Verification of GossipSub in ACL2s

Ankit Kumar Max von Hippel Panagiotis Manolios Cristina Nita-Rotaru
Northeastern University
Boston, USA
{kumar.anki, vonhippel.m, p.manolios, c.nitarotaru}@northeastern.edu

GossipSub is a popular new peer-to-peer network protocol designed to disseminate messages quickly and efficiently by allowing peers to forward the full content of messages only to a dynamically selected subset of their neighboring peers (mesh neighbors) while gossiping about messages they have seen with the rest. Peers decide which of their neighbors to graft or prune from their mesh locally and periodically using a score for each neighbor. Scores are calculated using a score function that depends on mesh-specific parameters, weights and counters relating to a peer's performance in the network. Since a GossipSub network's performance ultimately depends on the performance of its peers, an important question arises: Is the score calculation mechanism effective in weeding out non-performing or even intentionally misbehaving peers from meshes? We answered this question in the negative in our companion paper [31] by reasoning about GossipSub using our formal, official and executable ACL2s model. Based on our findings, we synthesized and simulated attacks against GossipSub which were confirmed by the developers of GossipSub, FileCoin, and Eth2.0, and publicly disclosed in MITRE CVE-2022-47547. In this paper, we present a detailed description of our model. We discuss design decisions, security properties of GossipSub, reasoning about the security properties in context of our model, attack generation and lessons we learnt when writing it.

1 Introduction

GossipSub is a new peer-to-peer network protocol used by popular applications like Eth2.0 [12] and FileCoin [6]. Messages transmitted in a GossipSub network are typically categorized into *topics*, which peers of the network can subscribe to or unsubscribe from. A peer can be part of several meshes corresponding to different topics. In contrast to *flood publishing* where a peer forwards every message it receives to all of its neighboring peers subscribed to the corresponding topic, GossipSub uses *lazy pull*, wherein a peer forwards full messages only to its *mesh neighbors* in the relevant topic. A peer can graft or prune a mesh neighbor based on various heuristic security mechanisms that ultimately rely on a locally computed score. The score is calculated periodically by each peer for each of its neighbors and is never shared. The score function, which is used to calculate a neighboring peer's score, depends on application-specific parameters and weights, and takes into account the performance of the neighbor both generally and on a given topic. Ideally, the score of misbehaving peers (*e.g.*, peers that drop messages or forward invalid ones) is penalized, which matters because negatively scored mesh neighbors get pruned.

The GossipSub developers specified their protocol in English prose [55, 56, 57] and implemented it in GoLang [4]. They relied on unit-tests and network emulation [58, 36] of pre-programmed scenarios for testing to show that misbehaving peers in a GossipSub network are eventually pruned. However, simple testing is not enough. Dijkstra famously quoted: "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."

In our companion paper [31], we formalized the GossipSub specification in the ACL2s (the ACL2 Sedan) [22, 13] theorem prover. ACL2s extends ACL2 [27, 28] with an advanced data definition framework (*Defdata*) [16], the *cgen* [17, 15, 18, 14] framework for automatic counterexample generation,

Submitted to:
ACL2 Workshop 2023

a powerful termination analysis based on calling context graphs [48] and ordinals [45, 46, 47], and a property-based modelling/analysis framework, each of which helped immensely in our formalization and verification effort. Our publicly available model [3] is designed to be modular and pluggable *i.e.*, it allows us to reason about parts of the model in isolation as well as about applications running on top of the network. Officially, GossipSub does not come with any properties. We formalized and attempted to prove security properties which (1) we thought should be reasonable for a score-based protocol like GossipSub to satisfy and which (2) the GossipSub developers agree with. One such property, which states that continuously misbehaving peers are eventually pruned, turned out to be invalid in the case of Eth2.0. We leveraged the cgen facility in ACL2s to automatically discover vulnerable network states that invalidate our property. We built attack gadgets using sequences of events which can take a network from a reasonable starting state to one of the discovered vulnerable states. We synthesized attacks which were confirmed by the GossipSub, FileCoin, and Eth2.0 developers and publicly disclosed in MITRE CVE-2022-47547. At the time of writing, the GossipSub developers are actively working on a fix. These results are explained in our companion paper [31]. In this work, we focus on our formalization of the GossipSub ACL2s model and its use for reasoning, simulation and attack synthesis.

Paper Outline. Section 2 describes the GossipSub protocol and our ACL2s model simultaneously. Section 3 describes properties we used to reason about GossipSub. Based on the insights gleaned from proving/disproving these properties, Section 4 describes how we synthesized attacks that can disrupt communication in an Eth2.0 GossipSub network. Section 5 describes some limitations of our model. Section 6 presents related work on mechanized theorem proving efforts in the field of distributed systems. Section 7 concludes.

2 GossipSub Model Description

In this section, we describe our ACL2s model, while also giving an overview of how the GossipSub protocol works. Interested readers are encouraged to walk through our ACL2s formalization whose presentation mirrors this section [2]. Consider the mesh shown in Figure 1. Full-message payloads are

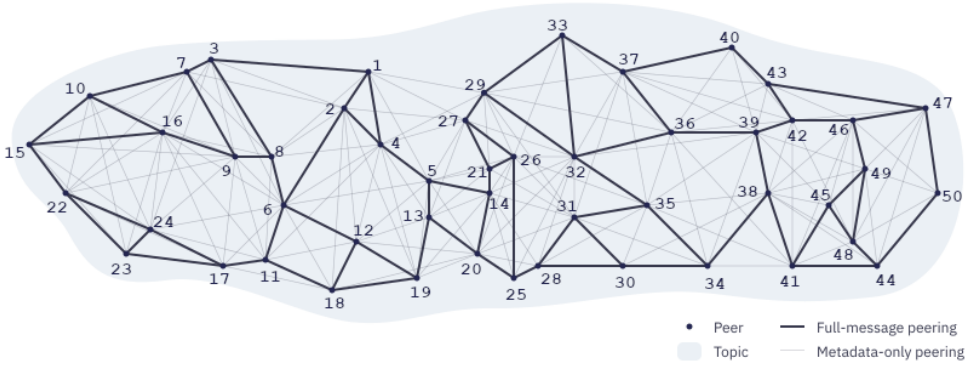

Figure 1: A mesh of peers subscribing to the same topic. This Figure was taken from [5].

forwarded on the full-message peering edges, which consume more bandwidth, while the metadata-only peering edges are used only to advertise and request full-messages using corresponding "`IHAVE`" and "`IWANT`" Remote Procedure Calls (RPCs). These RPCs carry the metadata of the full-message being advertised or requested, which is considerably smaller than the message itself. In this way, network

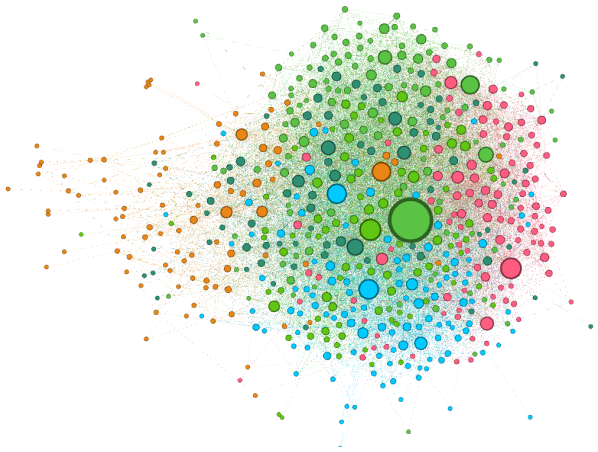

Figure 2: A community graph of Eth2.0 nodes where a bigger node implies greater degree. Similarly colored nodes have more edges among them than to the rest of the graph.

congestion is kept in check, and full-messages are supposed to be eventually disseminated to all peers who subscribe to the given topic. As an example, peers numbered `1` and `2` can send full-messages to each other, since they are mesh neighbors on the illustrated topic and share a full-message peering edge. However, peer `29` can receive a full-message from `1` only if it requests one, by sending an `IWANT` message in response to a corresponding `IHAVE` message received from peer `1`. Note, Figure 1 only shows a single GossipSub mesh. However, real world applications can have an arbitrary number of topics and corresponding meshes, *e.g.*, Eth2.0 supports up to 71 topics. Figure 2 illustrates a community graph of an actual Eth2.0 network from Li et. al. [35].

In order to model and analyze GossipSub in ACL2s, we rely heavily on Defdata to easily model network components, and on cgen for property-based testing. Distributed systems are often described in terms of *state machines*, namely automata that encode the discrete behaviors of a peer in the system [32, 53]. In our model, we describe the state-space of a GossipSub peer with a `peer-state` type, and then implement the GossipSub state machine using a state transition function. We use a Defdata map from peers to their corresponding states to capture the state of an entire network.

```
(defdata group (map peer peer-state))
```

We use records whenever we need to store state because of the convenience of using named fields to access the internals of the state. While proving theorems referring to states, we noticed that we routinely had to prove helper lemmas about the types of the contents of those states. This motivated us to rewrite records in the ACL2s books to enable such helper lemmas automatically, leading to cleaner code.

The local state of a peer is modeled as a record using the following definition:

```
(defdata peer-state
  (record (nts        . nbr-topic-state)
          (mst        . msgs-state)
          (nbr-tctrs  . pt-tctrs-map)
          (nbr-gctrs  . p-gctrs-map)
          (nbr-scores . peer-rational-map)))
```

where (1) `nts` is a record that stores information about the peer's neighbors' subscriptions, the peer's mesh neighbors, and the peer's fanout (which we describe later); (2) `mst` is a record that stores the peer's

messages and related state; (3) `nbr-tctrs` and `nbr-gctrs` are total maps that store counters used for computing a neighbor's topic-specific and general scores (respectively); and finally (4) `nbr-scores` is a total map from peers to their scores. Note that `peer` and `topic` are ACL2 symbols, while `tctrs` is a record that keeps track of a peer's behaviors in each topic:

```
(defdata tctrs
  (record (invalidMessageDeliveries . non-neg-rational)
          (meshMessageDeliveries    . non-neg-rational)
          (meshTime                 . non-neg-rational)
          (firstMessageDeliveries   . non-neg-rational)
          (meshFailurePenalty       . non-neg-rational)))
```

Similarly, `gctrs` keeps track of a peer's general behaviors, not pertaining to any single topic:

```
(defdata gctrs
  (record (apco . rational)  ;; application provided score
          (ipco . non-neg-rational) ;; ip-colocation factor
          (bhvo . non-neg-rational))) ;; track misbehavior
```

Notice that each of the counters is a rational, not a natural. This is because counters are supposed to fractionally decay at regular intervals. The rate of decay is dependent on the application running on top of GossipSub. We define maps for each of these counters:

```
(defdata pt (cons peer topic))
(defdata pt-tctrs-map (map pt tctrs))
(defdata p-gctrs-map (map peer gctrs))
(defdata peer-rational-map (map peer rational))
```

For each map, we define a lookup function with default values. ACL2s automatically proves termination and input-output contracts, which suffices to show that the maps are total. As an example, the following is the lookup function for scores:

```
(definecd lookup-score (p :peer prmap :peer-rational-map) :rational
  (let ((x (mget p prmap)))
    (match x
      (nil 0)  ;; default value
      (& x))))
```

We now explore each component of the `peer-state`.

Neighbor topics state. A key objective of the GossipSub protocol is to reduce network congestion, while forwarding data as quickly as possible. To achieve this, a peer forwards full messages only to a subset of its neighboring peers. A GossipSub peer keeps track of the topics its neighbors subscribe to, using the `nbr-topicsubs` field in `nbr-topic-state`. Using this information, it is able to build up a picture of the topics around it and which peers are subscribed to each topic. The peers to which it forwards full messages in topics it does not itself subscribe to constitute a list called a *fanout*, and is stored in the `topic-fanout` field, which is a map from topics to lists of peers (`lop`). The peer forwards full messages to other peers with whom it shares mesh membership. Mesh memberships are stored in the `topic-lop-map` field `topic-mesh`. Finally, the peer stores the last time it published a message to its fanout. This is used to expire a peer's fanout, if it has been too long since the peer last published.

```
(defdata peer symbol)
(defdata lop (listof peer))
(defdata topic-lop-map (map topic lop))
```

```
(defdata topic-nnr-map (map topic non-neg-rational))

(defdata nbr-topic-state
  (record (nbr-topicsubs . topic-lop-map)
          (topic-fanout  . topic-lop-map)
          (last-pub      . topic-nnr-map)
          (topic-mesh    . topic-lop-map)))
```

Messages state. `msgs-state` is a record type used to store information about messages received, requested, seen, or forwarded by a peer. First we define the types `pid-type`, which is just an alias for the type `symbol`, and the record type `payload-type`.

```
(defdata-alias pid-type symbol)
(defdata payload-type (record (content . symbol)
                              (pid     . pid-type)
                              (top     . topic)
                              (origin  . peer)))
```

`payload-type` is a record used to represent full messages, carrying the message content, payload id, the topic of this message and the peer who originated it. pid-type represents payload ids, a hash of a message payload which can identify the full message content. The msgs-state is defined as follows:

```
(defdata msg-peer (v (cons payload-type peer)
                     (cons pid-type peer)))
(defdata msgpeer-rat (map msg-peer rational))
(defdata msgs-waiting-for (map pid-type peer))
(defdata mcache (alistof payload-type peer))

(defdata msgs-state
  (record (recently-seen   . msgpeer-rat)
          (pld-cache       . mcache)
          (hwindows        . lon)
          (waitingfor      . msgs-waiting-for)
          (served          . msgpeer-rat)
          (ihaves-received . nat)
          (ihaves-sent     . nat)))
```

`msgs-state` stores (1) `recently-seen`, a map from either a full-message or a message hash to the time since receipt; (2) `pld-cache`, an association list of full messages and their senders; (3) `hwindows`, or history windows, a list of naturals where each is the number of messages received in an interval; (4) `waitingfor`, a map from message ids of messages that haven't been received yet, to peers one has sent corresponding `IWANT` requests to; and (5) `served`, a map from either a full-message or a message hash to a rational, denoting the count of the number of times this message was served. The `served` map helps to detect peers sending too many `IWANT` messages. Finally, `msgs-state` contains (6) `ihaves-received` and `ihaves-sent`, which store the number of `IHAVE` messages received and sent, respectively.

Events. Our model includes events that can occur in a network. Events include peers sending or receiving (1) control messages like `GRAFT` or `PRUNE` for mesh control, `SUB`, `UNSUB`, `JOIN` and `LEAVE` for topics, or `CONNECT1` or `CONNECT2` messages to edit neighbors; (2) `PAYLOAD` for carrying message payload, or `IHAVE` for advertising or `IWANT` for requesting message payloads; and (3) `HBM` for heart-beat events occurring at each peer at regular intervals. A list of events will have type `loev`. Events are defined as follows:

```
(defdata verb (enum '(SND RCV)))
(defdata rpc (v (list 'CONNECT1 lot)
                (list 'CONNECT2 lot)
                (list 'PRUNE topic)
                (list 'GRAFT topic)
                (list 'SUB topic)
                (list 'UNSUB topic)))
(defdata data (v (list 'IHAVE lopid)
                 (list 'IWANT lopid)
                 (list 'PAYLOAD payload-type)))
(defdata mssg (v rpc data))
(defdata evnt (v (cons peer (cons verb (cons peer mssg)))
                 (list peer 'JOIN topic)
                 (list peer 'LEAVE topic)
                 (list peer verb peer 'CONNECT1 lot)
                 (list peer verb peer 'CONNECT2 lot)
                 (list peer 'HBM pos-rational)
                 (list peer 'APP payload-type)))
(defdata hbm-evnt (list peer 'HBM pos-rational))
(defdata-subtype hbm-evnt evnt)

(defdata loev (listof evnt))
```

Heart-beat events occur at regular intervals at each peer in a GossipSub network. Several maintenance activities are performed during each such event. Scores are updated for neighboring peers, which are then used to update mesh memberships. Counters are multiplied by some application-specific decay factors. Neighboring peers that are not part of any mesh are sent `GRAFT` messages at regular intervals, provided that their addition could improve the average score of peers in the corresponding mesh. Several of these actions depend on application-specific weights (used for calculating scores) and parameters.

Parameters and Scoring. We store the weights and parameters relevant for scoring in a record called a `twp`. This record totally captures the application-specific configuration of a GossipSub instance, *e.g.*, we can uniquely specify the configuration used by Eth2.0 in a `twp`. Thus, in order to simulate an application running on top of GossipSub with our model, all we need to know is the application-specific `twp`. This is what we mean when we say our model is "pluggable".

```
(defdata weights
  (record (w1  . non-neg-rational)
          (w2  . non-neg-rational)
          (w3  . non-pos-rational)
          (w3b . non-pos-rational)
          (w4  . neg-rational)
          (w5  . non-neg-rational)
          (w6  . neg-rational)
          (w7  . neg-rational)))

(defdata params
  (record (activationWindow              . nat)
          (meshTimeQuantum               . pos)
          (p2cap                         . nat)
          (timeQuantaInMeshCap           . nat)
```

```
          (meshMessageDeliveriesCap       . pos-rational)
          (meshMessageDeliveriesThreshold . pos-rational)
          (topiccap                       . rational)
          (grayListThreshold              . rational)
          (d                              . nat)
          (dlow                           . nat)
          (dhigh                          . nat)
          (dlazy                          . nat)
          (hbmInterval                    . pos-rational)
          (fanoutTTL                      . pos-rational)
          (mcacheLen                      . pos)
          (mcacheGsp                      . non-neg-rational)
          (seenTTL                        . non-neg-rational)
          (opportunisticGraftThreshold    . non-neg-rational)
          (topicWeight                    . non-neg-rational)
          (meshMessageDeliveriesDecay     . frac)
          (firstMessageDeliveriesDecay    . frac)
          (behaviourPenaltyDecay          . frac)
          (meshFailurePenaltyDecay        . frac)
          (invalidMessageDeliveriesDecay  . frac)
          (decayToZero                    . frac)
          (decayInterval                  . pos-rational)))

(defdata wp (cons weights params))
(defdata twp (map topic wp))
```

Note that GossipSub required weights `w3` and `w3b` to be negative. However, the use of Defdata allowed us to automatically find that FileCoin used an invalid `twp` because it set `w3` and `w3b` to zero. We discussed this with the GossipSub developers who then agreed to allow zero values. Given `T`, a set of topics which the neighbor subscribes to; `tctrs`, the neighbor's topic specific counters; `gctrs`, the neighbor's global counters; and a `twp` containing entries for each topic our neighbor subscribes to, the score function calculates a neighbor's score as shown below.

$$score(q) = \min\left(\mathrm{TC}, \sum_{\tau \in T} \mathrm{tw}^{\tau} \times \sum_{i \in \{1,2,3,3b,4\}} w_i^{\tau} P_i^{\tau}\right) + w_5 P_5 + w_6 P_6 + w_7 P_7$$

where

$$\texttt{(weights . params)} = \texttt{(mget } \tau \texttt{ twp)}$$

$$w_i^{\tau} = \texttt{(weights-w}i\texttt{ weights)}$$

$$P_1^{\tau} = \texttt{(calcP1(tctrs-meshTime tctrs) (params-meshTimeQuantum params)}$$
$$\texttt{(params-timeQuantaInMeshCap params))}$$

$$P_2^{\tau} = \texttt{(calcP2(tctrs-firstMessageDeliveries tctrs) (params-p2cap params))}$$

$$P_3^{\tau} = \texttt{(calcP3(tctrs-meshTime tctrs) (params-activationWindow params)}$$
$$\texttt{(tctrs-meshMessageDeliveries tctrs)}$$
$$\texttt{(params-meshMessageDeliveriesCap params)}$$
$$\texttt{(params-meshMessageDeliveriesThreshold params))}$$

$$P_{3b}^{\tau} = \texttt{(calcP3b(tctrs-meshTime tctrs) (params-activationWindow params)}$$

```
        (tctrs-meshFailurePenalty tctrs)
        (tctrs-meshMessageDeliveries tctrs)
        (params-meshMessageDeliveriesCap params)
        (params-meshMessageDeliveriesThreshold params))
```

$P_4^\tau =$ `(calcP4(tctrs-invalidMessageDeliveries tctrs))`

$P_5 =$ `(gctrs-apco gctrs)`

$P_6 =$ `(gctrs-ipco gctrs)`

$P_7 =$ `(calcP7 (gctrs-bhvo gctrs))`

$\mathrm{tw}^\tau =$ `(params-topicweight params)`

$\mathrm{TC} =$ `(params-topiccap (cddar twp))`

TC does not depend on any topic, but since it is stored in a `twp` which is indexed by `topic`, its value is replicated in each of the corresponding `params`. So, it is fine to extract its value from the first entry of a `twp`. Note that for score calculations, we require a non-empty `twp`.

Each of the `calcPi` functions where `i` $\in \{$`1,2,3,3b,4,7`$\}$ is used to calculate contributions to the score by one or more of counter values from `tctrs`. `calcP1` calculates the contribution to a neighbor's score based on the time spent in common meshes. `calcP2` awards score for being one of the first few to forward a message. `calcP3` calculates penalties due to the mesh message transmission rate being below a given threshold of `(params-meshMessageDeliveriesThreshold params)`. Whenever a peer is pruned, its corresponding `tctrs` counter `meshFailurePenalty` is augmented by the mesh message transmission rate deficit. This counter is not cleared even after the peer has been pruned. `calcP3b` scores mesh message delivery failures based on the value of this counter. Hence, P_{3b}^τ is a "sticky" value which is supposed to discourage a peer that was pruned because of under-delivery from quickly getting re-grafted in a mesh. `calcP4` calculates the penalty on score due to sending invalid messages. `calcP7` calculates penalties due to several kinds of misbehaviors described by the GossipSub specification. An example of such misbehaviors includes spamming with too many `IHAVE` messages which are either bogus and/or not following up to the corresponding `IWANT` requests.

The Transition Function. We define a transition function `run-network`, which, given an initial `Group` state and a list of `evnt`, produces a trace of type `egl`, which is an alist of `evnt` and `group`. Hence, after running a simulation, we have access to the state of the `Group` after each `evnt` was processed. `run-network` depends on the transition function for the `peer-state` (`transition`), which depends on the transition functions for `nbr-topic-state` (`update-nbr-topic-state`) and for `msgs-state` (`update-msgs-state`). For brevity, we mention only the signatures of each of these functions below. Notice that each of these signatures represents neatly and concisely the types of the formal arguments and the function return type, which is very useful in a large code base.

```
(defdata egl (alistof evnt group)) ;; simulation trace

(definecd run-network (gr :group evnts :loev i :nat r :twp s :nat) :egl
...)

(defdata peer-state-ret
  (record (pst . peer-state)
          (evs . loev)))

(definecd transition
```

```
  (self :peer pstate :peer-state evnt :evnt r :twp s :nat) :peer-state-ret
...)

(defdata msgs-state-ret
  (record (mst . msgs-state)
          (evs . loev)
          (tcm . pt-tctrs-map)
          (gcm . p-gctrs-map)))

(definecd update-msgs-state (mst :msgs-state evnt :evnt pcm :pt-tctrs-map
                             gcm :p-gctrs-map r :twp) :msgs-state-ret
...)

(defdata nbr-topic-state-ret
  (record (nts . nbr-topic-state)
          (evs . loev)
          (tcm . pt-tctrs-map)
          (gcm . p-gctrs-map)
          (sc  . peer-rational-map)))

(definecd update-nbr-topic-state (nts :nbr-topic-state
                                  nbr-scores :peer-rational-map
                                  tcm :pt-tctrs-map gcm :p-gctrs-map
                                  evnt :evnt r :twp s :nat) :nbr-topic-state-ret
...)
```

A GossipSub peer can select a random subset of its fanout and promote them as mesh members. It can also selects a random subset of its neighbors to advertise with "IHAVE" messages. Such non-determinism is handled by sending a random seed `s` as a formal parameter to `run-network`, which is then propagated to the other transition functions it depends on. Observe that a single event like full message forwarding can trigger several more forwards, causing a cascade of events. Such events are represented by the `evs` field in the return types of `update-msgs-state` and `update-nbr-topic-state`. In order to limit the total number of events processed, we send a natural number `i` as a formal parameter to the `run-network` function.

When proving contract theorems for the transition functions, we needed to prove the types of terms returned by utility functions, like `shuffle`, or ACL2 functions like `set-difference-equal`. For this, we used polymorphism and automated type-based reasoning provided by Defdata, as shown below:

```
(sig set-difference-equal ((listof :a) (listof :a)) => (listof :a))
(sig shuffle ((listof :a) nat) => (listof :a))
```

We made heavy use of higher-order macros written by Manolios [1] to improve the readability of our code. For example, `create-map*` is a list functor. Given an admitted function name or a lambda expression `f` of type `a` $\rightarrow$ `b`, `create-map*` defines a function `map*-*f` of type `(listof a)` $\rightarrow$ `(listof b)`. In the following code snippet, we show theorems proving that it obeys the functor laws, and give an example of its usage. `map*` is a syntactic sugar that maps function `f` onto a list without referring to the generated function name `map*-*f`.

```
(definec id (x :all) :all
  x)
;; Proof that create -map* is a list functor
;; 1) functor mapping preserves the identity function
```

```
(property functor-id (xs :tl)
  (== (map* id xs) ;; id is an identity function for lists
      (id xs)))

;; 2) functor mapping preserves function composition. f and g are declared
;; using defstub. gof is defined as a composition of g and f
(property functor-comp (xs :tl)
  (== (map* gof xs)
      (map* g (map* f xs))))

;; function to create a list of SND GRAFT events from peer p to a list of peers
(create-map* (lambda (tp p) `(,p SND ,(cdr tp) GRAFT ,(car tp)))
             lotopicpeerp
             loevp
             (:name mk-grafts)
             (:fixed-vars ((peerp p))))

(check= (map* mk-grafts '((FM . A) (DS . B)) 'P)
        '((P SND A GRAFT FM) (P SND B GRAFT DS)))
```

Given an admitted function name or a lambda expression `f` of type `a` $\times$ `b` $\rightarrow$ `b`, the higher order function `create-reduce*` defines a function `reduce*-*f` which accepts a list of elements of type `a`, an initial accumulator value of type `b`, and returns a reduction of the list using `f`, from left to right.

```
;; function to extract all the subscribers (neighboring peers) from a topic-lop-map
(create-reduce* (lambda (tp-ps tmp) (app tmp (cdr tp-ps)))
                lopp
                topic-lop-mapp
                (:name subscribers))

(check= (reduce* subscribers '()
                 '((T1 P1 P2 P3)
                   (T2 P4 P5 P1)))
        '(P4 P5 P1 P1 P2 P3))
```

3 Reasoning about the scoring function

Based on the observation that honest peers can be distinguished from malicious ones based on their observable behaviors (using local counters and scores), and thus, the overall network can be made more secure and performant if every honest peer promotes their well-behaving neighbors and demotes poorly-behaved ones, we came up with the following informal fundamental property.

Fundamental Property of GossipSub Defense Mechanisms. *Peers who behave poorly will be demoted by their neighbors. Peers who behave better-than-average will be promoted by their neighbors. Promotion/demotion is entirely based on local peer behavior.*

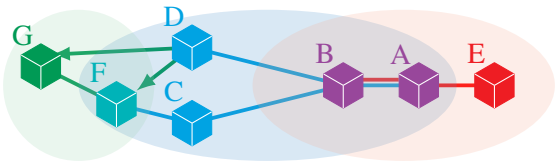

Figure 3: An example network.

Before reasoning about the formalization of this fundamental property later in the paper, we discuss the importance of this fundamental property. Consider the simple example shown in Figure 3, where peers A and B subscribe to, and are mesh neighbors in both Red and Blue topics. It might be possible for B to observe that A behaves perfectly well in the Blue topic while simultaneously misbehaving in the Red topic. This is not good for B because it depends on A for all of its messages in the Red topic. Note that this is a very simplified example. In an actual network, B could have several other neighbors subscribed to the Red topic. But as we will show later, it is equally trivial to have a scenario where all of B's neighbors isolate it from communications in the Red topic. In this example, we want B to prune A from its Red mesh in hopes of finding a better mesh neighbor later on. Reasoning about this fundamental property directly would be difficult due to the massive search-space of possible attack vectors. Hence, we focus on the following liveness property capturing the essence of the fundamental property:

Property 1 If a peer's score relating to its performance in any topic is continuously non-positive, then the peer's overall score should eventually be non-positive:

$$\forall q, \tau :: \langle \boldsymbol{G}(score(q) \textit{ for topic } \tau \leq 0) \Rightarrow \boldsymbol{F}(score(q) \leq 0) \rangle$$

where $score(q)$ *for topic* t *is defined below.*

$$\mathrm{tw}^{\tau} \times \sum_{i \in \{1,2,3,3b,4\}} w_i^{\tau} P_i^{\tau}$$

Notice that Property 1 is temporal. We write the non-temporal version of this property in context of Eth2.0 (using Eth2.0 `twp`) in ACL2s as shown below, and disprove the temporal version using an induction argument later in the paper.

Property 2

```
(property (ptc :pt-tctrs-map pcm :p-gctrs-map p :peer top :topic)
  :hyps  (^ (member-equal '(,p . ,top) (acl2::alist-keys ptc))
            (> (lookup-score p (calc-nbr-scores-map ptc pcm *eth-twp*)) 0))
  (> (calcScoreTopic (lookup-tctrs p top ptc) (mget top *eth-twp*)) 0))
```

The following is one of the counter-examples to the above property, generated by cgen in ACL2s:

```
((top 'agg)
 (p 'p4)
 (pcm '((p3449 (:0tag . gctrs) (:apco . 0) (:bhvo . 0) (:ipco . 0))
        (p3450 (:0tag . gctrs) (:apco . 0) (:bhvo . 0) (:ipco . 0))
        (p3451 (:0tag . gctrs) (:apco . 0) (:bhvo . 0) (:ipco . 0))))
 (ptc '(((p4 . agg)
         (:0tag . tctrs)
         (:firstmessagedeliveries . 0)
         (:invalidmessagedeliveries . 0)
```

```
        (:meshfailurepenalty . 0)
        (:meshmessagedeliveries . 1)
        (:meshtime . 42))
       ((p4 . blocks)
        (:0tag . tctrs)
        (:firstmessagedeliveries . 324)
        (:invalidmessagedeliveries . 0)
        (:meshfailurepenalty . 0)
        (:meshmessagedeliveries . 330)
        (:meshtime . 377))
       ((p4 . sub1)
        (:0tag . tctrs)
        (:firstmessagedeliveries . 371)
        (:invalidmessagedeliveries . 0)
        (:meshfailurepenalty . 0)
        (:meshmessagedeliveries . 377)
        (:meshtime . 324))
       ((p4 . sub2)
        (:0tag . tctrs)
        (:firstmessagedeliveries . 318)
        (:invalidmessagedeliveries . 0)
        (:meshfailurepenalty . 0)
        (:meshmessagedeliveries . 324)
        (:meshtime . 371))
        ... ))
```

For brevity, we omit entries for peer-topic key values for peers other than `p4`. In property-based testing, the free variables of a property under test are assigned values using a synergistic combination of theorem proving and random assignments computed using type-based enumerators (generators) in an effort to discover counterexamples to the property. Observe that Property 2 depends on `ptc` and `pcm` which do not have trivial types. These are maps containing records which themselves consist of several numerical values, which makes the search space of possible counter-examples immensely large. In order to make it easier for cgen to find counter-examples, we wrote custom enumerators to enumerate restricted values for `topic`, `tctrs`, `gctrs`, `pt-tctrs-map` and `p-gctrs-map`. Specifically, we limit the penalties on the score due to some counters, such that the negative contributions due to misbehavior are comparable to the positive contributions due to good behavior. Below we define custom enumerators for `topic` and `tctrs`.

```
(definec topics () :tl
  ;; valid topics used in Ethereum
  '(AGG BLOCKS SUB1 SUB2 SUB3))

(definec nth-topic-custom (n :nat) :symbol
  (nth (mod n (len (topics))) (topics)))
(defdata lows (range integer (0 <= _ <= 1))) ;; high values
(defdata-subtype lows nat)
(defdata highs (range integer (300 < _ <= 400))) ;; low values
(defdata-subtype highs nat)

(defun nth-bad-counters-custom (n)
;; setting invalidMessageDeliveries and meshFailurePenalty to 0 due to high penalty
```

```
(defun nth-good-counters-custom (n)
  (tctrs 0 (nth-highs (+ n 2)) (nth-highs (+ n 3)) (nth-highs (+ n 4)) 0))

(defun nth-counters-custom (n) ;; custom enumerator for tctrs
  (if (== 0 (mod n 4))
      (nth-bad-counters-custom n)
    (nth-good-counters-custom n)))
```

Besides Property 2, we formalize three safety properties for GossipSub, stated below. Together, these four are the most general properties of the score function, which must hold in order for the fundamental property to hold.

Property 3 Increasing bad-performance counters (which are multiplied with negative weights) should decrease the overall score.

Property 4 Increasing good-performance counters (which are multiplied with positive weights) will not decrease the score for a mesh peer that has been in the mesh for a sufficiently long time.

Property 5 If two peers subscribe to the same topics, and achieve identical per-topic counters, and identical global counters, then they achieve identical scores.

We are able to find counterexamples to Property 3 in much the same way as 2, however, in this work we focus on the counterexamples to Property 2, which are more interesting in terms of attack generation. We manually prove that Property 4 holds over all configurations (available with the paper artifacts). The proof that Property 5 holds over all configurations follows directly from referential transparency.

We also prove a limit on the maximum score achievable in a topic, as shown below.

```
(property max-topic-score (tctrs :tctrs weights :weights params :params)
  (<= (calcScoreTopic tctrs (cons weights params))
      (* (params-topicweight params)
         (+ (* (mget :w1 weights) (params-timeQuantaInMeshCap params))
            (* (mget :w2 weights) (params-p2cap params))))))
```

4 Attack Generation

The counter-example 3 which we obtained in the previous section is a specification for an unsafe state that does not satisfy Property 2 for an Eth2.0 network. However, we are also interested in characterizing and generating attacks against an Eth2.0 GossipSub network for three main reasons: (1) to show that an unsafe state is **reachable** from a reasonable start state, (2) **invalidation** of our temporal Property 1 using the trace generated from the attack, and finally (3) demonstration of **scalability** of our attack to large networks, typically of the size and shape used by real world applications.

Counter-example 3 suggests that an unsafe state is one where a neighboring peer throttles communication in a particular topic while maintaining an overall positive score, hence, avoiding getting pruned. Using this insight, we design attack gadgets that can perpetrate such attacks locally. We define an *attack gadget* as a tuple $\langle A, V, S \rangle$, where A, V are peers (A is the attacker and V is the victim), S is a set of subnet topics (the attacked topics), and A, V are mesh neighbors over a set of topics that is a superset of S. For each $i \in \mathbb{N}$, we define AG_i to be the set of attack gadgets where $|S| = i$. Figure 4 illustrates an example attack gadget in AG_2 where peers A and V are neighbors in four meshes corresponding to topics: Red, Yellow, Blue and Green, out of which A is attacking V in $S = \{\text{Red}, \text{Blue}\}$. We generate a sequence of

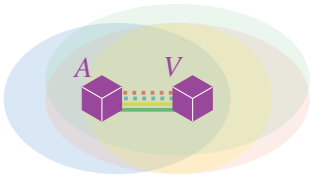

Figure 4: An example AG_2 attack gadget $\langle A, V, \{\text{Red}, \text{Blue}\}\rangle$.

events consisting of message transmission events from A to V (referred to as `a` and `v` in code) as well as heart-beat events at V (V updates the scores of its peers during heart–beat events). These events are designed to either restrict or completely block communication to V in the attacked topics while maintaining normal communication rate in all the other topics, as shown in the following code snippet.

```
(definecd emit-evnts (a v :peer ts ats :lot n m e :nat) :loev
  ;; mesh message deliveries in attacked topics
  (app (emit-meshmsgdeliveries-peer-topics a v ats m)
       ;; mesh message deliveries in other topics
       (emit-meshmsgdeliveries-peer-topics a v (set-difference-equal ts ats) n)
       ;; heart-beat events at the victim node
       '((,p2 HBM ,e))))
```

The expression `(emit-meshmsgdeliveries-peer-topics a v ts ats n m e)` generates a list of events E sending `m` mesh messages in the attacked topics and `n` mesh messages in the other topics from peer `a` to peer `v` per heart-beat at `v` which happens every `e` seconds. `m` is generally set to `0` or `1` in order to block or throttle communication in the attacked topic meshes. We choose a start state of the network based on actual full-network topologies of the Eth2.0 testnets Ropsten (shown in Figure 2), Goerli and Rinkeby, as measured by Li et. al [35]. Table 1 characterizes each of these topologies.

Network	Nodes	Degree			Diameter
		min	max	avg	
Ropsten	588	1	418	25.49	5
Goerli	1355	1	712	28.26	5
Rinkeby	446	1	191	68.96	6

Table 1: Eth2.0 Network Characteristics

We create our start state as a `Group`, using the topologies provided. In this `Group`, we initialize our attack gadget and simulate its run over the generated sequence of events E using the `run-network` function. We use function `scorePropViolation` described below, to detect violations of Property 2 in peer states occurring in the output trace.

```
;; ats is the list of topics being attacked
(definec scorePropViolation (ps :peer-state p :peer ats :lot twpm :twp) :boolean
  (match ats
    (() (> (lookup-score p (calc-nbr-scores-map (peer-state-nbr-tctrs ps)
                                                (peer-state-nbr-gctrs ps) twpm))
           0))
    ((top . rst) (^ (< (calcScoreTopic
```

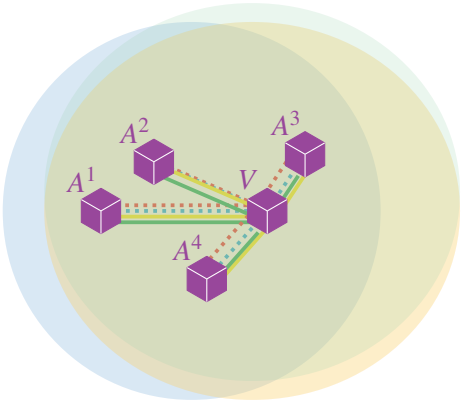

Figure 5: An eclipse attack using AG_2 gadgets

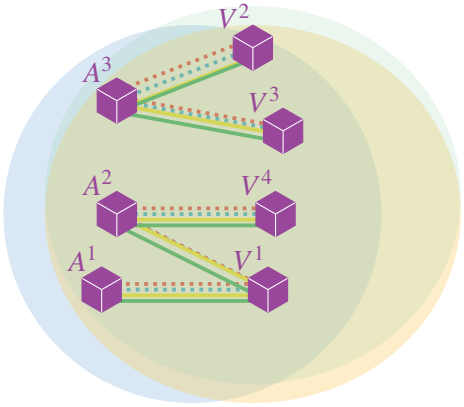

Figure 6: A partition attack using AG_2 gadgets

```
                (lookup-tctrs p top (peer-state-nbr-tctrs ps))
                (mget top twpm))
               0)
        (scorePropViolation ps p rst twpm)))))
```

We observe that after the first heart-beat event at the victim peer, the state of the network at each subsequent heart-beat is identical. Since Property 2 is being violated at each of these events, we make an inductive claim that it will forever be violated, thus giving a counter-example to our liveness Property 1. We wrote optimized versions of the `run-network` function to generate traces of property violations as a list of booleans, instead of generating the full trace of type `egl`. We ran our simulations for 100,000 events to ensure that we ended up in identical states, taking about a minute on an M1 Macbook Air.

Finally, we scaled our attacks to build *eclipse* and *network partition* attacks using a combination of attack gadgets. Figures 5 and 6 give an intuition of how to combine our attack gadgets to carry out these attacks. Our companion paper discusses the specifics of these attacks in more detail.

5 Limitations

We now discuss limitations of our model. The most crucial aspect of property-based testing is counter-example generation for invalid properties. As explained previously, our properties depend on complex types, for which we had to write custom enumerators. Coming up with enumerators that had a high probability of satisfying the hypotheses of our properties required considerable analysis of Eth2.0 so as to restrict certain `tctrs` values from skewing the scores too much. Testing properties for new applications will likewise require writing new custom enumerators. One might need to write a new event generator as well, possibly generating events of shapes different from the ones we showed.

6 Related Work

The Protocol Labs ResNetLab and software audit firm Least Authority tested GossipSub against a list of specific pre-programmed attack scenarios [59] designed to degrade overall network performance using a network emulator called TESTGROUND [7]. Due to their use of simplified configurations with only one topic (and because simple testing is not enough to find bugs) they found that all the attacks failed against

GossipSub, and the score function made GossipSub more resilient to malicious nodes attacks than the other tested protocols [58]. Separate from simulation testing, Least Authority also audited the Golang implementation and provided recommendations for improvement [36]. Though our work contributes the first formalization of GossipSub, there has been considerable previous work on utilizing formal methods to reason about distributed systems. We survey such works below.

Model Checking based approaches. Lamport's modeling language TLA+ [33] and the corresponding TLC model checker [63] have been used to analyze properties of distributed systems including DISK PAXOS [23], MONGORAFTRECONFIG [54], Byzantine PAXOS [34], SPIRE [30], etc. McMillan and Zuck applied specification-based testing to the QUIC protocol, and found vulnerabilities [49]. Wu et. al. formally modeled the Bluetooth stack using PROVERIF, a model checker, and found five known vulnerabilities and two new ones [62]. Chothia et. al. demonstrated the use of PROVERIF to verify distance-bounding protocols, *e.g.*, those used by MasterCard and NXP [20]. Separately, Chothia modeled the MUTE anonymous file-sharing system using the π-calculus, and proved the system insecure (discovering a novel attack) [19]. Cremers et. al. modeled all handshake modes of TLS 1.3 using TAMARIN, another model checking tool, and discovered an unexpected behavior [21]. An issue with using model checking tools like ProVerif or Tamarin to verify a protocol like GossipSub is the immense size of the state space needed to be checked, making them infeasible for our use.

Refinement-based proof formalization. The theory of refinement has proved to be useful for enabling the mechanical verification of distributed systems' properties. Manolios' work on refinement [38, 40, 39] has been previously used for mechanical verification of pipelined processors [37, 41, 42, 44, 43]. Manolios et. al. combined theorem proving (using refinement maps) with model checking to verify the alternating bit protocol. [40]. IRONFLEET [25] refines TLA style state-machine specification of a PAXOS-based library and a sharded key-value store to low level implementation in Dafny (a SMT based program verifier) for verification using Hoare-logic. Woo et. al. [61] formally verified 90 properties of the RAFT protocol using VERDI [60], a tool they built in the COQ proof assistant. Though they did not build an executable model, their framework can be used to extract an executable protocol implementation in OCaml. VERDI provides verified *system transformers* used to refine a system in an ideal fault model to a more realistic fault model, without any proof overhead on part of the user. We believe that looking at GossipSub through the lens of refinement will be interesting because, not only will it allow us to explain why it failed our properties, but also guide us towards improving it.

Inductive-invariant based proof formalization. Padon et. al. [51] proved the correctness of a simple model of Paxos described in Effectively Propositional Logic (a decidable fragment of First Order Logic) using IVY [52], a SMT-based safety verification tool. IVY can be used for verifying inductive invariants about global states of a distributed protocol. Both the modeling and the specification languages of IVY are restricted to a decidable fragment of First Order Logic to ensure that all verification conditions can be checked algorithmically. Hippel et. al. [26] also used IVY to formally describe and reason about Karn's Algorithm, a mechanism used to study rount trip times of message transmissions. However, since IVY lacks a theory of rationals, modelling the scoring function of GossipSub would not have been possible using this tool.

Full stack verification. Certain high-assurance distributed systems might require the whole stack to be formally-verified. Such applications could, for instance, be implemented on top of SEL4: a high-performance operating system microkernel that was formally verified against an abstract specification using higher-order logic [29]. Another example is the fully verified *CLI stack* [8], a system comprising of an operating system with some applications running in it, operational semantics for two high level languages, a stack based assembly language and the instruction set architecture (ISA), all the way down to the register transfer level (RTL) design for a microprocessor. The full stack was verified in Nqthm [11].

7 Conclusion and Future Work

In this paper, we described the GossipSub protocol, as well as our official formalization based on its prose specification using the ACL2s theorem prover. We explained our state models, transition functions as well as design decisions. We showed our security property for GossipSub and how we were able to find counter-examples against it. Finally, we described several kinds of attacks we synthesized based on our attack gadgets, using the couter-examples as specifications.

In the future, we would like to characterize an ideal variant of GossipSub as a refinement of simpler protocols so as to prove safety properties, as well as to contrast the ideal variant with our current model in order to better explain why it is susceptible to attacks from misbehaving peers. We would also like to support reasoning for the application layer on top of our network model layer, since interesting bugs can be found at the interface of these two layers.

References

[1] *ACL2 Sources*. `https://github.com/acl2/acl2`. Accessed 12 June 2023.

[2] *Code walk of the GossipSub ACL2s formalization*. `https://github.com/maxvonhippel/gossipSub-FM/blob/main/model/demo.lisp`. Accessed 30 May 2023.

[3] *GossipSub Model and attacks*. `https://github.com/ankitku/gsacl2ws`. Accessed 22 July 2023.

[4] GO-LIBP2P-PUBSUB. `https://github.com/libp2p/go-libp2p-pubsub`.

[5] *What is Publish/Subscribe*. `https://docs.libp2p.io/concepts/pubsub/overview/`. Accessed 12 May 2023.

[6] (2017): *Filecoin: A Decentralized Storage Network*. `https://filecoin.io/filecoin.pdf`.

[7] (2022): TESTGROUND. `https://docs.testground.ai/`. Accessed 24 July 2022.

[8] William Bevier, Warren Hunt, Jstrother Moore & William Young (1989): *Special issue on system verification*. *Journal of Automated Reasoning*.

[9] Bruno Blanchet (2016): *Modeling and verifying security protocols with the applied pi calculus and ProVerif*. *Foundations and Trends® in Privacy and Security*, doi:10.1561/3300000004.

[10] Paul Bonsma (2010): *Most balanced minimum cuts*. *Discrete Applied Mathematics*, doi:10.1016/j.dam.2009.09.010.

[11] R.S. Boyer, M. Kaufmann & J.S. Moore (1995): *The Boyer-Moore theorem prover and its interactive enhancement*. *Computers and Mathematics with Applications*, doi:10.1016/0898-1221(94)00215-7. Available at `https://www.sciencedirect.com/science/article/pii/0898122194002157`.

[12] Vitalik Buterin (2014): *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. `https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf`. Accessed 13 July 2022.

[13] Harsh Chamarthi, Peter C. Dillinger, Panagiotis Manolios & Daron Vroon (2011): *The "ACL2" Sedan Theorem Proving System*. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, doi:10.1007/978-3-642-19835-9_27.

[14] Harsh Raju Chamarthi (2016): *Interactive Non-theorem Disproving*. Ph.D. thesis, Northeastern University, doi:10.17760/D20467205.

[15] Harsh Raju Chamarthi, Dillinger Peter C., Matt Kaufmann & Panagiotis Manolios (2011): *Integrating testing and interactive theorem proving*. doi:10.4204/EPTCS.70.1.

[16] Harsh Raju Chamarthi, Dillinger Peter C. & Panagiotis Manolios (2014): *Data Definitions in the ACL2 Sedan*. *ACL2*.

[17] Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann & Panagiotis Manolios (2011): *Integrating Testing and Interactive Theorem Proving*. In David S. Hardin & Julien Schmaltz, editors: *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications*, *EPTCS* 70, pp. 4–19, doi:10.4204/EPTCS.70.1.

[18] Harsh Raju Chamarthi & Panagiotis Manolios (2011): *Automated specification analysis using an interactive theorem prover*. In Per Bjesse & Anna Slobodová, editors: *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11*, FMCAD Inc., pp. 46–53. Available at `http://dl.acm.org/citation.cfm?id=2157665`.

[19] Tom Chothia (2006): *Analysing the MUTE anonymous file-sharing system using the pi-calculus*. In: *International Conference on Formal Techniques for Networked and Distributed Systems*, doi:10.1007/11888116_9.

[20] Tom Chothia, Joeri De Ruiter & Ben Smyth (2018): *Modelling and analysis of a hierarchy of distance bounding attacks*. In: *27th USENIX Security Symposium (USENIX Security 18)*.

[21] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott & Thyla van der Merwe (2017): *A Comprehensive Symbolic Analysis of TLS 1.3*. In: *Conference on Computer and Communications Security*, doi:10.1145/3133956.3134063.

[22] Peter C. Dillinger, Panagiotis Manolios, Daron Vroon & J Strother Moore (2007): *ACL2s: "The ACL2 Sedan"*. In: *International Conference on Software Engineering (ICSE)*, doi:10.1109/ICSECOMPANION.2007.14.

[23] Eli Gafni & Leslie Lamport (2003): *Disk paxos*. *Distributed Computing*, doi:10.1007/s00446-002-0070-8.

[24] David A. Greve, Matt Kaufmann, Panagiotis Manolios, J. Strother Moore, Sandip Ray, José-Luis Ruiz-Reina, Robert W. Sumners, Daron Vroon & Matthew Wilding (2008): *Efficient execution in an automated reasoning environment*. *J. Funct. Program.*, doi:10.1017/S0956796807006338.

[25] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty & Brian Zill (2015): *IronFleet: proving practical distributed systems correct*. In: *Proceedings of the 25th Symposium on Operating Systems Principles*, doi:10.1145/2815400.2815428.

[26] Max von Hippel, Kenneth L. McMillan, Christina Nita-Rotaru & Lenore D. Zuck (2023): *A Formal Analysis of Karn's Algorithm*. In: *Networked Systems*, doi:10.1007/978-3-031-37765-5_4.

[27] Matt Kaufmann & J Strother Moore (1996): *ACL2: An industrial strength version of Nqthm*. In: *Proceedings of 11th Annual Conference on Computer Assurance (COMPASS)*, doi:10.1109/CMPASS.1996.507872.

[28] Matt Kaufmann & J Strother Moore (2022): *ACL2 homepage*. Available at `https://www.cs.utexas.edu/users/moore/acl2/`.

[29] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski & Michael Norrish (2009): *seL4: Formal verification of an OS kernel*. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, doi:10.1145/1629575.1629596.

[30] Emil Koutanov (2021): *Spire: A Cooperative, Phase-Symmetric Solution to Distributed Consensus*. *IEEE Access*, doi:10.1109/ACCESS.2021.3096326.

[31] Ankit Kumar, Max von Hippel, Pete Manolios & Cristina Nita-Rotaru (2022): *Formal Model-Driven Analysis of Resilience of GossipSub to Attacks from Misbehaving Peers*. *arXiv preprint arXiv:2212.05197*, doi:10.48550/arXiv.2212.05197.

[32] Leslie Lamport (1978): *The implementation of reliable distributed multiprocess systems*. *Computer Networks*, doi:10.1016/0376-5075(78)90045-4.

[33] Leslie Lamport (2002): *Specifying systems: the TLA+ language and tools for hardware and software engineers*.

[34] Leslie Lamport (2011): *Byzantizing Paxos by refinement*. In: *International symposium on distributed computing*, doi:10.1007/978-3-642-24100-0_22. TLA+ proof available at `https://lamport.azurewebsites.net/tla/byzpaxos.html`, accessed 29 July 2022.

[35] Kai Li, Yuzhe Tang, Jiaqi Chen, Yibo Wang & Xianghong Liu (2021): *TopoShot*. In: *Internet Measurement Conference*, doi:10.1145/3487552.3487814.

[36] Dylan Lott (2020): *Audit of Gossipsub v1.1 Protocol Design + Implementation for Protocol Labs*. `https://leastauthority.com/blog/audit-of-gossipsub-v1-1-protocol-design-implementation-for-protocol-labs/`. Accessed 3 March 2022.

[37] Panagiotis Manolios (2000): *Correctness of Pipelined Machines*. In: *Formal Methods in Computer-Aided Design, FMCAD*, doi:10.1007/3-540-40922-X_11.

[38] Panagiotis Manolios (2001): *Mechanical Verification of Reactive Systems*. Ph.D. thesis, The University of Texas at Austin, Department of Computer Sciences, Austin TX.

[39] Panagiotis Manolios (2003): *A Compositional Theory of Refinement for Branching Time*. In: *Correct Hardware Design and Verification Methods, CHARME*, doi:10.1007/978-3-540-39724-3_28.

[40] Panagiotis Manolios, Kedar S. Namjoshi & Robert Summers (1999): *Linking Theorem Proving and Model-Checking with Well-Founded Bisimulation*. In: *Computer Aided Verification, CAV*, doi:10.1007/3-540-48683-6_32.

[41] Panagiotis Manolios & Sudarshan K. Srinivasan (2004): *Automatic Verification of Safety and Liveness for XScale-Like Processor Models Using WEB Refinements*. In: *Design, Automation and Test in Europe Conference and Exposition, DATE*, doi:10.1109/DATE.2004.1268844.

[42] Panagiotis Manolios & Sudarshan K. Srinivasan (2005): *Refinement Maps for Efficient Verification of Processor Models*. In: *Design, Automation and Test in Europe Conference and Exposition, DATE*, doi:10.1109/DATE.2005.257.

[43] Panagiotis Manolios & Sudarshan K. Srinivasan (2008): *Automatic verification of safety and liveness for pipelined machines using WEB refinement*. *ACM Trans. Design Autom. Electr. Syst.*, doi:10.1145/1367045.1367054.

[44] Panagiotis Manolios & Sudarshan K. Srinivasan (2008): *A Refinement-Based Compositional Reasoning Framework for Pipelined Machine Verification*. *IEEE Trans. Very Large Scale Integr. Syst.*, doi:10.1109/TVLSI.2008.918120.

[45] Panagiotis Manolios & Daron Vroon (2003): *Algorithms for Ordinal Arithmetic*. In: *Conference on Automated Deduction CADE*, doi:10.1007/978-3-540-45085-6_19.

[46] Panagiotis Manolios & Daron Vroon (2004): *Integrating Reasoning about Ordinal Arithmetic into ACL2*. In: *Formal Methods in Computer-Aided Design FMCAD*, LNCS, Springer–Verlag, doi:10.1007/978-3-540-30494-4_7.

[47] Panagiotis Manolios & Daron Vroon (2005): *Ordinal Arithmetic: Algorithms and Mechanization*. *Journal of Automated Reasoning*, doi:10.1007/s10817-005-9023-9.

[48] Panagiotis Manolios & Daron Vroon (2006): *Termination Analysis with Calling Context Graphs*. In: *Computer Aided Verification CAV*, doi:10.1007/11817963_36.

[49] Kenneth L. McMillan & Lenore D. Zuck (2019): *Formal specification and testing of QUIC*. In: *ACM Special Interest Group on Data Communication (SIGCOMM)*, doi:10.1145/3341302.3342087.

[50] Simon Meier, Benedikt Schmidt, Cas Cremers & David Basin (2013): *The TAMARIN prover for the symbolic analysis of security protocols*. In: *International conference on computer aided verification*, doi:10.1007/978-3-642-39799-8_48.

[51] Oded Padon, Giuliano Losa, Mooly Sagiv & Sharon Shoham (2017): *Paxos made EPR: decidable reasoning about distributed protocols*. *Proceedings of the ACM on Programming Languages*, doi:10.1145/3140568.

[52] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv & Sharon Shoham (2016): *Ivy: safety verification by interactive generalization*. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, doi:10.1145/2908080.2908118.

[53] Fred B. Schneider (1990): *Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial*. *ACM Comput. Surv.*, doi:10.1145/98163.98167.

[54] William Schultz, Ian Dardik & Stavros Tripakis (2022): *Formal verification of a distributed dynamic reconfiguration protocol*. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, doi:10.1145/3497775.3503688.

[55] Dimitris Vyzovitis: *gossipsub: An extensible baseline pubsub protocol*. `https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/README.md`. Accessed 28 Nov 2022.

[56] Dimitris Vyzovitis (2020): *GossipSub v1.0: An extensible baseline pubsub protocol*. `https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.0.md`. Accessed 17 May 2022.

[57] Dimitris Vyzovitis (2020): *GossipSub v1.1: Security extensions to improve on attack resilience and bootstrapping*. `https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.1.md`. Accessed 3 March 2021.

[58] Dimitris Vyzovitis, Yusef Napora, Dirk McCormick, David Dias & Yiannis Psaras (2020): *GossipSub: Attack-resilient message propagation in the Filecoin and ETH2. 0 networks*. *arXiv preprint arXiv:2007.02754*.

[59] Dimitris Vyzovitis, Yusef Napora, Dirk McCormick, David Dias & Yiannis Psaras (2020): *Gossipsub-v1.1 Evaluation Report*. `https://gateway.ipfs.io/ipfs/QmRAFP5DBnvNjdYSbWhEhVRJJDFCLpPyvew5GwCCB4VxM4`. Accessed 21 May 2022.

[60] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst & Thomas Anderson (2015): *Verdi: a framework for implementing and formally verifying distributed systems*. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, doi:10.1145/2737924.2737958.

[61] Doug Woos, James R Wilcox, Steve Anton, Zachary Tatlock, Michael D Ernst & Thomas Anderson (2016): *Planning for change in a formal verification of the Raft consensus protocol*. In: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, doi:10.1145/2854065.2854081.

[62] Jianliang Wu, Ruoyu Wu, Dongyan Xu, Dave Jing Tian & Antonio Bianchi (2022): *Formal Model-Driven Discovery of Bluetooth Protocol Design Vulnerabilities*. doi:10.1109/SP46214.2022.9833777.

[63] Yuan Yu, Panagiotis Manolios & Leslie Lamport (1999): *Model checking TLA+ specifications*. In: *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, doi:10.1007/3-540-48153-2_6.