions include adding the latency and the throughput of a link, allowing not only single links between two parties but several links (a multigraph topology), and including weights to each link to model routing preferences. Such extensions could for example be used for the analysis of denial-of-service resistance mechanisms or for the for the analysis of more sophisticated path selection algorithms for onion routing or analyzing denial of service attacks.

There is a line of work on automated verification techniques for timed automata. It would be interesting to explore in which cases timed automata are a sound abstraction for TUC protocols. Such a result would allow to obtain strong guarantees, i.e., against computational adversaries that can perform time-measurements, from established automated verification tools [15, 33, 7, 40, 37, 38]. Moreover, there is an information theoretic analysis of web traffic which uses an abstraction of web-traffic. It would be interesting to utilize TUC to prove that their abstraction is sound, i.e., that all attacks in TUC are reflected in their abstraction.

References

[1] Michael Backes. Unifying Simulatability Definitions in Cryptographic Systems under Different Timing Assumptions. *Journal of Logic and Algebraic Programming (JLAP)*, 2:157–188, 2005.

[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, Praveen Manoharan, Sebastian Meiser, and Esfandiar Mohammadi. Anoa: A framework for analyzing anonymous communication protocols. Cryptology ePrint Archive, Report 2014/087, 2014.

[5] 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.

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

[7] 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.

[8] 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.

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

[10] Ran Canetti. Security and Composition of Multi-party Cryptographic Protocol. *Journal of Cryptology*, 13:143–202, 2000.

[11] 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.

[12] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2013.

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

[14] 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.

[15] 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.

[16] Anupam Datta, Ralf Küsters, John C. Mitchell, and Ajith Ramanathan. On the Relationships Between Notions of Simulation-Based Security. In *Proceedings of the 2nd Conference of the Theory of Cryptography (TCC)*, pages 476–494, 2005.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] Nethanel Gelernter and Amir Herzberg. On the limits of provable anonymity. In *Proceedings of the 12th ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 225–236, 2013.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

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

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

[30] 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.

[31] Amir Houmansadr and Nikita Borisov. SWIRL: A Scalable Watermark to Detect Correlated Network Flows. In *Proceedings of the 18th Annual Network & Distributed System Security Symposium (NDSS)*, 2011.

[32] Amir Houmansadr and Nikita Borisov. The Need for Flow Fingerprints to Link Correlated Network Flows. In *Proceedings of the 13th Privacy Enhancing Technologies Symposium (PETS)*, 2013.

[33] 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.

[34] Yael Tauman Kalai, Yehuda Lindell, and Manoj Prabhakaran. Concurrent Composition of Secure Protocols in the Timing Model. *Journal of Cryptology*, 20(4):431–492, 2007.

[35] Jonathan Katz, Ueli Maurer, Bjrn Tackmann, and Vassilis Zikas. Universally Composable Synchronous Computation. In *Proceedings of the 10th Theory of Cryptography Conference (TCC)*, pages 477–498, 2013.

[36] 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.

[37] 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.

[38] Miroslaw Kurkowski, Wojciech Penczek, and Andrzej Zbrzezny. SAT-Based Verification of Security Protocols Via Translation to Networks of Automata. In *Proceedings of the 4th Workshop on Model Checking and Artificial Intelligence*, volume 4428, pages 146–165, 2007.

[39] Ralf Küsters and Max Tuengerthal. The iitm model: a simple and expressive model for universal composability. IACR Cryptology ePrint Archive, Report 2013/025, 2013.

[40] 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.

[41] 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.

[42] P. Mateus, J. Mitchell, and A. Scedrov. Composition of Cryptographic Protocols in a Probabilistic Polynomial-Time Process Calculus. In *Proceedings of the 14th International Conference on Concurrency Theory (CONCUR)*, pages 327–349. Springer Berlin Heidelberg, 2003.

[43] Prateek Mittal, Ahmed Khurshid, Joshua Juen, Matthew Caesar, and Nikita Borisov. Stealthy Traffic Analysis of Low-Latency Anonymous Communication Using Throughput Fingerprinting. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[44] 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.

[45] Jesper Buus Nielsen. *On Protocol Security in the Cryptographic Model*. dissertation, University of Aarhus, 2003.

[46] 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.

[47] 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.

[48] 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.

[49] Birgit Pfitzmann, Matthias Schunter, and Michael Waidner. Cryptographic Security of Reactive Systems: (Extended Abstract). *Electronic Notes in Theoretical Computer Science*, 32:59–77, 2000.

[50] Manoj Prabhakaran and Mike Rosulek. Homomorphic Encryption with CCA Security. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 5126, pages 667–678, 2008.

[51] 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.

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

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

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

[55] 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.