2014 IEEE 27th Computer Security Foundations Symposium

# TUC: Time-sensitive and Modular Analysis of Anonymous Communication

Michael Backes
CISPA, Saarland University
backes@cs.uni-saarland.de

Praveen Manoharan
CISPA, Saarland University
manoharan@cs.uni-saarland.de

Esfandiar Mohammadi
CISPA, Saarland University
mohammadi@cs.uni-saarland.de

***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.**

## I. Introduction

Anonymous communication protocols, as provided by the Tor network [1], 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 [2].

For precisely understanding the anonymity guarantees provided by Tor, several rigorous analyses have been conducted [3], [4], [5], [6], [7], [8], 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 [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]. 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. [21] and put forward by [22], [23], [24], [25], [26], [27], 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 [25], which is one of the recent pieces of work [23], [26] that address many of the problems faced by earlier designs for simulation-based security frameworks [23].

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 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. As a case study, we leverage TUC to analyze a simple countermeasure against website fingerprinting attacks and to prove $k$-recipient anonymity guarantees for this countermeasure.

**Outline.** Section II discusses related work. Section III introduces the time-sensitive TUC framework, and presents the activation order independence of TUC. Section IV 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 V, we discuss how known traffic-related timing attacks on Tor can be represented in TUC. Moreover,


DOI 10.1109/CSF.2014.34

we provide a countermeasure against website fingerprinting attacks and prove it secure in TUC.

## II. Related Work

Tor [1] is one of the most widely used anonymous communication protocols to date [2] and is based on the (first generation) onion routing protocol by Goldschlag et al. [28]. There has been significant work in analyzing the anonymity guarantees provided by Tor [4], [5], [6], [7], [8], [29]. 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 [17], [13], [9], [19], [16], [30], [31], [32], [33] 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 [22], [23], [24], [25], [26], [27] 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 [22], [23], [24], [25], [26], [27]. 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 [25]. 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 [25, 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[22], IITM [26], and UC [23] 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], [35], [24], [36], [37], [38]. 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. While there has been work on the time sensitive analysis of Dolev-Yao style abstracted cryptographic protocols using timed automata [39], [40], [41], [42], [43], [44], this line of work analyzes Dolev-Yao style abstractions and does not allows for more fine-grained adversary types we want to capture in our model. We therefore stick to a Turing Machine based network model which is typically used in the analysis of cryptographic protocols.

## III. Modeling Time for Anonymity Analysis

In this section, we present TUC: the first simulation-based composability framework that considers a time-sensitive adversary. TUC builds upon previous asynchronous simulation-based frameworks, such as GNUC [25], RSIM [22], UC [23] and IITM [26], but fundamentally extends these frameworks by incorporating a notion of time while preserving universal composability.

We begin with presenting relevant aspects of TUC regarding time and explain how each of the design decisions are motivated by our goal to produce a formal model for the analysis of anonymous communication protocols. Due to space-constraints, we concentrate on the non-trivial mechanisms behind and the main problems for extending previous frameworks to account for time-sensitive adversaries. For a more detailed exposition we refer the interested reader to the extended version of this paper [45].

Thereafter, we proceed with describing properties of TUC that simplify a formal analysis of anonymous communication protocols and serve as sanity checks that emphasize the comprehensiveness of TUC. Finally, we discuss whether it is possible to encode our time-sensitive extension in a traditional simulation-based framework.

However, before we get to timing, we first need to lay the formal groundwork in form of a formal communication model to which we then apply our modifications.

**Notation: Computational indistinguishability.** We for two distributions $D$ and $D'$, we write $D \approx D'$ to denote that $D$ and $D'$ are computationally indistinguishable, i.e., for a negligible

function $\mu$ and for sufficiently large security parameter $\eta$

$$\left|Pr[D = 1] - Pr[D' = 1]\right| < \mu(\eta)$$

In this work, the distribution $D$ and $D'$ typically describes the output of a network execution, which internally runs an arbitrary environment, the protocol parties, and the adversary.

### A. Communication Model in TUC

Similar to previous frameworks, we model a network consisting of regular *network parties* $P_i$, a *network adversary* $\mathcal{A}$ and an *environment* ENV. $\mathcal{A}$ represents adversarial behavior in the network, while ENV represents user behavior, an operating systems, or other entities that call protocols.

Each machine $M$ in the network is identified with a unique *machine id* $\mathsf{id}(M)$ which is needed for identifying and addressing machines. This machine ID in particular includes the protocol name protNAME, the session parameter $sp$, and the role role, called *basename* $\mathsf{name}(M) = (\mathsf{protNAME}, sp, \mathsf{role})$ of $M$. This basename determines the code that $M$ executes. The code is supplied by the *protocol* $\Pi$ of the network.

**Definition 1.** *A* protocol *is a function* $\Pi : D \to \{0,1\}^*$*, which for every* machine-id $id$ *in its* domain $D$ *gives the code* $c \in \{0,1\}^*$ *run by every machine* $d$*.*

A regular network party $P$ consists of a tree of machines $M_1, \ldots, M_l$ that represents the protocol stack used by $P$: every child-machine $M_c$ of a parent-machine $M_p$ executes a sub-routine invoked by $M_p$. The protocol $\Pi$ restricts the set of protocol-names $\{d_1, \ldots d_l\} \subset D$ that a machine with protocol-name $d \in D$ can call as subroutines. We inherit this tree-structure from GNUC [25] since it allows for a simple definition of protocol composition (see Section IV).

Machines in the network communicate by sending messages to each other. In TUC this is subject to several restrictions: inside a regular network party $P$, machines can only communicate with their children and their parent machines, which can be the environment ENV if the machine is the root of $P$. Across the network, i.e. between parties, a machine $M$ can only communicate to another machine $M$ with the same protocol name in its basename, or with the adversary $\mathcal{A}$. The network adversary can further directly communicate with the environment ENV, which is depicted in Figure 1.

**The network execution.** For each machine in the network we maintain external parameters, needed to ensure concurrent computation and local unsynchronized clocks. Hence, we embed the entire network inside a single 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. In the following we also present pseudocode of fragments of the execution. A detailed specification of EXEC can be found in the extended version of this paper [45].

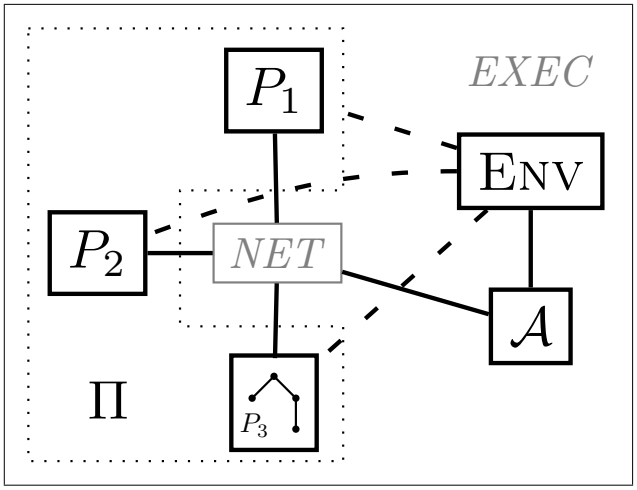



Figure 1: The Communication Model in TUC. The party $P_3$ exemplifies the machine tree that exists in each party.

**Definition 2.** *The execution* EXEC *is a probabilistic, polytime Turing Machine which receives the security parameter* $\eta$ *and outputs a value in* $\{0,1\}$*.* $\mathsf{EXEC}_\eta(\Pi, \mathcal{A}, \text{ENV})$ *denotes the output of* ENV *after* EXEC *ran the network of machines running protocols in* $\Pi$ *together with the network adversary* $\mathcal{A}$ *and the environment* ENV*.* EXEC *stops whenever* ENV *halts and outputs a bit.*

### B. Adversary Model in TUC

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.

*1) 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 [19], [18], [16], [9]. Previous work on the analysis of Tor shows how partial compromisation of the network can be modeled by introducing special network functionalities $\mathcal{F}_{\text{NET}}$, which are used as a link between parties [3]. $\mathcal{F}_{\text{NET}}$ 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 $\mathcal{A}$.

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 message can be directly forwarded to the recipient, or is intercepted by the network adversary. As we discuss for future work in Section V-C and VI, NET can also be extended to include network latency and other parameters of the network.

*2) 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. $cd_{comp}$ 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.

Static corruption can be modelled by only allowing the environment to initially compromise machines and to ignore all compromisation messages after an initial phase.

### C. Timing in TUC

We extend the basic communication model presented in the previous sections to include time. To each machine in the network we attach a timer and utilize the execution EXEC to maintain these timers. In order to allow unsynchronized clocks between network parties, we introduce local-time functions, which transform the timer's value to the local time experienced by each machine. We achieve time-consistency for messages exchanged between machines by introducing time-stamps for these messages. The execution EXEC utilizes these time-stamps for delivering messages to the recipient at the correct time.

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

**Definition 3.** *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 [45, Section 3.1.8] 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 4.** *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 $\eta$ 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 request 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 needs to be able to efficiently compute and invert the local time; thus, we require that the local time function be efficiently computable and efficiently invertible. We additionally capture low-precision local clocks by not requiring that the local time function is injective. We only require that the local time function is *pseudo invertible* in the following sense: the pre-image of a local time $f_M(t)$ is a closed interval $[T, T')$ from which the corresponding global time is randomly chosen.

**Definition 5.** *A function $f : \mathbb{Q} \to \mathbb{Q}$ is* pseudo-invertible *if for every value $x \in f[\mathbb{Q}]$ there is exactly one non-empty, closed or right-open interval $\mathcal{I}$ (i.e. $\mathcal{I} = [T, T']$ with $T \leq T'$ or $\mathcal{I} = [T, T')$ with $T < T'$) such that $\forall y \in \mathcal{I} : f(y) = x$ and $\forall y' \in \mathbb{Q} \setminus \mathcal{I} : f(y') \neq x$. We denote the interval $\mathcal{I}$ as the* pseudo-pre-image *of $x$.*
*Given a pseudo-invertible function $f$, we denote with $f^{(-1)}$ the* pseudo-inverse *of $f$, which, given a value $x \in f[\mathbb{Q}]$, returns its pseudo-pre-image $f^{(-1)}(x) = \mathcal{I}$.*

With the definition of pseudo-invertible functions, we can now define local time functions.

**Definition 6.** *The* local time function $f_M : \mathbb{Q} \to \mathbb{Q}$ *of $M$ is a monotonically increasing, efficiently computable and efficiently pseudo-invertible function that transforms the value of $T_M$ to $M$'s* local time $f_M(T_M)$.

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. Figure 2 illustrates the methods used by EXEC to manage basic timing information.

**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, the current global time $T_M$ for $M$ is updated: $T_M := T_M + \frac{n}{c(M)}$, where $n$ is the number of steps performed by $M$ in its last activation, and $c(M)$ is its speed coefficient.

**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** $(\mathrm{cmd}, t)$ **from** $M \in \{\mathcal{A}, \textsc{Env}\}$
1: **if** $t > 0$ **then**
2: set $T_M := T_M + t$
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 $M \in \{\textsc{Env}, \mathcal{A}\}$

Speed coefficients and local time functions are fixed once the respective machine is spawned. Formally, the speed coefficient and the local time function depend on the machine ID, i.e, on the party ID and the basename. In order to assign speed coefficients to dynamically created machines, we suitably extend our definition of protocol introduced earlier.

We require that the protocol $\Pi$ consists of two functions: one function that maps machine IDs to distributions of speed coefficients and local time functions and one function that maps basenames to the protocol code (see Definition 1). The execution draws the speed coefficients and local time functions from these distributions whenever a new machines is created during runtime. 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 $Mon$ the set of local time functions functions from $A$ to $B$ and with $D$ the set of machine IDs.

**Definition 7** (Protocol)**.** *For a set $P$ of party IDs and a set of basenames $D$, a* protocol *is a pair of functions $\pi := (\pi_p, \pi_c)$ consisting of a function $\pi_p : P \times D \to Dist(\mathbb{N}[X]) \times Dist(Mon)$ from party IDs to an efficiently computable distribution of* speed coefficients *and an efficiently computable distribution of* local-time functions *local time functions and a function $\pi_c : D \to \{0,1\}^*$ from basenames to a* code*.*
*A protocol $\pi' = (\pi'_p, \pi'_c)$ is a* subprotocol *of $\pi = (\pi_p, \pi_c)$ over domain $D'$ if $D' \subseteq D$ and $\pi_c$ restricted to $D'$ equals $\pi'_c$ and $\pi'_p = \pi_p$.*

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, we abstract this strength of the environment and the adversary by allowing both parties to make an arbitrary amount of computation steps per point in time, for the sake of simplifying proofs.[1]

**Definition 8.** *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.*

In contrast to other sequential activation models, messages in TUC are not directly delivered to the recipient as there might be another message from a yet unactivated machine that has to arrive earlier. The execution remedies this problem by utilizing time stamps.

**Definition 9.** *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.*

This time-stamp is attached 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.

**Definition 10.** *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 $T_m \leq T$.*

### D. Activation Order

*1) 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 are sent. We call these decisions the *activation strategy*. Keeping to this unrestricted sequential activation model, however, causes several

[1]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 Lemma 1).

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 $(\mathsf{listen}, T)$ 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 11** (Consistency Enforcing)**.** *An activation order of machines is* consistency enforcing*, if* $\forall M' \neq M : \widetilde{T}_M \leq \widetilde{T}_{M'}$ *whenever* $M$ *is activated from the listening state. Here* $\widetilde{T}_M$ *is the* virtual time *of the machine* $M$ *defined as follows:*

- $\widetilde{T}_M = \min\{T, T_m\}$*, where* $T_m$ *is the smallest time-stamp of a message in* $Q_M$ *(or* $\infty$ *if no such message exists), if* $M$ *is in the* $(\mathsf{listen}, T)$ *state*
- $\widetilde{T}_M = T_M$*, if* $M$ *is not in the* $(\mathsf{listen}, T)$ *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.

**Corollary 1.** *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* $T' > T$*.*

*2) 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 orders, 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 12.** *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 by checking the required conditions whenever ACT wants to activate a machine. The activation methods in EXEC are depicted in Figure 3.

*3) 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** (SAS is equivalent to all deadlock-free activation strategies)**.** *The activation strategy* SAS *is indistinguishable from any other, deadlock-free activation strategy.*

*4) Discussion:*

**Modeling asynchronous communication despite consistency enforcing scheduling.** In other simulation-based frameworks, 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 [23, 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.[2] 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 they can model disabled or crashed network nodes by not activating the respective machine, but only if this machine would not have received any message (otherwise this message would have activated the machine). However, since this mechanism only covers machines that would not have received any messages, we believe that such crashes should rather be modeled explicitly, i.e. in the same way as node corruptions, and not via an activation strategy.

[2]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 III-C).

**activate_listen($M$)**
1: **if** $\forall M' \in \mathcal{M} \setminus \{M\} : \widetilde{T}_{M'} \geq \widetilde{T}_M$ **then**
2: **if** $M$ is timeless **then**
3: set $T_M = \widetilde{T}_M$
4: **else**
5: set $T_M = \frac{\lceil \widetilde{T}_M \cdot c_M \rceil}{c_M}$
6: **if** $Q_M$ is not empty **then**
7: 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 $Q_M$
8: activate $M$ with $m_1, \ldots, m_l$ on ports $p_1, \ldots, p_l$
9: **else**
10: activate $M$ without input
11: activate ACT with a activation request

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

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

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

Figure 3: Activation order methods in EXEC with machine set $\mathcal{M}$ and activation strategy ACT

**Encoding time-sensitive adversaries 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.

### *E. More realistic machine models*

For the sake of a simpler presentation, we assume in this work Turing machines as the underlying machine model. As soon as timing information is taken into account, Turing machines do not properly model real-world systems: e.g., a Turing machine needs in the worst case linear time (in the size of the already written cells) for a memory access, whereas a real-world system performs a memory access in constant time. Such a behavior is much more accurately modelled by other machine models, such as random access machines (RAMs) [46].

It is possible to extend our model to more realistic machine models. Interpreting a single step of a machine as a clock cycle, we can formalize more realistic machine models by mapping different basic instructions, such as addition and multiplication, to different amounts of clock cycles. We can even formalize architectures with caches by mapping the internal state and a basic instruction to some amount of clock cycles.

With such a modular treatment of the machine models, we can, moreover, capture scenarios in which different parties in the network use different machine architectures. Similar to the distribution of the speed coefficients and the local time functions, we require that the machine model is determined by the machine ID, i.e, party ID and the basename. Accordingly, the execution EXEC has be adjusted such that the clock cycle functions are maintained, and instead of increasing the time of a party by 1 for each instruction EXEC increases the time by the sum of cycles given by the clock cycle function. We stress that all the results about our framework are independent of the machine model; hence, our results hold for generalized machine models as well.

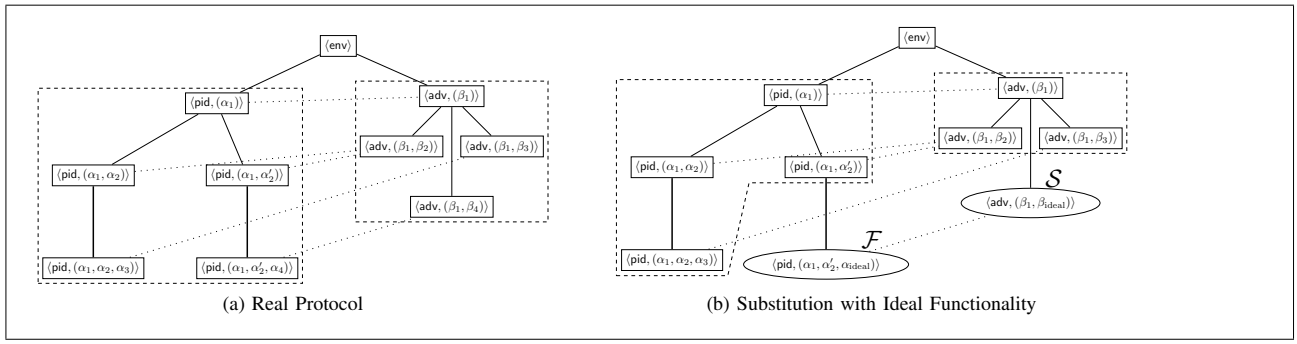
## IV. Secure Realization

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.

### *A. Security Definition*

In the same spirit as in other simulation-based frameworks, we adopt the notion of *secure realization*. A protocol $\pi$ is compared to a simplified protocol $\rho$ and is shown to be at least as secure: $\pi$ securely realizes $\rho$, if every attack against $\pi$ is also possible against $\rho$. More formally we require that the output distribution of the execution running the protocol $\pi$, an adversary $\mathcal{A}$ and an environment ENV is indistinguishable from the output distribution of the execution running the simplified protocol $\rho$ with a simulator $\mathcal{S}$ and the same environment ENV.

**Definition 13.** *A protocol* $\pi$ securely realizes *another protocol* $\rho$*, written* $\pi \geq_{\mathrm{t}} \rho$*, if for all* PPT *adversaries* $\mathcal{A}$ *there is a* PPT




(a) Real Protocol

(b) Substitution with Ideal Functionality

Figure 4: 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

*simulator $\mathcal{S}$ such that for all* PPT *environments* ENV

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

We stress that typically, the (distribution of the) speed coefficient of the realized protocol $\rho$ depends on the (distribution of the) speed coefficient of the realizing protocol $\pi$.

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

**Corollary 2.**

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

### *B. 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.

The most important design decisions with regard to showing these properties include making ENV and $\mathcal{A}$ timeless as well as having machines run with polynomially bounded speed coefficients (see Section III).

**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 sufficient 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.

**Lemma 1** (Completeness of the dummy adversary)**.** *If there exists an adversary $\mathcal{S}$ for a protocol $\Pi$ such that for all environments* ENV*,*

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

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

**Composition theorem.** The central building block of simulation-based security is the notion of *composability*: the composition of secure protocols is secure as well. Composition of protocols here is defined by substituting a sub-protocol $\Pi'$ of the protocol $\Pi$ used in the execution with another sub-protocol $\Pi'_1$.

We denote with $\Pi \setminus x$ the protocol which contains all protocol names that are reachable from the root protocol name $r$ of $\Pi$ without going through a node with basename $x$.

**Definition 14.** *Let $\Pi' = \Pi|x$ be a sub-protocol of $\Pi$ and let $\Pi'_1$ be a protocol rooted at $x$. $\Pi'_1$ is* substitutable *for $\Pi'$ if for all $y \in D(\Pi \setminus x)$ it holds that $\Pi(y) = \Pi'_1(y)$*

We denote the substitution of $\Pi'$ in $\Pi$ as $\Pi_1 = \Pi[\Pi'/\Pi'_1]$. That is, $\Pi_1|x = \Pi'_1$ and $\Pi_1 \setminus x = \Pi \setminus x$. Figure 4 illustrates protocol substitution in our model.

Using this notion of substitution, we can show that the composition of securely realized protocols yields securely realized protocols in TUC as well.

**Theorem 2.** *Let $\Pi$ be a protocol and $\Pi' = \Pi|x$ a sub-protocol of $\Pi$ rooted at $x$. Suppose that $\Pi'_1$ is a protocol rooted at $x$ such that $\Pi'_1 \geq_{\mathsf{t}} \Pi'$. Then*

$$\Pi[\Pi'/\Pi'_1] \geq_{\mathsf{t}} \Pi.$$

### *C. Ideal Functionalities*

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

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 (via shared memory). We adopt the restriction from GNUC that ideal machines can only communicate with ideal peers in the network and that ideal machines upon a compromise-message do not reveal their entire internal state to the network adversary $\mathcal{A}$ but can see the compromise-message in plain, i.e., as a normal message. Depending on the code of the ideal functionality protocol, the ideal machine then, e.g., just marks a party as compromised or sends sensitive information to the network adversary $\mathcal{A}$.

**upon input** $(\mathsf{createcircuit}, \mathcal{P} = \langle P, P_1, \ldots, P_\ell \rangle)$
1: set $Start$ to the current time
2: store $\mathcal{P}$ and $\mathcal{C} \leftarrow \langle P \rangle$
3: let $t \leftarrow d_3$
4: $ExtendCircuit(\mathcal{P}, \mathcal{C}, t)$

**upon receiving a handle** $(P', P, h)$ **from the network**
1: **if** $(P, cid, \mathsf{relay}, (\mathsf{data}, m)) = lookup(h)$ and $P$ is the OP for $cid$ **then**
2: lookup the current $t$; let $t' \leftarrow d_6$
3: **if** $prev(cid) = \perp$ **then**
4: output $(\mathsf{received}, cid, m)$ at time $t + t'$

**upon input** $(\mathsf{send}, \mathcal{C} = \langle P \overset{cid_1}{\Longleftrightarrow} P_1 \Longleftrightarrow \cdots P_\ell \rangle, m)$
1: **if** current time $- Start(cid_1) \leq ttl_C$ **then**
2: let $t \leftarrow d_4$
3: $SendMessage(P_1, cid_1, \mathsf{relay}, (\mathsf{data}, m), t)$
4: **else**
5: $DestroyCircuit(\mathcal{C}, cid_1)$
6: lookup the current time $t$
7: let $t' \leftarrow d_5$
8: output $(\mathsf{destroyed}, \mathcal{C}, m)$ at time $t + t'$

Figure 5: The ideal functionality $\mathcal{F}_{\text{OR}}^{\mathcal{N}}$ (short $\mathcal{F}_{\text{OR}}$) for Machine $P$: Client

Previous work [23], [25], [26] 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 V). A discussion about the equivalence of distributed and centralized ideal functionalities can be found in the extended version of this paper [45].

**Shared memory.** As mentioned above, we have to equip ideal functionalities with additional capabilities in order to be able to successfully abstract from more complex protocols. 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.

In the next section we present the application of TUC to analyzing the onion routing protocol used in Tor.

## V. Application

In this section we show how TUC can be used to analyze Tor to provide formal anonymity guarantees, even against time-sensitive adversaries. We first introduce the formalization of the onion routing (OR) protocol [28] that is used in Tor, which is an extension of an existing formalization presented by Backes et al. [3]. We then provide an overview about prominent timing-features of internet traffic that are used in traffic analysis attacks on Tor and show how they are represented in TUC. We conclude by proposing a countermeasure against website-fingerprinting attacks and by proving that this countermeasure provides k-anonymity.

### A. Overview of the Onion Routing Network Tor

In Tor messages are sent using a technique called onion routing where the message is routed via three proxies, called onion routers. On a regular basis (typically every 10 minutes), the client chooses the onion routers and establishes a ephemeral shared key with each of the onion routers. The routers together with the ephemeral keys form a so-called circuit. The client then sends a layered encryptions of the messages via the current circuit. The anonymity guarantee of Tor follows from the fact that every onion router in a circuit only knows its predecessor and its successor. However, as a low-latency protocol, Tor is prone to all kinds of traffic pattern attacks, such as traffic correlation or website fingerprinting.

**Clients, entry nodes, exit nodes, and entry links.** For illustration, we partition the Tor network into *clients nodes*, called onion proxies, *entry nodes*, i.e., nodes that are connected to a client node, and *exit nodes*, i.e., nodes that communicate with web servers outside of the Tor network. With such a partitioning an *entry link* is an edge between a client node and an entry node.

### B. Formalization of Onion Routing

We formalize the onion routing protocol that underlies Tor as a protocol $\Pi_{\text{OR}}$ in TUC, and we prove that it securely realizes (see Definition 13) an ideal functionality $\mathcal{F}_{\text{OR}}$. This ideal functionality abstracts from all cryptographic operations in $\Pi_{\text{OR}}$, stores messages in a shared memory, and sends handles for these messages through the network. Secure realization implies that all traffic-related timing attacks, such as traffic correlation or website fingerprinting, that are successful against $\Pi_{\text{OR}}$, are successful against $\mathcal{F}_{\text{OR}}$ as well. Thus, we use $\mathcal{F}_{\text{OR}}$ for proving anonymity guarantees for $\Pi_{\text{OR}}$.

**Ideal onion routing $\mathcal{F}_{\text{OR}}$.** The abstraction $\mathcal{F}_{\text{OR}}$, the protocol $\Pi_{\text{OR}}$, and the realization proof tightly follow previous work [3]. In contrast to previous work, however, the construction for the realization proof in TUC has to compensate for differences in computation times of the real protocol $\Pi_{\text{OR}}$ and the ideal functionality $\mathcal{F}_{\text{OR}}$. We achieve this compensation by requiring that the number of computation step each cryptographic operation in $\Pi_{\text{OR}}$ causes is predictable[3]. Furthermore, we make use of TUC's delayed sending command, which allows the ideal functionality to pre-compute the time at which $\Pi_{\text{OR}}$ would have sent a message and send its messages for exactly the same time.[4] We write our protocol specification reactively (using **upon**). Technically, the protocol transitions into a $(\mathsf{listen}, \infty)$ state after each **upon** command is processed.

Due to space constraints, and since the formulation of $\Pi_{\text{OR}}$, $\mathcal{F}_{\text{OR}}$ and the realization proof closely follow previous work, we refer to the full version of this paper for a detailed description [45, Section 5]. For illustration purposes we solely depict the client-subprotocol of $\mathcal{F}_{\text{OR}}$ (See Figure 5).

The client expects two types of commands from the environment, which in this case represents the operating system: $(\mathsf{send}, \mathcal{C}, m)$ and $(\mathsf{createcircuit}, \mathcal{P})$. Upon input $(\mathsf{send}, \mathcal{C}, m)$, $\mathcal{F}_{\text{OR}}$ checks whether the time-to-live of the current circuit is already exceeded. If not, this circuit is used for sending the message $m$, calling the subroutine $SendMessage$. This subroutine stores the message in the shared memory and creates a handle that is then sent over the network to the next onion router in the circuit. If the time-to-live of the current circuit is exceeded, the circuit is destroyed, calling the subroutine $DestroyCircuit$, which in turn uses $SendMessage$ to send destroy-messages to the onion routers in the circuit.

Upon input $(\mathsf{createcircuit}, \mathcal{P})$, $\mathcal{F}_{\text{OR}}$ creates a new circuit.

Upon a receiving a handle $(P', P, h)$ from the network, $\mathcal{F}_{\text{OR}}$ checks looks up the information for $h$ in the shared memory and checks whether $P$ indeed created the circuit. If so, it computes the delay $d[6]$ (for being synchronized with $\Pi_{\text{OR}}$) and outputs the message $m$ and the circuit id $cid$.

We show that $\Pi_{\text{OR}}$ securely realizes $\mathcal{F}_{\text{OR}}$ in a setting with secure channels, modeled as $\mathcal{F}_{\text{SCS}}$, and a public-key infrastructure, modeled as $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ for some delay vector $v$ (which depends on the speed coefficients of $\mathcal{F}_{\text{OR}}$ and $\Pi_{\text{OR}}$) that quantifies how long the ideal functionality has to wait to produce an indistinguishable input/output pattern (from $\Pi_{\text{OR}}$). The proof tightly follows previous work [3].

[3]Formally, our abstraction $\mathcal{F}_{\text{OR}}$ is thereby parametric in the cryptographic implementation. Since we use our ideal functionality for the analysis of a concrete protocol, we do not consider this limitation a severe restriction.

[4]We stress that at this point, we need that the protocol code can depend on the speed coefficient.

**Theorem 3.** *$\Pi_{\text{OR}}$ securely realizes $\mathcal{F}_{\text{OR}}$ in the $\mathcal{F}_{\text{SCS}}, \mathcal{F}_{\text{REG}}^{\mathcal{N}}-$ hybrid model for some delay vector $v$.*

**Protocol interface $\Pi_{\text{WOR}}$.** The protocol $\Pi_{\text{OR}}$ (and thus also $\mathcal{F}_{\text{OR}}$) model the basic onion routing protocol for single message blocks. In order to have a more convenient protocol interface, we introduce a wrapper $\Pi_{\text{WOR}}$ (see Figure 6) that performs the circuit construction, splits messages into message blocks and sends these message blocks over a subprotocol (in our case $\Pi_{\text{OR}}$ or $\mathcal{F}_{\text{OR}}$), and re-assembles the messages blocks received from the subprotocol. For the sake of brevity, we omit commands, such as setup and destroyed, that are merely forwarded to the subprotocol.

$\Pi_{\text{WOR}}$ uses a uniform path selection algorithm $\mathrm{PathSelection}$; however, note that by adjusting the distribution used in $\mathrm{PathSelection}$, any path selection algorithm can be utilized.

**Re-assembling and splitting in $\Pi_{\text{WOR}}$.** The stateful routine $\mathrm{Reassemble}(m, s)$ expects as input a message block $m$ (and a state $s$) and outputs, together with a new state $s'$, either a dummy message ready, if a complete message could not be reassembled yet, or a re-assembled message $m' \neq \mathsf{ready}$, if $m$ and the state $s$ allowed re-assembling a complete message. The current state of $\mathrm{Reassemble}$ inside an instance of $\Pi_{\text{WOR}}$ is saved in the variable $s$. If looking up $s$ fails, we assign the empty state, i.e., the empty string, to $s$.

Dual to the re-assembling routine, $\Pi_{\text{WOR}}$ contains the routine $\mathrm{Split}(m)$, which splits a message $m$ into message blocks $m_i$ of length $blockln$ and potentially pads the last block.

As a corollary of Theorem 2, we conclude that it suffices to analyze $\Pi_{\text{WOR}\mathcal{F}_{\text{OR}}}$ instead of $\Pi_{\text{WOR}\Pi_{\text{OR}}}$.

**Corollary 3.** *$\Pi_{\text{WOR}\Pi_{\text{OR}}}$ securely realizes $\Pi_{\text{WOR}\mathcal{F}_{\text{OR}}}$ in the $\mathcal{F}_{\text{SCS}}, \mathcal{F}_{\text{REG}}^{\mathcal{N}}-$ hybrid model for some delay vector $v$.*

### C. 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 [47]. Due to space constraints, we use a simplified version of the anonymity notions presented therein.

*1) The Set-Up:* We consider the class of environments that consist of two sub-machines, an environment adversary $\mathcal{A}_{\text{ENV}}$ and the challenger $\mathrm{Ch}$. The environment adversary $\mathcal{A}_{\text{ENV}}$ 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

**upon a message** $(\mathsf{received}, (m, cid))$ **from** $\rho$
1: lookup the state $s$ of Reassemble for $\mathsf{sid}'$
2: call $(m', s') \leftarrow \text{Reassemble}(m, s)$; store $s'$ for $\mathsf{sid}'$
3: **if** $\mathsf{ready} \neq m'$ **then**
4: send $(\mathsf{received}, m')$ to parent

**upon a message** $(\mathsf{exit}, (m, \mathsf{sid}))$ **from** $\rho$
1: lookup the state $s$ of Reassemble for $\mathsf{sid}'$
2: call $(m', s') \leftarrow \text{Reassemble}(m, s)$; store $s'$ for $\mathsf{sid}'$
3: **if** $\mathsf{ready} \neq m' = (m'', \mathsf{sid}'', a)$ and $a$ is a server **then**
4: send $(\mathsf{exit}, (m'', \mathsf{sid}'', a), \mathsf{sid})$ to parent

**upon input** $(\mathsf{response}, (\mathsf{sid}, m))$ **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 $(\mathsf{response}, \mathsf{sid}, m_i)$ to $\rho$

**upon input** $(\mathsf{send}, m)$ **from parent**
1: $\mathcal{P} \leftarrow \text{PathSelection}(P_u)$
2: **if** if there is no open circuit $cid$ **then**
3: send message $(\mathsf{createcircuit}, \mathcal{P})$ to $P_u$
4: wait for response $(\mathsf{created}, \mathcal{C})$
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 $(\mathsf{send}, \mathcal{C}, m_i)$ to $\rho$

Figure 6: Wrapper $\Pi_{\text{WOR}\rho}$ for client $P_u$ and a sub-protocol $\rho$

collaboration of the environment adversary and the network adversary. By the completeness of the dummy adversary (Lemma 1) it suffices to consider the network dummy adversary and the environment adversary.

Each party consists of a the $\Pi_{\text{WOR}}$ protocol with the onion routing protocol as a child. By Corollary 3, we directly consider the ideal functionality $\mathcal{F}_{\text{OR}}$ instead of the onion routing protocol $\Pi_{\text{OR}}$ ($\mathcal{F}_{\text{OR}}$ is depicted in Figure 5).

This set-up is exemplified in Figure 7. It depicts three instances of $\mathcal{F}_{\text{OR}}$ together with the wrapper $\Pi_{\text{WOR}}$, the shared memory MEM used by $\mathcal{F}_{\text{OR}}$, a challenger Ch, the network adversary and the environment adversary $\mathcal{A}_{\text{ENV}}$.

**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 7). In this sender anonymity game the environment adversary $\mathcal{A}_{\text{ENV}}$ has to determine the sender of a specific message-stream in the presence of noise, i.e., other message-streams. $\mathcal{A}_{\text{ENV}}$ has to link at least one session to the correct sender address. Recall that $\mathcal{A}_{\text{ENV}}$ 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 $\mathsf{UM}_S$ 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 $\mathsf{UM}_S$ 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 $\mathsf{fresh}$ 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 $\mathsf{UM}_S$. SACh forwards each message $msg_i$ from $\mathsf{UM}_S$ 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 $\mathsf{UM}_S$ 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 $\mathcal{A}_{\text{ENV}}$. Finally, upon an input $(\mathsf{guess}, (P', S'))$, SACh checks whether the user $u$ is assigned to the user model $\mathsf{UM}_S$ 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., $\forall i, j. O_i \neq P_j$.

*2) Mounting Attacks that use Timing Features:* In TUC timing-based traffic features can be measured by the adversary. Example 2 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 [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20].

**Example 2: 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 $(\mathsf{send}, m)$ to party $P$ at time $T$
2: $\Pi_{\text{WOR}}$ in $P$ receives $(\mathsf{send}, m)$ at $T' = \frac{k}{c_P} \geq T$, where $T'$ is the first time after $T$ where $\Pi_{\text{WOR}}$ is in the $\mathsf{listen}$ state
3: Let $n$ be the number of steps needed to split $m$ into $q := \lceil |m| / blockln \rceil$ packets $m_1, \ldots, m_q$.
$\Pi_{\text{WOR}}$ sends $m_i$ at time $T_i = T' + \frac{n+i}{c_P}$ to $\mathcal{F}_{\text{OR}}$
4: Let $n'$ be the time that $\mathcal{F}_{\text{OR}}$ needs to send a message and $d_4$ the delay for being synchronized with $\Pi_{\text{OR}}$.

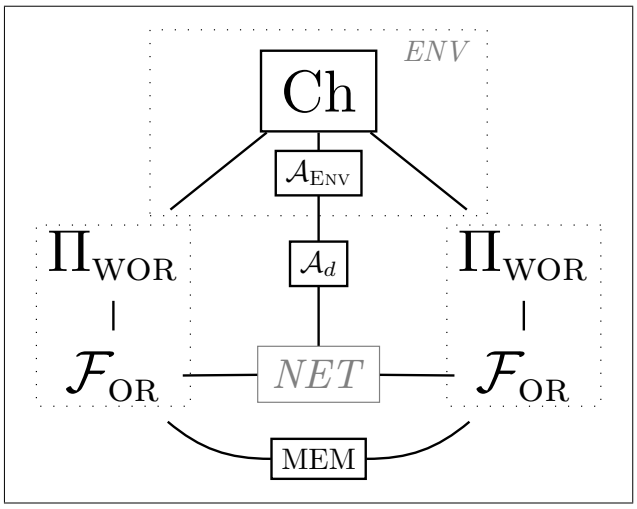



Figure 7: The Attack Scenario

$\mathcal{F}_{\text{OR}}$ forwards $m_i$ at time $T' + d_4 + \frac{n+1+i\cdot n'}{c_P} =: T' + g(c_P, i)$, since messages $m_2, \ldots, m_q$ arrive at $\mathcal{F}_{\text{OR}}$ before $n'/c_P$ time has passed.

The *traffic pattern* a user model $\mathsf{UM}_S$ 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 $\Pi_{\text{WOR}\,\mathcal{F}_{\text{OR}}}$ sends its message blocks.

If the network links through which the messages, or in the case of $\mathcal{F}_{\text{OR}}$ 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 [10], [11], [12], [20]).

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.[5] ◇

**Inter-packet delay.** Inter-packet delay is the time difference in the time stamps of two consecutive packets sent through a connection. Example 2 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 2, 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 2 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

[5]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.

**Exit node: upon** $(\mathsf{exit}, ((m'', \mathsf{sid}'', a), \mathsf{sid}'))$ **from** $\rho$

1: **if** $bun_{\mathsf{sid}''} = \perp \wedge a$ 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 $\mathsf{pageBuck}$
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 $(\mathsf{response}, (\mathsf{sid}', m'))$ to $\rho$ at time $t$

Figure 8: The protocol $\mathrm{WFC}_\rho$ for party pid, where sid is its session ID

can however be modeled by refining the network topology NET in the execution EXEC.

**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 [30], [31], [32] 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.

### *D. Analyzing a Countermeasure against Website Fingerprinting*

As a case study, we leverage our time-sensitive framework and our Tor abstraction to analyze a simple countermeasure against website fingerprinting and to 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 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 $t_{\mathrm{buf}}$ 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 $\mathsf{pageBuck}$ contains a bucket for each target web page size and upon input of a size $n$, $\mathsf{pageBuck}(n)$ returns the list of web pages that have size $n$ or are padded to size $n$. The countermeasure protocol WFC is depicted in Figure 8.

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

*1) The $k$-recipient anonymity challenger* $\mathrm{RACh}$*:* 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 V-C1) except that the challenger $\mathrm{Ch}$ is replaced by the following k-recipient anonymity challenger $\mathrm{RACh}$. This challenger $\mathrm{RACh}$ initially allows the environment adversary to define the page-buckets used in the game, but requires that each bucket in $\mathsf{pageBuck}$ contains exactly $k$ web pages.

In contrast to the sender anonymity game, $\mathrm{RACh}$ 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 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.

**upon** $(\mathsf{register\text{-}webpages}, \mathsf{pageBuck})$ **from** $\mathcal{A}_{\mathbf{ENV}}$

**for** all sizes $n$ **do**
  **if** $|\mathsf{pageBuck}(n)| = k$ **then**
    **for** all $(a, loc, pg) \in \mathsf{pageBuck}(n)$ **do**
      **if** $(a, loc)$ is unregistered and $loc \neq \mathsf{challenge}$ **then**
        send $(\mathsf{register}, loc, pg)$ to $\Pi_{\text{server}}$ at party $a$
  **else**
    return error

**upon** $(\mathsf{challenge}, n)$ **from** $\mathcal{A}_{\mathbf{ENV}}$

draw a random $j \in \{1, \ldots, k\}$
let $(a_j, loc_j, pg_j) \leftarrow \mathsf{pageBuck}(n)$
$(a_{ch}, loc_{ch}) \leftarrow (a_j, loc_j)$

**upon** $(\mathsf{send}, (a, loc))$ **from** $\mathcal{A}_{\mathbf{ENV}}$

in the first call: send $\mathsf{setup}$ to $\rho$; wait for $\mathsf{ready}$
**if** $loc = \mathsf{challenge}$ **then** $(a, loc) \leftarrow (a_{ch}, loc_{ch})$
send $(a, loc)$ to P

**upon** $(\mathsf{guess}, (a, loc))$ **from the** $\mathcal{A}_{\mathbf{ENV}}$

**if** $(a, loc) = (a_{ch}, loc_{ch})$ **then** output $(\mathsf{guess}, 1)$
**else** $(\mathsf{guess}, 0)$

Figure 9: $k$-recipient anonymity challenger: $\mathrm{RACh}_k$

**Server protocol** $\Pi_{\text{server}}$**.** Web servers are modeled as a protocol $\Pi_{\text{server}}$, which can be thought of an abstraction of server software, e.g., Apache HTTP Server.[6]

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, $\Pi_{\text{server}}$ 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 3), the definition of $\mathcal{F}_{\text{OR}}$ and from the fact that all webpages in the same bucket are padded to the same size and that WFC removes timing features from the traffic by introducing artificial delays.

**Lemma 2.** *Let* $\mathsf{EXEC}'$ *be defined as* $\mathsf{EXEC}$ *except that* $\mathsf{EXEC}'$ *outputs the bit* $b$ *from the first output of the form* $(\mathsf{guess}, b)$ *by* RACh*. Let* $\langle \mathrm{RACh}, \mathcal{A}_{\text{ENV}} \rangle$ *denote the machine that contains* RACh *and* $\mathcal{A}_{\text{ENV}}$*, as described in Section V-C1. Let* $\langle \mathrm{WFC}_{\Pi_{\text{WOR}}\Pi_{\text{OR}}}, \Pi_{\text{server}} \rangle$ *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* $\mathcal{A}_{\text{ENV}}$ *and a dummy network adversary* $\mathcal{A}_d$ *that only compromises entry links or entry nodes , for sufficiently large* $t_{\text{buf}}$ *and* $\eta$ *and a negligible function* $\mu$ *we have*

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

[6]Technically, $\Pi_{\text{server}}$ is the "server" role in the WFC protocol; otherwise they cannot be peers.

## VI. 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 [3] can be suitably extended to account for timing and that it is realized in TUC by a similarly extended onion routing protocol. We then leveraged this abstraction and our framework to formalize, as a case study, a simple 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 routing yields 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 extend existing models based on timed automata to better capture real world networks and adversaries, and explore in which cases timed automata are a sound abstraction for TUC protocols. Such a result would allow obtaining strong guarantees, i.e., against computational adversaries that can perform time-measurements, from established automated verification tools [40], [41], [39], [44], [43], [42].

Moreover, there is an information theoretic analysis of web traffic which uses an abstraction of web-traffic [48]. 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] "The Tor Project," https://www.torproject.org/, accessed May 2013.
[2] "Tor Metrics Portal," https://metrics.torproject.org/, accessed May 2013.
[3] M. Backes, I. Goldberg, A. Kate, and E. Mohammadi, "Provably Secure and Practical Onion Routing," in *Proceedings of the 25th IEEE Computer Security Foundations Symposium (CSF)*, 2012, pp. 369–385.
[4] J. Camenisch and A. Lysyanskaya, "A Formal Treatment of Onion Routing," in *Advances in Cryptology (CRYPTO)*, 2005, pp. 169–187.
[5] J. Feigenbaum, A. Johnson, and P. F. Syverson, "Probabilistic Analysis of Onion Routing in a Black-Box Model," *ACM Transactions on Information and Systems Security (TISSEC)*, vol. 15, no. 3, p. 14, 2012.

[6] P. F. Syverson, G. Tsudik, M. G. Reed, and C. E. Landwehr, "Towards an Analysis of Onion Routing Security," in *Designing Privacy Enhancing Technologies - International Workshop on Design Issues in Anonymity and Unobservability*, 2000, pp. 96–114.

[7] M. Backes, A. Kate, P. Manoharan, S. Meiser, and E. Mohammadi, "AnoA: A Framework for Analyzing Anonymous Communication Protocols," in *Proceedings of the 26th IEEE Computer Security Foundations Symposium (CSF)*, 2013, pp. 163–178.

[8] N. Gelernter and A. Herzberg, "On the limits of provable anonymity," in *Proceedings of the 12th ACM Workshop on Privacy in the Electronic Society (WPES)*, 2013, pp. 225–236.

[9] S. Chakravarty, A. Stavrou, and A. 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)*, 2010, pp. 249–267.

[10] N. S. Evans, R. Dingledine, and C. Grothoff, "A Practical Congestion Attack on Tor Using Long Paths," in *Proceedings of the 18th USENIX Security Symposium (USENIX)*, 2009, pp. 33–50.

[11] P. Mittal, A. Khurshid, J. Juen, M. Caesar, and N. 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, pp. 215–226.

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

[13] Y. Gilad and A. Herzberg, "Spying in the Dark: TCP and Tor Traffic Analysis," in *Proceedings of the 12th Privacy Enhancing Technologies Symposiun (PETS)*, 2012, pp. 100–119.

[14] N. Hopper, E. Y. Vasserman, and E. Chan-Tin, "How much anonymity does network latency leak?" *ACM Transactions on Information and Systems Security (TISSEC)*, vol. 13, no. 2, p. 13, 2010.

[15] Z. Ling, J. Luo, Y. Zhang, M. Yang, X. Fu, and W. 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)*, 2012, pp. 2390–2398.

[16] S. J. Murdoch and P. Zieliński, "Sampled Traffic Analysis by Internet-Exchange-Level Adversaries," in *Proceedings of the 7th Workshop on Privacy Enhancing Technologies (PET)*, 2007, pp. 167–183.

[17] G. O'Gorman and S. Blott, "Improving Stream Correlation Attacks on Anonymous Networks," in *Proceedings of the 24th ACM Symposium on Applied Computing (SAC)*, 2009, pp. 2024–2028.

[18] L. Øverlier and P. F. Syverson, "Locating Hidden Servers," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P)*, 2006, pp. 100–114.

[19] X. Wang, D. S. Reeves, and S. F. 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)*, 2002, pp. 244–263.

[20] A. Houmansadr and N. Borisov, "The Need for Flow Fingerprints to Link Correlated Network Flows," in *Proceedings of the 13th Privacy Enhancing Technologies Symposium (PETS)*, 2013.

[21] O. Goldreich, S. M. Micali, and A. Wigderson, "How to Play ANY Mental Game," in *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, 1987, pp. 218–229.

[22] M. Backes, B. Pfitzmann, and M. Waidner, "The Reactive Simulatability (RSIM) Framework for Asynchronous Systems," *Information and Computation*, vol. 205, no. 12, pp. 1685–1720, 2007.

[23] R. Canetti, "Universally Composable Security: A New Paradigm for Cryptographic Protocols," Cryptology ePrint Archive, Report 2000/067, 2013.

[24] A. Datta, R. Küsters, J. C. Mitchell, and A. Ramanathan, "On the Relationships Between Notions of Simulation-Based Security," in *Proceedings of the 2nd Conference of the Theory of Cryptography (TCC)*, 2005, pp. 476–494.

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

[26] R. Küsters and M. Tuengerthal, "The IITM Model: a Simple and Expressive Model for Universal Composability," IACR Cryptology ePrint Archive, Report 2013/025, 2013.

[27] 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)*, 2003, pp. 327–349.

[28] D. M. Goldschlag, M. G. Reed, and P. F. Syverson, "Onion Routing," *Communications of the ACM (CACM)*, vol. 42, no. 2, pp. 39–41, 1999.

[29] J. Feigenbaum, A. Johnson, and P. F. Syverson, "A Model of Onion Routing with Provable Anonymity," in *Proceedings of the 11th International Conference on Financial Cryptography and Data Security (FC)*, 2007, pp. 57–71.

[30] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson, "Touching From a Distance: Website Fingerprinting Attacks and Defenses," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012, pp. 605–616.

[31] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel, "Website Fingerprinting in Onion Routing based Anonymization Networks," in *Proceedings of the 10th ACM Workshop on Privacy in the Electronic Society (WPES)*, 2011, pp. 103–114.

[32] X. Gong, N. Kiyavash, and N. Borisov, "Fingerprinting Websites using Remote Traffic Analysis," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010, pp. 684–686.

[33] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. 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)*, 2012, pp. 332–346.

[34] Y. T. Kalai, Y. Lindell, and M. Prabhakaran, "Concurrent Composition of Secure Protocols in the Timing Model," *Journal of Cryptology*, vol. 20, no. 4, pp. 431–492, 2007.

[35] M. Backes, "Unifying Simulatability Definitions in Cryptographic Systems under Different Timing Assumptions," *Journal of Logic and Algebraic Programming (JLAP)*, vol. 2, pp. 157–188, 2005.

[36] J. Katz, U. Maurer, B. Tackmann, and V. Zikas, "Universally Composable Synchronous Computation," in *Proceedings of the 10th Theory of Cryptography Conference (TCC)*, 2013, pp. 477–498.

[37] J. B. Nielsen, "On Protocol Security in the Cryptographic Model," dissertation, University of Aarhus, 2003.

[38] B. Pfitzmann, M. Schunter, and M. Waidner, "Cryptographic Security of Reactive Systems: (Extended Abstract)," *Electronic Notes in Theoretical Computer Science*, vol. 32, pp. 59–77, 2000.

[39] L. Bozga, C. Ene, and Y. Lakhnech, "A Symbolic Decision Procedure for Cryptographic Protocols with Time Stamps," *Journal of Logic and Algebraic Programming (JLAP)*, vol. 65, pp. 1–35, 2005.

[40] R. Corin, S. Etalle, P. H. Hartel, and A. Mader, "Timed Model Checking of Security Protocols," in *Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering (FMSE)*, 2004, pp. 23–32.

[41] G. Jakubowska and W. Penczek, "Is Your Security Protocol on Time?" in *Proceedings of the 2007 International Symposium on Fundamentals of Software Engineering (FSEN)*, 2007, pp. 65–80.

[42] M. Kurkowski, W. Penczek, and A. 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 (MoChArt)*, vol. 4428, 2007, pp. 146–165.

[43] M. Kurkowski and W. Penczek, "Timed Automata Based Model Checking of Timed Security Protocols," *Fundamenta Informaticae - Concurrency, Specification and Programming*, vol. 93, no. 1-3, pp. 245–259, 2009.

[44] R. Lanotte, A. Maggiolo-Schettini, and A. Troina, "Time and Probability-Based Information Flow Analysis," *IEEE Transactions on Software Engineering (TSE)*, vol. 36, no. 5, pp. 719–734, 2010.

[45] M. Backes, P. Manoharan, and E. Mohammadi, "TUC: Time-sensitive and Modular Analysis of Anonymous Communication," Cryptology ePrint Archive, Report 2013/664, 2013.

[46] S. A. Cook and R. A. Reckhow, "Time-bounded random access machines," in *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing (STOC)*. ACM, 1972, pp. 73–80.

[47] M. Backes, A. Kate, P. Manoharan, S. Meiser, and E. Mohammadi, "AnoA: A Framework For Analyzing Anonymous Communication Protocols," Cryptology ePrint Archive, Report 2014/087, 2014.

[48] M. Backes, G. Doychev, and B. Köpf, "Preventing Side-Channel Leaks in Web Traffic: A Formal Approach," in *Proceedings of the 20th Network and Distributed Systems Security Symposium (NDSS)*, 2013.