# Moving Participants Turtle Consensus*

Stavros Nikolaou[1] and Robbert van Renesse[2]

1 Department of Computer Science, Cornell University, Ithaca, NY, US
snikolaou@cs.cornell.edu

2 Department of Computer Science, Cornell University, Ithaca, NY, US
rvr@cs.cornell.edu

## Abstract

We present Moving Participants Turtle Consensus (MPTC), an asynchronous consensus protocol for crash and Byzantine-tolerant distributed systems. MPTC uses various *moving target defense* strategies to tolerate certain Denial-of-Service (DoS) attacks issued by an adversary capable of compromising a bounded portion of the system. MPTC supports on the fly reconfiguration of the consensus strategy as well as of the processes executing this strategy when solving the problem of agreement. It uses existing cryptographic techniques to ensure that reconfiguration takes place in an unpredictable fashion thus eliminating the adversary's advantage on predicting protocol and execution-specific information that can be used against the protocol.

We implement MPTC as well as a State Machine Replication protocol and evaluate our design under different attack scenarios. Our evaluation shows that MPTC approximates best case scenario performance even under a well-coordinated DoS attack.

**1998 ACM Subject Classification** D.4.5 Fault-Tolerance

**Keywords and phrases** Consensus, adaptation, moving target defense

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2016.20

## 1 Introduction

Most distributed systems today are designed to tolerate failures. Existing fault-tolerance methods typically assume that failures are rare. They are tailored to provide good performance when no failures occur but might perform poorly under failure scenarios. However, as shown in works like [11], such designs allow malicious adversaries to craft workloads and Denial-of-Service (DoS) attacks that can substantially degrade the performance of certain state-of-the-art fault-tolerance protocols. As such DoS attacks become more common, it is becoming increasingly important to design fault-tolerance mechanisms that perform well in good scenarios while also gracefully handle adversarial ones. A core building block of many of these mechanisms are consensus protocols used by a set of replicas to agree on some state. One way to improve existing fault tolerance solutions is by enhancing the underlying consensus protocols with reconfiguration capabilities that allow them to change their execution parameters on the fly in order to better deal with adversarial workloads.

Our prior work on *Turtle Consensus* [22] also aims at attack-tolerant consensus. Turtle Consensus is a round-based consensus protocol that operates by using different consensus strategies across different rounds. The system's processes try to reach agreement running a

* This work was partially supported by AFOSR DURIP grant FA2386-12-1-3008, by NSF grants CCF-1047540, CNS-1040689, CNS-1422544, CNS-1561209, CNS-1601879, by a Google Faculty Research Award, by MDCN/iAd grant 54083, and by gifts from Infosys, Facebook, and Amazon.com. The authors would also like to thank the anonymous reviewers.

round of some consensus protocol in the literature; if they fail to do so they move onto the next round using a different protocol. The selection of each round's strategy is predetermined and known to all processes running the protocol. The main strategy of Turtle Consensus is to use the best approach available for normal operation in a particular setting and switch to different "backup protocols" as soon as the approach becomes inefficient, for example in the case of a DoS attack. We showed that the approach used sub-optimal strategies during DoS attacks, thus bounding the protocol's efficiency to the capabilities of these backup protocols. In addition, an adversary capable of compromising even a single consensus participant can learn and use the predetermined nature of the protocols' succession to constantly drive the system to sub-optimal executions.

In this paper we address these concerns by adding another degree of freedom in the reconfiguration capabilities of Turtle Consensus. We present Moving Participants Turtle Consensus (MPTC), an extension to the Turtle Consensus protocol that allows switching not only the protocols but also the set of processes on which they run across different rounds of a single consensus instance. The consensus protocol round and the processes participating in its execution form what we call a *configuration*, which our approach changes unpredictably at each round. While the configuration selection for each round is predetermined by a trusted dealer, it is unknown to the processes during MPTC execution. Using existing cryptographic techniques, we ensure that, only if sufficiently many processes collaborate during some round, the next round's configuration can be determined. This renders MPTC a valuable tool for building systems that can tolerate DoS attacks in crash-tolerant environments where a bounded portion of the system may be compromised. An extended version of this work that also handles Byzantine failures appears in [23].

## 2 Model

### 2.1 Processes and communication

Our system consists of a set of processes $\mathcal{N}$ that communicate using message passing. Each process is modeled as a state machine with a potentially unbounded set of states that executes transitions according to some protocol. The protocol specifies the transition function of the processes as well as the messages they exchange. Each process's state consists of two components, the *public* and the *private* or *secret state*. The public state contains the description of the protocol that each process executes and any public cryptographic keys associated with the process. The secret state contains any run-time state the process manages during the execution of the protocol as well as any secret cryptographic keys and/or shares associated with the process.

Protocol execution and communication are *asynchronous*, meaning that there are no bounds on the time it takes processes to execute transitions and deliver messages.

A process can be correct or faulty. A correct process faithfully executes the protocol and is guaranteed to make progress as long as the conditions specified by the protocol at any given step are eventually met. In this paper we only consider crash failures, although we extend our techniques to Byzantine failures in [23]. A faulty process may crash at any time after which it stops executing the protocol. Up to the point of the crash, processes faithfully follow the steps of the protocol. Communication between correct processes is reliable and secure. This means that, in the absence of a DoS attack (see below), messages sent by some correct process to another correct process are eventually delivered. It also means that messages are authenticated and cannot be tampered with or fabricated. We assume an upper bound, $f_c < |\mathcal{N}|$, on the total number of processes that might fail during the protocol's execution.

Finally, we assume the existence of a special process $T \notin \mathcal{N}$ that from now on we will refer to as *trusted dealer* or simply dealer. The dealer is only used during initialization of the system during which it generates the initial public and secret state of all processes. We assume that during this setup phase the dealer is correct, that it can communicate via private channels with any process in the system, and that it does not disclose its state. After initialization, however, the dealer does not execute any protocol steps or exchange messages with any other process, and the dealer's state is destroyed.

## 2.2 Adversary and attacks

We assume an adversary, $A$, that controls which processes fail and when. $A$ is limited on the number of processes it can fail by $f_c$ and cannot fail the dealer. $A$ can also control the delivery order of messages of all processes as well as delay communication, but must yield to the previously stated reliable communication assumption.

The adversary can also issue *denial of service* (DoS) attacks against the system that can fully saturate the bandwidth resources of at most $f_a < |\mathcal{N}|$ correct processes. This can effectively prevent the targeted processes from progressing in the protocol's execution since they can no longer communicate with the rest of the system. $A$ can change the targets of an attack over time and, in this way, can introduce communication and computation delays on certain processes. The adversary's objective is to prevent the system from making progress. From now on we will denote by $f$ the maximum number of processes that can be crashed or under attack during the execution of the protocol, that is $f = f_c + f_a < |\mathcal{N}|$.

In this work, we ignore DoS attacks that target other resources like CPU using legitimate traffic. These attacks can be mitigated using rate limiting techniques such as client cryptographic puzzles [2].

The adversary has read access to the public state of all processes as well as the secret state of up to $f$ processes. We call the processes whose secret state is disclosed to $A$ *compromised*. While $A$ cannot modify this state, it can use this state to select the target processes of a DoS attack. Once $A$ has selected the set of compromised processes it can no longer change that set thus preventing $A$ from accessing the secret state of more than $f$ processes. Note that the set of compromised processes is not necessarily related to the set of processes that are crashed or under attack.

Finally, we assume that the various cryptographic schemes we are employing, like public key cryptography and threshold signatures, are secure in the random oracle model.

## 2.3 Cryptographic primitives

Our protocol relies on Threshold Coin Tossing [5]. Here we present a high-level description of this primitive that we will further formalize in Section 3. We employ an $(n, f+1, f)$ threshold coin-tossing scheme in which $n$ parties maintain shares of an unpredictable function, $F$, mapping an arbitrary bit string, $r$, to a binary value $\{0, 1\}$. Each of these shares can be used along with an input $r$ to create a value that from now on we will refer to as *function shares*[1]. At least $f+1$ of these function shares of $r$ are required to reconstruct the result $F(r)$, while at most $f$ parties may be compromised. We will use the term *function share of* $F(r)$ to denote a function share of $r$ generated with a secret share of $F$.

The scheme defines three functions: 1) The *split* function, which takes as input a function $F$ (represented as a bit string) and creates a set of shares as well as a verification key for

[1] The term used in [5] for these values is coin shares.

each of these shares. 2) The share combining function, *combine*, which takes an input $r$ of $F$ along with $f+1$ valid function shares of $r$ and produces $F(r)$. 3) The share verification function *verify*, which takes an input $r$ of $F$, a function share on $r$, and the verification key corresponding to the share that generated the input function share and determines whether the function share is valid.

This scheme is based on threshold signatures [26] and can be used to create an unpredictable sequence of bits while ensuring that it is computationally infeasible for the adversary to produce an input $r$ and $f+1$ valid function shares that once combined do not yield $F(r)$. More formally, the scheme satisfies the following properties taken from [5]:

- *Robustness*: It is computationally infeasible for the adversary to produce a value $r$ and $f+1$ valid shares of $r$ such that the result of the combine function is not $F(r)$.
- *Unpredictability*: Given a value $r$ and functions shares from fewer than $f+1-f$ correct processes, the adversary can predict the value of $F(r)$ with probability at most $\frac{1}{2}+\epsilon$ for negligible value $\epsilon \in \mathbb{R}$.

The previous unpredictability property was extended to sequences of output bits in [5], such that, given a sequence of values $C_i$ for $i \in \{1, 2, \dots, b\}$, an adversary with fewer than $f+1-f$ valid shares of some $C_i$ has negligible advantage in predicting $F(C_i)$. From now on, when we talk about unpredictability we will refer to this *extended unpredictability property* of threshold coin-tossing.

Note that the previously described extended unpredictability property allows us to share unpredictable functions in $[\{0,1\}^* \to \{0,1\}^b]$ for any finite $b$. In other words, we can model each such function as a random number generator that can produce $2^b$ different values and requires $f+1$ processes to collaborate in order to produce the random (unpredictable) value corresponding to some arbitrary bit string $r$.

Threshold coin-tossing can be implemented using any non-interactive threshold signatures scheme that ensures unique valid signature per message as in [26]. A direct implementation of this scheme can be found in [5].

## 2.4 Underlying consensus protocols

MPTC, like other consensus protocols, solves the problem of agreement. In this problem, a set of possibly distributed processes, each of which is initialized with some input value, unanimously and irrevocably output one of those input values. More formally, let $\mathcal{N}$ be a set of processes each of which is initialized with some value from a value set $\mathcal{V}$. Each process can employ either of the following primitives:

- *propose* a value which allows a process to communicate its value to the rest of the processes in $\mathcal{N}$,
- *decide* a value which allows a process to output a value.

Every correct consensus protocol must satisfy the following properties:

- *Validity*: If a process decides a value, then that value must be the input value of some process in $\mathcal{N}$.
- *Agreement*: If any two processes decide they must decide the same value.
- *Termination*: All correct processes eventually decide.

[13] has shown that in an asynchronous environment no consensus protocol exists that satisfies all of the above properties when even only a single failure can occur. To circumvent this result, a variety of protocols have been proposed [3, 10] that use a probabilistic approach and can guarantee the previous properties with the following modification on termination:

*All correct processes eventually decide with probability 1.* For the remainder of this work we will refer to the non-probabilistic description of termination as *definite termination* and to the probabilistic one as *probabilistic termination.*

A consensus protocol that implements the previous specification (using either definite or probabilistic termination) even under the presence of $f$ crash failures is called $f$-crash-resilient. Note that our adversary can additionally perform denial-of-service attacks which can fully saturate a bounded number of processes and render them entirely unavailable. In an asynchronous environment there is no difference between a crashed process and a process that is under DoS attack from the other processes' perspective. For this reason we say that a consensus protocol is correct in our model if it is $f$-crash-resilient where $f = f_c + f_a$. From now on we will refer to such consensus protocols as $f$-resilient protocols.

Each process executing MPTC may run different consensus protocols at different rounds. We denote the set of possible protocols each process can choose from by $\mathcal{P}$. Different consensus protocols make different assumptions under which they meet the previously described specification. The crash-tolerant consensus protocol of Ben-Or [3], for instance, assumes an asynchronous environment and that each infinite schedule has a bounded number of processes performing a finite number of steps. Other protocols make assumptions such as bounds on the number of failures, different degrees of synchrony, the existence of failure detectors [8], etc. We consider a consensus protocol correct if it satisfies agreement, validity and either definite or probabilistic termination. For each protocol $P \in \mathcal{P}$, we denote the set of assumptions required to hold for $P$ to be correct by $\mathcal{A}_P$. In other words, if assumptions $\mathcal{A}_P$ hold, then $P$ satisfies validity, agreement, and termination. A protocol $P$ is a valid candidate for $\mathcal{P}$ if it is correct under both the assumptions in $\mathcal{A}_P$ and our previous model assumptions regarding failures, network reliability, and adversary.

We only consider consensus protocols operating in rounds and we follow the framework introduced in [22] for the specification of the round outcomes. According to this specification, every process running a round of a consensus protocol ends up in one of the following states: $\{D, U, M\} \times \mathcal{V}$, where states $(D, v), v \in \mathcal{V}$ indicate that the process has decided $v$, states $(U, v), v \in \mathcal{V}$ indicate that no process has decided up to the current round, and finally, states $(M, v), v \in \mathcal{V}$ indicate that while the process is not decided, if a decision was made by some process then it must have been $v$. We will refer to these states as round outcomes or simply *outcomes*. We denote by $o_p^r$ the outcome of process $p \in \mathcal{N}$ at the end of round $r \in \mathbb{N}$.

More formally the following invariants hold about the outcomes of processes completing a round of a correct consensus protocol in $\mathcal{P}$:

▶ **Invariant 1.** *If* $\exists p \in \mathcal{N},\ r \in \mathbb{N}$ *such that* $o_p^r = (D, v)$*, where* $v \in \mathcal{V}$*, then for each correct* $q \neq p \in \mathcal{N}$ *it holds that* $o_q^r = (M, v)$ *or* $o_q^r = (D, v)$*.*

▶ **Invariant 2.** *If* $\exists p \in \mathcal{N}$*,* $r \in \mathbb{N}$ *such that* $o_q^r = (U, v)$ *for some* $v \in \mathcal{V}$ *then* $\forall q \in \mathcal{N}$*,* $u \in \mathcal{V}$*:* $o_q^r \neq (D, u)$*.*

This framework facilitates the description of MPTC in the next section and can be used to describe most consensus protocols in literature, including [3, 8, 18].

**Problem.** Our goal is to design a round-based consensus protocol that is correct under the previous system and adversary assumptions and that runs a different existing consensus protocol on a different set of processes each round. The selection of protocols and processes for each round must not be predictable by the adversary without the collaboration of correct processes. For the purposes of this work, unpredictability is as described in Section 2.3.

## 3 Moving Participants Turtle Consensus

In this section we describe our Moving Participants Turtle Consensus (MPTC) protocol. MPTC is an $f$-resilient consensus protocol operating in rounds such that in each round a different subset of processes may run a different consensus protocol. We start with some preliminary definitions and notation and then describe the protocol.

### 3.1 Participants and participant sets

MPTC is run by all processes in $\mathcal{N}$. In each round, only a subset of $\mathcal{N}$ is actively running a consensus protocol from a set of correct consensus protocols, $\mathcal{P}$. Let $\mathcal{P}_f$ correspond to the minimum number of processes required to run each protocol in $\mathcal{P}$. As an example, let $\mathcal{P}$ consist of the Ben-Or [3] and One-Third [9] consensus protocols. The first one requires $2f+1$ processes to solve the agreement problem tolerating up to $f$ crash failures while the second one needs $3f+1$. Thus $\mathcal{P}_f = 3f+1$. We assume that $|\mathcal{N}| \gg f$ and thus $|\mathcal{N}| > \mathcal{P}_f$ for most $f$-resilient consensus protocols.

In the remainder of this paper, we say that a process *runs* or *executes* a protocol in $\mathcal{P}$ when it executes a round of that protocol. We will refer to a process executing a protocol in $\mathcal{P}$ at some round of MPTC as a *participant* or an *active participant* of that round. Let $PS = \{S \subseteq \mathcal{N} : |S| = \mathcal{P}_f\}$ be the set of all possible subsets of $\mathcal{N}$ where each subset has size $\mathcal{P}_f$. We call each such set a *participant set.* A process may be a member of multiple participant sets. In each round $r$ of MPTC, only a single participant set, $S_r$, is *active*, that is executing a consensus protocol in $\mathcal{P}$. We assume that participants in each participant set of some round $r$, $S_r \in PS$, are ordered and denote the $i^{th}$ participant in $S_r$ as $S_r^i$. The active participant set for each round is determined by $T$ during initialization, which we describe later in this section.

### 3.2 Configurations

Before describing the initialization procedure and the core of MPTC, we need to define an important concept that encapsulates the information required for a set of processes to run a consensus protocol. We define a *configuration* of MPTC as a tuple $(P, S) \in \mathcal{P} \times PS$. $P \in \mathcal{P}$ describes the consensus protocol to run along with its initialization parameters. To better understand the information contained in the initialization parameters, consider a protocol like Lamport's Paxos [18] and the core consensus protocol he called Synod. In Synod, processes play multiple roles, such as proposers and acceptors. In that sense, $P$ needs to encapsulate not only the protocol under execution, e.g. Synod, but also information related to its initialization such as mapping of proposers and acceptors to processes. The participant set $S \in PS$ corresponds to the set of processes that shall execute the consensus protocol specified by $P$. Let the set of all possible configurations $\mathcal{C} = \mathcal{P} \times PS$. Our approach implements a multi-party computation scheme for an unpredictable mapping between natural numbers (rounds) and configurations. We omit details regarding how to represent $P$ since this is an implementation issue and does not affect our protocol. We assume that $|\mathcal{C}|$ is bounded.

### 3.3 Initialization and trusted dealer

We are now ready to describe the initialization of our protocol, how we are using $T$ to create an unpredictable sequence of configurations, and how the active participants of a round can compute the corresponding configuration.

$T$ is a special process that generates the configuration that each process in $\mathcal{N}$ starts with in the first round. It also provides the processes the means to generate configurations for subsequent rounds. To achieve this, $T$ employs a $(\mathcal{P}_f, f+1, f)$ threshold coin tossing scheme like the one described in Section 2.3. Using this scheme, $T$ shares a function $F_S$ between the $\mathcal{P}_f$ processes of each participant set $S \in PS$. Recall that threshold coin-tossing can be implemented using threshold signatures, thus when we say that $T$ shares a function $F_S$ with each participant set, in reality it simply selects a different public-secret key pair for each $S \in PS$ and shares the secret key. Given some round number $r$, at least $f+1$ processes in $S$ need to collaborate to produce $F_S(r)$ while up to $f$ of them may get compromised. $f+1$ is both a sufficient and necessary number of processes to compute the result of the function shared. $T$ cannot be compromised, failed or attacked by the adversary.

At a high level, $T$ operates as follows:

1. For each $S \in PS$ the dealer picks a function $F_S : \{0,1\}^* \to \mathcal{C}$ and generates a secret share, $h_S^q$, for each $q \in S$.
2. $T$ picks a configuration $C_0 \in \mathcal{C}$.
3. $T$ distributes $C_0$ and shares to processes over private channels. $\forall S \in PS$ each process $p \in S$ receives $h_S^p$ and $C_0$.

Observe that each function shared by the dealer maps arbitrary strings to configurations. This differs from the functions we defined in Section 2.3 which map arbitrary bit strings to bit strings of some finite length $b$. Since $\mathcal{C}$ is finite, there exists $b = \lceil log_2|\mathcal{C}|\rceil$ such that we can trivially obtain an onto function $\{0,1\}^b \to \mathcal{C}$. Thus, the functions we need to share can be trivially obtained by the ones supported by the threshold coin-tossing scheme. Note that, by this high-level algorithm, a process in $\mathcal{N}$ will receive multiple shares, one for each participant set it belongs to. The dealer selects each $F_S$ such that the output is computationally indistinguishable from a randomly chosen function.

We now discuss how $T$ generates the secret shares. Given model parameters $\mathcal{P}$ and $f$, $T$ generates a different set of secret key shares for each subset, $S \in PS$. Each such set of secret key shares implicitly defines a function $F_S$ mapping bit strings to configurations. We call this operation *split* and it is similar to the threshold coin-tossing dealer initialization described in [5]. *split* can be implemented using Shamir's secret sharing [25] $(\mathcal{P}_f, f+1)$. Note that the implementation in [5] is based on verifiable secret sharing because they are considering Byzantine failures. In our model, processes cannot lie and messages cannot be tampered with. As a result, no verification is needed within this context.

Given a secret share, $h_S^p$, of some function $F_S : \mathbb{N} \to \mathcal{C}$ and some input, $r \in \mathbb{N}$, process $p \in S$ can create a function share of $F_S(r)$ using the *share generation* function, $GFS : \mathcal{S} \times \mathbb{N} \to \mathcal{F}$ where $\mathcal{F}$ is the space of valid function shares that can be generated given a share $h \in \mathcal{S}$ and a natural number. We define, $F_S^p(r) = GFS(h_S^p, r)$. A straightforward implementation of $GFS$ can be derived from the signature share generation for threshold signatures [26].

We define the *combine* functions as:

$$combine : \mathcal{F}^{f+1} \times \mathbb{N} \to \mathcal{C}$$

*combine* works by receiving function shares of some function $F_S$ and some input, $r$ and outputting $F_S(r)$ which corresponds to a configuration. More formally, let

$$F_S^Q(r) = \{F_S^q(r) \in \mathcal{F} \mid \forall q \in Q, Q \subseteq S \textit{ and } |Q| = f+1\}$$

be any set of $f+1$ function shares of $F_S(r)$, i.e. $F_S^Q(r) \in \mathcal{F}^{f+1}$. Then we have that:

$$combine(F_S^Q(r), r) = F_S(r)$$

## 3.4 Protocol description

We can now describe the operation of each process executing MPTC. MPTC is an $f$-resilient round-based consensus protocol in which each round is executed under a different configuration. Let $C_r = (S_r, P_r) \in \mathcal{C}$ be the configuration used for round $r$, where $S_r \in PS$ is the active participant set and $P_r \in \mathcal{P}$ the consensus protocol specification for that round. Let $o_r^p$ be the outcome of a process $p \in S_r$ running $P_r$ at round $r$. Let a special value $\perp \notin \mathcal{V} \cup \mathcal{C} \cup \{\{D, M, U\} \times \mathcal{V}\}$ represent the value of an uninitialized variable.

We assume that all processes have common knowledge of $\mathcal{N}$, $f$, $\mathcal{P}$, $\mathcal{C}$ as well as of the functions *GFS combine*. Each process $p \in \mathcal{N}$ runs MPTC with its identifier and some value $x_p \in \mathcal{V}$ as input and at any point in time maintains the following state:

- its current round number, $r_p$, initialized to 0;
- its proposal $proposal_p$, initialized to $x_p$;
- the outcome of a round, $o_p$ representing $p$'s decision state and initialized to $\perp$ at the beginning of each round; and
- the current configuration $c_p$ describing the currently known active participant set and the consensus protocol the active participants execute; it is initialized to $C_0$, which is provided by $T$ during the initialization phase.
- the secret shares, $h_S^p$, $\forall S \in PS$ such that $p \in S$ provided by $T$ during initialization.

We have organized MPTC description in phases. Messages exchanged between processes carry the number of the phase, the id of the sending process, and the current round along with the payload. Messages are of the form $\langle$*phase number*, *process id*, *round*, $\ldots\rangle$. Each round, $r$, of MPTC works in the following 3 phases:

- **Phase 1**: Each process $p \in S_r$ runs a round of the consensus protocol specified by $C_r$. Let $o_p$ be $p$'s outcome for round $r$. If $o_p = (D, v)$, then process $p$ updates $proposal_p = v$, decides $v$ and never updates $o_p$ and $proposal_p$ again in any future round. If $o_p = (M, v)$, then $p$ updates $proposal_p = v$. Regardless of $o_p$'s value, $p$ goes to Phase 2.
- **Phase 2**:
  - *Step 1*: Each $p \in S_r$ computes function share $F_{S_r}^p(r) = GFS(h_{S_r}^p, r)$ and sends a Phase 2 message $\langle 2, p, r_p, o_r^p, F_S^p(r)\rangle$ to all processes in $S_r$. Then $p$ waits for Phase 2 messages from $\mathcal{P}_f - f$ processes in $S_r$. Once $p$ receives enough messages from some $Q \subseteq S_r$, it proceeds to Step 2.
  - *Step 2*: If $o_p = (U, *)$ where $*$ can be any value in $\mathcal{V}$, then $p$ updates its proposal to a value $v$, selected arbitrarily from the outcomes contained in the received Phase 2 messages. It also updates $o_p = (U, v)$.
  - *Step 3*: Let $F_S^Q(r)$ be the set of function shares received from processes in $Q$. $p$ computes the configuration of the next round, $r + 1$, as $C_{r+1} = combine(F_{S_r}^Q(r), r)$ and moves on to Phase 3.
- **Phase 3**: Each $p \in S_r$ sends a Phase 3 message $\langle 3, p, r_p, o_p, C_{r+1}\rangle$ to each process in $S_{r+1}$. $p$ updates its state: $r_p = r + 1$, $c_p = C_{r+1}$ and if it is still undecided, it updates its outcome, $o_p = \perp$. Each process $q \in S_{r+1}$ that receives Phase 3 messages with the same configuration value, $C_{r+1}$, from $\mathcal{P}_f - f$ processes, updates its proposal as follows. Let $R$ denote the set of outcomes received:
  - Case 1: If $\exists o \in R$ such that $o = (D, v)$, then $q$ updates $proposal_q = v$, decides $v$, sets its outcome $o_q = (D, v)$ and never updates $o_q$ and $proposal_q$ again in any future round.
  - Case 2: If $\forall o \in R$ it holds $o = (M, v)$ for some $v \in \mathcal{V}$ then $proposal_q = v$.
  - Case 3: Otherwise, $q$ selects an arbitrary outcome $(*, v) \in R$ where $*$ can be any value in $\{M, U\}$ and updates $proposal_q = v$.

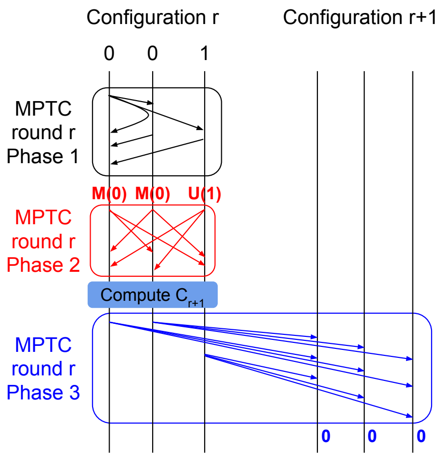


**Figure 1** A round of MPTC consensus.

Then $q$ sets $r_q = r + 1$, $c_q = C_{r+1}$ and if it is still undecided, it sets $o_q = \bot$. Finally, it starts the next round.

Figure 1 shows a visualization of the previous round description. MPTC runs for an unbounded number of rounds and eventually reaches a state in which a decision is made and all correct processes can eventually learn this decision. Messages from old rounds, either delayed in the network or sent by slow processes, are ignored while messages from future rounds are queued to be processed when the receiver reaches that round. The correctness of the previous protocol is presented in the full version of this paper in [23].

## 4 Implementation

In this section, we describe a simple implementation of MPTC as well as a state machine replication protocol we built on top of it. To implement MPTC we need to decide on the following parameters: the choice of protocol set $\mathcal{P}$, the set of possible configurations $\mathcal{C}$, the configuration selection functions $F_S$, $\forall S \in PS$, generated by the trusted dealer, and the implementations of *split*, *GFS* and *combine* functions.

Our set of protocols, $\mathcal{P}$, contains only a single consensus protocol, a parameterized version of single decree Paxos [18] in which each round comes with a predetermined leader known to all active participants. Paxos tolerates $f$ crash failures using $2f + 1$ processes and under failure-free execution conditions, it can reach a decision within a single round-trip of communication. We assume the weakest failure detector, $\diamond\mathcal{W}$, presented in [7] which we implement using timeouts with exponentially increasing timeout periods. This way we ensure that there will be enough rounds executed by sufficiently many processes, which is critical for ensuring termination in our Paxos variant.

The timeouts mentioned above may cause certain processes executing our Paxos variant to exit a round without knowledge of the round's decision. Such processes need to retrieve this knowledge from the rest of the processes. To avoid incurring another round of communication in our Paxos variant, we piggyback this decision state retrieval onto Phase 2 of MPTC. Timed out processes can use the set of outcomes received to update their proposal.

Our set of configurations is $\mathcal{C} = \{(S, P) \mid S \in PS \text{ and } |S| = 2f + 1\}$ where $P \in \mathcal{P}$ is the described Paxos variant. Observe that in contrast to prior work on Turtle Consensus [22] we use the same protocol across configurations. In Turtle Consensus, different configurations used the same $2f+1$ set of processes. As a result, the adversary could try to track the current leader within that set of processes even if the leader changed across different configurations. Therefore, a competent adversary could eventually locate and force Turtle Consensus rounds to fail, which can lead to poor performance. For that reason, Turtle Consensus kept switching between a leader-based (Paxos) and fully decentralized (Ben-Or) consensus protocols across configurations to prevent the adversary from exploiting the leader vulnerability. A side-effect of that approach, however, was that by falling back to a less efficient protocol (Ben-Or) it only achieved sub-par performance compared to the graceful execution using only Paxos rounds. With MPTC we do not need to employ such tactics since the adversary now needs to scan through $|\mathcal{N}| \gg f$ processes before it can identify the leader of our Paxos configuration.

In the implementation that we evaluate in Section 5 we did not implement the Threshold coin-tossing scheme. We emulated it instead by assuming that all participant sets use the same unpredictable function given to all processes via a configuration file. This file defines a sequence of configurations, one for each round, that processes move to in a round-robin fashion. We emulate the restrictions that the cryptographic framework imposes on the adversary by assuming that only the processes involved in rounds $r$ and $r+1$ can learn $C_{r+1}$ and only after Phase 2 of round $r$ completes.

The interested reader can find an actual implementation of Threshold coin- tossing in [5]. In that work, they used cryptographically secure hash functions modeled as random oracles to implement unpredictable functions as well as for the *GFS* function. They also used Feldman's verifiable secret sharing [12] for *split* function, though in our crash-tolerant case Shamir's secret sharing [25] can be used instead. Finally, for *combine* they use Lagrange interpolation with coefficients the computed function shares.

For more details, see the full version of this work in [23].

## 4.1 MPTC-based state machine replication

We used the previous implementation of MPTC to build a SMRP, similar to the one described in [22]. While the components of the implementation are similar, their interactions are different. There are three sets of processes, the clients, the replicas $\mathcal{R}$, and the participants $\mathcal{N}$. The clients issue requests to the participants who order these requests and forward them to replicas. Replicas execute the received requests in the order established by participants and send the results back to participants who then forward them back to clients. Participants can additionally send reconfiguration messages to each other in order to update the configuration of the MPTC execution.

In greater detail, clients send uniquely identifiable requests to sufficiently many participants in order to ensure that at least one correct participant receives each request. The participants receive requests from clients and are responsible for ordering these requests and send them for execution to the replicas. Only one participant set can be active at any point in time. Any participant outside that set receiving a client request relays that request to the currently known active participant set. Active participants receiving client requests spawn MPTC instances, one for each request that needs to be ordered. Each instance has its own identifier and decided requests are ordered according to the identifiers of the MPTC instances that decided them. Clients can only communicate with participants and thus they are unable to launch DoS attacks on the replicas. MPTC is lazily instantiated for each slot and MPTC messages carry instance identifiers so incoming protocol messages are properly processed by the correct instance. If an instance has not yet been created, messages for that instance are queued and processed when it is created. Finally, there are at least $f+1$

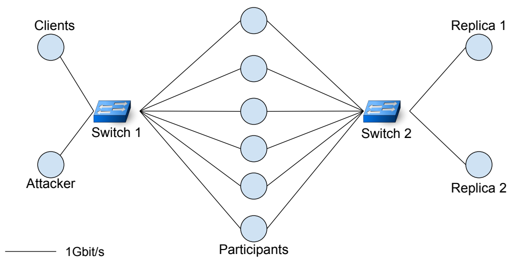


**Figure 2** Experiment topology.

replicas, each of which maintains a copy of state of the service implemented by the SMRP. All replicas are initialized in the same state and execute the clients' requests in the order determined by id of the consensus instance created by the participants for each request.

For a detailed description of this SMRP implementation see [23].

## 5 Evaluation

In this section we present an evaluation of MPTC using the SMRP protocol presented in Section 4.1. In Section 5.1 we present the experiment setup and in Section 5.2 the performance results of MPTC under different attack scenarios.

### 5.1 Setup

We implemented MPTC and the SMRP described in Section 4.1 using `C++`. Our testbed consists of 10 nodes in Emulab [27], each with 8 cores running at 2.4 GHz, with 64GB of memory. For our experiments we used $f = 1$. Two nodes where designated as replicas, six as participants, one as clients, and one as the attacker. Nodes are connected by 1Gbps switched Ethernet as shown in Figure 2. Note that clients and attacker can only connect to participants, while participants connect to both replicas and clients. This choice was made to prevent the attacker from directly attacking the replicas of SMRP, thus degrading performance without attacking the consensus mechanism. All communication between participants takes place through Switch 1. Switch 2 is only used for participant to replica communication. We do not allow participants to communicate through Switch 2 since this would prevent the attacker from saturating the participants' bandwidth with respect to the MPTC execution. This would give MPTC an unfair advantage and would not showcase the benefits of its reconfiguration capabilities. All communication is over TCP/IP except for the DoS attack traffic, which is entirely UDP/IP. One of the two client nodes is used by the attacker and the other for creating legitimate client threads. We use a separate node for attacks in order to limit the effect of bandwidth attacks on the clients' ability to issue requests.

To simplify our evaluation, we set $\mathcal{C}$ to contain only two configurations such that the corresponding participant sets are disjoint. The configuration selection function provided by the trusted dealer (in our implementation by a configuration file) simply alternates between these two configurations every time a round fails. The predetermined Paxos leader of each configuration depends on the round in which the configuration is run and is rotated in a round-robin fashion every time the same participant set is reused. We consider that the attacker does not have this knowledge to make informed decisions regarding targeting processes.

Clients first connect to $f + 1$ random participants to which they issues requests. Once connected, each client executes the following loop: It issues each request to all $f + 1$ participants, waits for a response, discarding duplicate responses, and then sends the next request. Note that by connecting to $f + 1$ participants, we ensure that each client request reaches at least one correct participant who will further forward the request to the active participants. We have client requests contain no-ops, which means that when a decided request becomes ready for execution, replicas can immediately reply with a response.

The attacker creates a small number of attack threads, each of which targets a single participant, selects a random port, and sends UDP dummy messages as fast as it can. Note that these messages are not requests and are not processed by our participants since they never get to the application level. As in the Turtle Consensus evaluation [22], the goal of the attack is to prevent at most one participant from participating in MPTC instances. The attacker can focus all threads on the same participant or spread them across different ones. Since all attack threads are created on a single node, the aggregate bandwidth the attacker threads can saturate from the service cannot exceed 1Gbps.

We conducted experiments to test the throughput and latency of our implementation under normal execution and DoS attacks. Both metrics were measured at the client side. For throughput we measured the aggregate number of operations per second completed by client threads. Note that this is not the actual number of instances completed per second by our SMRP implementation since the same request might be decided more than once.

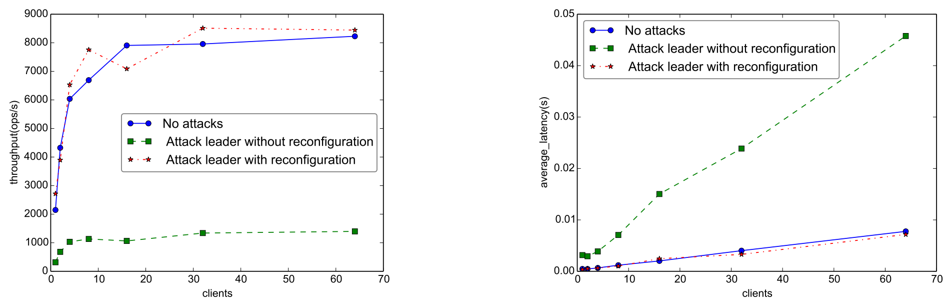
Other parameters of our experiment include:

- Duration: Each experiment lasted 1 minute. We found longer experiments did not significantly affect our metrics.
- Load: The number of concurrent clients, which ranged in our experiments from 1 to 64.
- Request size: The size of the command contained in each client request, which we set to 100 bytes.
- Attack message size: The size of the UDP messages send by attack threads to saturate the participants bandwidth; we set that to 1KB since our experimentation with our platform showed it is the smallest message size with the best results for the attacker.
- Number of attacker threads: Each run involving a DoS attack had 8 attack threads. We found that this number of threads yields best results for the attacker even when all 64 clients are connected to the target sharing the same link.
- Timeout: This is the initial timeout period used in our Paxos variant (Section 4) for each MPTC instance. Every time a round of some instance fails we double the timeout period for that instance.

## 5.2 Results

In our evaluation, we investigated three main scenarios. In the first, we run our implementation of MPTC without any attacks taking place. The performance of this scenario will be our baseline since any attack scenarios drain resources from the system and thus is expected to perform similar or worse. This scenario is labeled “No attacks” in our figures.

The second scenario has the attacker focusing the DoS attack on a single node, the one that hosts the Paxos leader. This attack depletes the leader’s bandwidth. In this scenario no reconfiguration occurs. More specifically, we assume that in each round of MPTC the exact same configuration is chosen and the leader remains the same. Note that this scenario tries to simulate the case where the adversary can accurately track and attack the leader of the Paxos configuration. While any reasonable implementation of Paxos would change leaders



**(a)** Throughput as a function of load. **(b)** Latency as a function of load.

**Figure 3** Moving Participants Turtle Consensus performance under different attack scenarios.

among the $2f + 1$ processes, we set up the scenario to simplify issuing a very efficient attack. In our figures, this scenario is labeled "Attack leader without reconfiguration".

Finally, the third scenario uses an attacker who like in the previous scenario focuses on a single node. In this scenario the attacker is given the initial position of the leader but this time our implementation uses the MPTC version we described in Section 5.1 where consensus instances execution alternates between two disjoint sets of nodes. The attacker strategy here is to saturate the bandwidth of the known leader. It keeps attacking that node for the entirety of the experiment run. This attack is labeled "Attack leader with reconfiguration".

Figure 3a shows the throughput comparison of the previous three experiment scenarios as a function of the load on the SMRP. Each point represents the average throughput over 10 runs for each number of clients. In each of these runs clients connect to random participants, which in turn means that performance will vary across experiments. The first scenario is our best case scenario since the system operates at full resource capacity. The second scenario shows that performance suffers substantially when the Paxos leader is under attack. This is to be expected since the leader's participation is critical for making progress in each MPTC instance. In the third scenario we observe the benefits of the reconfigurable version of MPTC in action. The SMRP throughput is close to that of the No Attacks case. The main reason for this behavior is that since the leader of the first configuration is under attack and lacks the bandwidth to handle the valid traffic, some instance will inevitably fail the first round since the remaining participants will eventually time out. That will cause a reconfiguration that changes the active participant set. The new participants will pick up the failed instances as well as future requests and continue operating at full capacity. The minor deviations observed between scenarios 1 and 3 are mainly due to the randomness of client distribution over the set of all participants.

Figure 3b shows a comparison of the same scenarios as load increases, but this time with respect to latency. Observe that all scenarios behave similarly with latency linearly increasing with load. This behavior is to be expected since, as load increases, the number of concurrent MPTC instances increases, which in turn increases latency for each client. After all, each of them has to wait for a response to their previous request before sending the next one. As in the case of throughput, we see that both scenarios 1 and 3 have similar latencies while scenario 2 performs poorly. The reasoning is the same. In the second scenario the leader under attack is slower in completing instances, which raises the wait time for each client.

Note that this evaluation does not take into account the additional cost of reconfiguration that stems from the cryptographic operations required for threshold coin tossing like RSA exponentiations. We therefore expect that under frequent reconfigurations there will be a wider gap between the performances of scenarios 1 and 3. However, we also expect that such reconfigurations will be infrequent, especially as the number of processes increases. Thus, while not an absolute comparison, our evaluation showcases the expected behavior and advantage of MPTC.

## 6 Related Work

A wide range of crash-tolerant consensus protocols have been proposed in literature each optimized for a different setting and/or metric. Some were designed to handle datacenter-scale systems like [6] which describes how Paxos was used to implement a fault-tolerant database for the Chubby locking service, an instance of which lies in each Google's datacenter. Others are focused on wide area deployments such as Mencius [21], which is a Paxos variant that employs multiple leaders each of which is responsible for a different set of consensus instances and may reside at different datacenters. Another important differentiating aspect of consensus protocols is whether they employ a special leader process like in [8, 18] or whether they are fully decentralized like the protocol proposed in [3]. This can greatly affect the behavior of a consensus protocol under different failure scenarios, including attacks, and was thus used by previous work on reconfigurable consensus [22] to design consensus protocols that provide acceptable performance under certain DoS attacks.

Our work resembles the work on Vertical Paxos [19]. Vertical Paxos is a reconfigurable state machine replication protocol that uses a special auxiliary master process to decide the next configuration of the system including the set of replicas participating in that configuration. Unlike Vertical Paxos, MPTC does not require additional online master processes to compute the next configuration. Our assumed trusted dealer is only active during initialization. In addition, Vertical Paxos is not designed for an adversary capable of compromising even a single process and thus would not perform as well against the DoS attacks described in this work.

Moving target defenses have often been used as response to DoS and Distributed DoS (DDoS) attacks. [14] proposes changing the IP address of the target node for dealing with local IP-based DoS attacks. More recently in [16], Software-Defined Networking (SDN) has been used to implement moving target defense approaches like "random host mutation" in which, similarly to [14], the controller periodically alters the virtual IP addresses of hosts to hide the real IP addresses from an intruder. Our Moving Participants Turtle Consensus approach resembles more the "proactive server roaming" approach in [17]. That is an adaptive approach in which the active server proactively switches servers from an existing pool in order to deal with unpredictable and undetectable attacks. Their approach ensures that only legitimate clients can track the moving server. Like in the case of our MPTC protocol, proactive server roaming performs gracefully during attacks. However, it imposes significant overhead in attack-free scenarios, which is not the case for MPTC since we only reactively change configurations.

Our work assumes an adversary that cannot change the set of corrupted processes over time. Other related work has focused on dynamic models of corruption. [15] introduced proactive secret sharing, an instance of proactive security [24] for supporting secure computation in synchronous distributed systems. These ideas have been adapted to asynchronous ones in [4, 28]. While these approaches did not consider DoS attacks, they are orthogonal to ours and can be used to further improve this work for dealing with mobile adversaries.

Running consensus on a subset of a larger set of processes to decrease message complexity has been explored in [1]. It has also been explored more recently in [20] for improving the scalability of Byzantine agreement on blockchains.

## 7 Conclusions

In this paper we presented Moving Participants Turtle Consensus (MPTC), an extension to the Turtle Consensus protocol [22] that allows running different consensus protocols, on different sets of processes, across different rounds of a single consensus instance. MPTC can deal with adversaries with bounded information on the system by making unpredictable changes in the execution of the protocol. Our evaluation of our prototype implementation of MPTC suggests that we can achieve the performance offered by the most efficient consensus protocols even when the system is under attack.

## References

1 Dan Alistarh, James Aspnes, Valerie King, and Jared Saia. Communication-efficient randomized consensus. In Fabian Kuhn, editor, *Distributed Computing – 28th International Symposium, DISC 2014, Austin, TX, USA, October 12–15, 2014. Proceedings*, volume 8784 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2014.

2 Tuomas Aura, Pekka Nikander, and Jussipekka Leiwo. DOS-resistant authentication with client puzzles. In *Security Protocols*, volume 2133 of *Lecture Notes in Computer Science*, pages 170–177. Springer Berlin Heidelberg, 2001. doi:10.1007/3-540-44810-1_22.

3 Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proc. of the 2nd Annual ACM Symp. on Principles of Distributed Computing*, PODC'83, pages 27–30, New York, NY, USA, 1983. ACM. doi:10.1145/800221.806707.

4 Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *in Proc. 9th ACM Conference on Computer and Communications Security (CCS*, pages 88–97. ACM Press, 2002.

5 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005. doi:10.1007/s00145-005-0318-0.

6 Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, PODC'07, pages 398–407, New York, NY, USA, 2007. ACM. doi:10.1145/1281100.1281103.

7 Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996. doi:10.1145/234533.234549.

8 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996. doi:10.1145/226643.226647.

9 Bernadette Charron-Bost and André Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009. doi:10.1007/s00446-009-0084-6.

10 Benny Chor, Amos Israeli, and Ming Li. Wait-free consensus using asynchronous hardware. *SIAM J. Comput.*, 23(4):701–712, August 1994. doi:10.1137/S0097539790192635.

11 Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages

153–168, Berkeley, CA, USA, 2009. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=1558977.1558988`.

12 Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *The 28th Annual Symposium on Foundations of Computer Science*, pages 427–438, Oct 1987. `doi:10.1109/SFCS.1987.4`.

13 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. `doi:10.1145/3149.214121`.

14 Xianjun Geng and Andrew B. Whinston. Defeating distributed denial of service attacks. *IT Professional*, 2(4):36–42, Jul 2000. `doi:10.1109/6294.869381`.

15 Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *Proc. of the 15th Annual Int. Cryptology Conf. on Advances in Cryptology*, CRYPTO'95, pages 339–352, London, UK, 1995. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=646760.706016`.

16 Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. Openflow random host mutation: Transparent moving target defense using software defined networking. In *Proc. of the 1st Workshop on Hot Topics in Software Defined Networks*, pages 127–132. ACM, 2012. `doi:10.1145/2342441.2342467`.

17 Sherif. M. Khattab, Chatree Sangpachatanaruk, Rami Melhem, Daniel Mosse, and Taieb Znati. Proactive server roaming for mitigating denial-of-service attacks. In *International Conference on Information Technology: Research and Education (ITRE 2003)*, pages 286–290, Aug 2003. `doi:10.1109/ITRE.2003.1270623`.

18 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. `doi:10.1145/279227.279229`.

19 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC'09, pages 312–313, New York, NY, USA, 2009. ACM. `doi:10.1145/1582716.1582783`.

20 Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS'16, pages 17–30, New York, NY, USA, 2016. ACM. `doi:10.1145/2976749.2978389`.

21 Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=1855741.1855767`.

22 Stavros Nikolaou and Robbert van Renesse. Turtle consensus: Moving target defense for consensus. In *Proceedings of the 16th Annual Middleware Conference*, Middleware'15, pages 185–196, New York, NY, USA, 2015. ACM. `doi:10.1145/2814576.2814811`.

23 Stavros Nikolaou and Robbert van Renesse. Moving Participants Turtle Consensus. Technical report, Cornell University, November 2016. `arXiv:1611.03562`.

24 Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks (extended abstract). In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'91, pages 51–59, New York, NY, USA, 1991. ACM. `doi:10.1145/112600.112605`.

25 Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979. `doi:10.1145/359168.359176`.

26 Victor Shoup. Practical threshold signatures. In *Proc. of the 19th Int. Conf. on Theory and Application of Cryptographic Techniques*, pages 207–220, Berlin, Heidelberg, 2000. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1756169.1756190`.

**27** Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the 5th Symp. on Operating Systems Design and Implementation (OSDI'02)*, pages 255–270, Boston, MA, December 2002. Usenix.

**28** Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. APSS: Proactive secret sharing in asynchronous systems. *ACM Trans. Inf. Syst. Secur.*, 8(3):259–286, August 2005. `doi:10.1145/1085126.1085127`.