IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 24 March 2024, accepted 8 April 2024, date of publication 16 April 2024, date of current version 23 April 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3389551

RESEARCH ARTICLE

Formal Verification of Justification and Finalization in Beacon Chain

HAMRA AFZAAL[1,2], NAZIR AHMAD ZAFAR[2], AQSA TEHSEEN[1], SHAHEEN KOUSAR[1], AND MUHAMMAD IMRAN[3], (Member, IEEE)
[1]Department of Computer Science, Information Technology University, Lahore 54000, Pakistan
[2]Department of Computer Science, COMSATS University Islamabad, Sahiwal Campus, Sahiwal 57000, Pakistan
[3]Center for Smart Analytics, Institute of Innovation, Science and Sustainability, Federation University Australia, Beriwick, VIC 3806, Australia

Corresponding author: Hamra Afzaal (hamraafzaal@hotmail.com)

This work was supported by the Ethereum Foundation under Grant FY22-0695.

ABSTRACT In recent years, Beacon Chain known as the core of Ethereum 2.0, has gained considerable attention since its launch. Many validators have staked billions of Ether in the Proof of Stake (PoS) network. It is a mission critical system and its security and stability rely on the justification and finalization of checkpoints. These are essential elements of the Casper FFG consensus algorithm utilized by the Beacon Chain. This process is critical for establishing a trustworthy foundation and finalizing proposed blocks by confirming agreed upon checkpoints. Hence, ensuring the correctness of checkpoints in the Beacon Chain has significant importance because any bug in it can cause serious implications. To address this challenge, we employ formal methods, a popular mathematical approach used for verifying the correctness of such critical systems. In this work, we have done formal verification of the processes of Beacon Chain state initialization, justification and finalization of checkpoints using the Process Analysis Toolkit (PAT) model checker. The adoption of model checking through the PAT model checker presents a novel contribution of our work, as this approach is not previously utilized in the formal verification of Beacon Chain. The presented work is specified through the Communicating Sequential Programs, formal specification language, and the properties are described through Linear Temporal Logic. The PAT model checker takes the specified formal model and properties as input to assess whether the properties are satisfied. The properties are analyzed with respect to the verification time, visited states, total transitions, and memory used. Through this research, we aim to increase confidence in the correctness and reliability of the Beacon Chain.

INDEX TERMS Ethereum 2.0, beacon chain, justification and finalization, model checking, PAT.

I. INTRODUCTION

Blockchain is an emerging technology that has received the attention from academia and industry due to its several features, such as transparency, immutability, decentralized and distributed nature, better security and privacy. It differs from traditional systems in storing data. In blockchain, the information is stored in blocks that are linked together in a chain using cryptographic techniques. Blockchain technology is considered as secure due to its immutable nature; once a transaction or record is officially approved in the chain, it cannot be altered. Because of the impressive features of blockchain, it is widely used in various safety and security systems, including, finance, healthcare, retail, real estate, supply chain, and Internet of Things (IoT) [1].

It was officially introduced in 2009 by the launch of its inaugural application, i.e., Bitcoin, that was based on Proof of Work (PoW) consensus protocol. However, PoW is quite energy-intensive and require sufficient computational resources. Nowadays, Ethereum blockchain is becoming more popular due to its transition from PoW to Proof of Stake (PoS) consensus protocol and improved scalability. It includes several other important features, such as smart contracts and decentralized applications (DApps). The evolution of Ethereum 1.0 (PoW based) to Ethereum 2.0 has brought forth a new component known as the Beacon Chain.

The associate editor coordinating the review of this manuscript and approving it for publication was Mehdi Sookhak.

Ethereum 2.0 is referred as Eth2 or Serenity, that includes a PoS consensus mechanism, shard chains, and various other enhancements. It plays a significant role in resolving the scalability challenges faced by traditional blockchains and coordinating validator activities. The primary responsibilities of Beacon Chain include monitoring information regarding validators, their staked assets, and attestations called votes. It implements penalties such as slashing the validators found to engage in dishonest behavior, resulting in the reduction of their staked assets.

In Beacon Chain, the concept of *Epoch* is quite important. An epoch is a unit of time that is divided into a fixed number of slots during which various activities are processed, for example, block proposals, attestations, and justification and finalization of checkpoints. The duration of an epoch is determined by the total number of slots it encompasses, and in Beacon Chain each epoch consists of 32 slots. Each slot has a designated proposer responsible for creating a block, and a validators committee who has the opportunity to perform other consensus related activities, as shown in Figure 1. The epoch processing function in the Beacon Chain is responsible for managing various aspects of the network. The justification and finalization of checkpoints are integral steps that ensure the security, efficiency, and stability of the Ethereum network.

A checkpoint is a block positioned at the initial slot of an epoch. In the cases where such a block is not present, the checkpoint is set as the most recent preceding block by default. Each epoch is associated with exactly one checkpoint block, which may serve as the checkpoint for multiple epochs [2]. When a validator casts an LMD GHOST (Latest Message Driven Greediest Heaviest Observed SubTree) vote [3], it simultaneously casts a vote for the checkpoint within its ongoing epoch, known as the target. This voting action is referred to as a Casper FFG (the Friendly Finality Gadget) vote [4], encompassing not only the current checkpoint (target) but also a preceding checkpoint referred to as the source. At the end of an epoch, if its checkpoint secures 2/3 attestation votes from the validators, the checkpoint becomes justified and later finalized. The concepts of justification and finalization are illustrated in the Figure 2. A block (B') can be justified under two conditions: (i) genesis block is justified, (ii) if B_0 is already justified, and has a supermajority link from B_0 to B', i.e., the attestations are equivalent to at least 2/3 of the validators stake. Similarly, the block B' achieves finalization if at least the next checkpoint block, i.e., C' is justified and all its preceding blocks are also justified.

As Beacon Chain is the main element of Ethereum 2.0, that controls the network of stakers and their critical activities, hence its correctness is important for the integrity of the entire network. Any inaccuracies or vulnerabilities in its operation could lead to severe consequences, including financial losses and compromised trust. Formal methods are important mathematical techniques to ensure the correctness of such mission-critical systems. Unlike traditional testing techniques, formal methods employ mathematical evidence

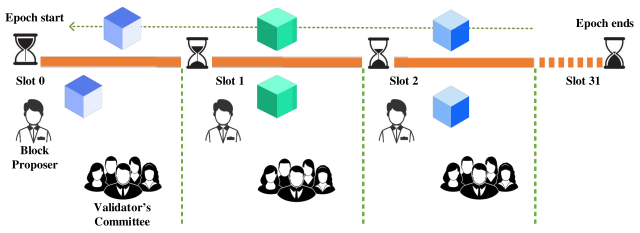


FIGURE 1. A scenario of epoch in the beacon chain.

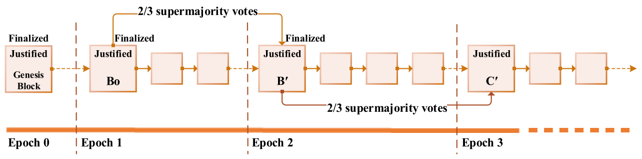


FIGURE 2. The checkpoint scenario of beacon chain.

to ensure the correctness of safety and security-critical systems. It plays a crucial role in identifying errors that might remain unnoticed or unverified otherwise.

Model checking is the most popular formal methods-based technique that is widely used for formal verification. One of the notable advantages of model checking is its ability to provide a systematic and automated way to verify the correctness of complex systems. When a system fails to meet a desired property, the model checker generates a counter-example illustrating the incorrect behavior. This offers insights to comprehend the specific cause of the failure and provides important clues for addressing the issue. In the context of blockchain and Beacon Chain technologies, it is important to employ the model checking technique to assure the correctness and security of decentralized systems. It enables the researchers to systematically analyze and validate critical aspects, such as consensus algorithms and cryptographic implementations. It contributes to the overall robustness and reliability of these technologies.

Because of the importance of formal methods, Ethereum Foundation has supported various projects related to applying formal methods to analyze Beacon Chain. For example, in 2019, a foundational project was undertaken by the Runtime Verification Inc. that provide a executable and formal semantics to the reference implementation in the K framework [5]. The use of K framework for the formal verification of Beacon Chain encounters challenges due to its complexity, particularly in modeling Python language features. The underlying Gasper and Casper FFG protocols [3], [4] were formally specified and verified by the Runtime Verification Inc. in Coq [6], [7]. Coq is a powerful formal verification tool but requires substantial learning for users which demands expertise in dependent type theory and functional programming paradigms to effectively utilize its capabilities. A verification friendly programming language, Dafny, was used to introduce a formally verified model for the Beacon Chain reference implementation [8]. However, it faces limitations in terms of scalability when applied to large-scale systems. It does not offer comprehensive

TABLE 1. Methodology of the proposed work.

Steps	Description
1. Analysis	At first, we have analyzed the Python reference implementation of Beacon Chain.
2. Flow diagrams and algorithms	Then, we have described the flow diagrams and algorithms to develop the better understanding of its reference implementation.
3. Formal model	At this level, we have transformed the algorithms into the formal model of the Beacon Chain focusing on state initialization, justification, and finalization processes using a formal methods-based specification language, i.e., CSP#.
4. Specify properties	At next, we have specified the safe initialization, justification, and finalization properties through a property specification language, i.e., LTL.
5. Formal verification	Finally, we have verified the specified properties against the formal model using the PAT model checker.

solutions for all verification challenges, necessitating careful consideration of its applicability to specific projects.

Unlike the previous projects, the originality and innovation of our project lies in the use of model checking technique for the formal verification of Beacon Chain that is not employed in the previous projects of formal verification of Beacon Chain. It is renowned for its automated nature that offers a systematic and thorough approach to uncover design errors effectively ensuring the correctness of complex systems. Furthermore, it provides several advantages over traditional verification methods. For example, its automated nature significantly reduces the likelihood of human error, ensuring a more robust and reliable verification process. It offers comprehensive coverage like systematically exploring all possible system states and transitions to identify potential issues that might otherwise go unnoticed.

Some preliminary work on the justification and finalization processes of Beacon Chain is done using SPIN model checker [9]. The process of validator's voluntary exit is formally analyzed in our prior work using the SPIN model checker [10]. This work is specifically focused on the formal specification and verification of the justification and finalization of checkpoints in the Beacon Chain. The Communicating Sequential Programs (CSP#) is used to formally specify the state initialization, justification, and finalization processes because it quite expressive to model various systems. The properties of safe initialization, justification, and finalization are described using Linear Temporal Logic (LTL). The Process Analysis Toolkit (PAT) model checker is employed to formally verify the defined properties against the developed formal model. The PAT model checker is used because it is a versatile tool designed for modeling and analyzing distributed systems.

The methodology that we have adopted in this work is improved as described in Table 1. Our contribution is significant because the previous projects on the formal verification of Beacon Chain [5], [6], [7], [8] do not provide such a detailed analysis, for example, algorithms, description of formal model and properties. We have done it critically to analyze each aspect of the Python reference implementation of Beacon Chain to ensure its correctness.

The major contributions of this work are listed below.

1) We have described the algorithms for the state initialization, justification and finalization based on the provided Python reference implementation of the Beacon Chain [11].
2) We have employed a formal methods-based specification language, i.e., Communicating Sequential Programs (CSP#) to formally specify the state initialization, justification, and finalization processes.
3) Linear Temporal Logic (LTL) is used to define the properties of safe initialization, justification, and finalization.
4) The Process Analysis Toolkit (PAT) is utilized for the verification of the properties against the specified formal model.

The scientific depth of our work is evident in our thorough examination of a specific aspect of Beacon Chain technology. Through the utilization of advanced formal verification techniques, we ensure a rigorous analysis of the Beacon Chain processes. The importance of verifying the correctness of the Beacon Chain due to its critical role in the Ethereum 2.0 network emphasizes the significance of our research in enhancing the reliability and security of the Beacon Chain. The novelty of our work is the use of model checking technique through the PAT model checker that is not previously applied for the formal verification within the Beacon Chain domain. The previous projects are based on theorem proving technique that has evident limitations. Hence, it signifies the innovation of our work from conventional approaches in verifying the Beacon Chain protocols. The details of our methodology as presented in Table 1 and findings can be seen in Section III, IV, V, and VI.

The rest of the paper is organized as follows: the background of blockchain, Ethereum, Beacon Chain and formal verification is provided in Section II; the literature review is discussed in Section III; the system model and algorithms of the state initialization, justification, and finalization are presented in Section IV; Section V describes the formal specification of the defined algorithms; the results are analyzed in Section VI; the paper is concluded in Section VII.

II. BACKGROUND

This section provides the background of blockchain technology including its consensus protocols, Ethereum and Beacon Chain, along with a discussion of formal verification through model checking approach.

A. BLOCKCHAIN

Blockchain technology is typically linked with cryptocurrencies such as Bitcoin and Ethereum. The inception of blockchain can be traced back to Nakamoto's Bitcoin white paper [12], which introduced a novel decentralized cryptocurrency. Blockchain serves as a decentralized and distributed ledger technology designed for secure and transparent transaction record-keeping across a network of computers. Generally centralized networks operate under a singular

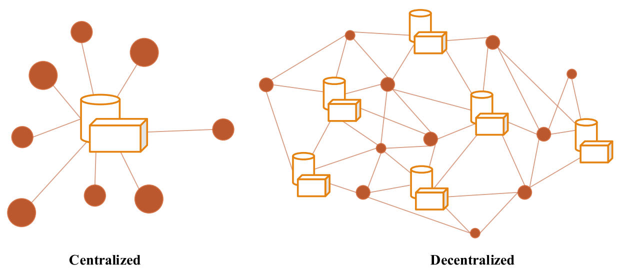


FIGURE 3. An example of centralized vs. decentralized network.

authority, while decentralized networks distribute authority across multiple nodes. The decentralized approach enhances transparency and reduces the risk of failures by eliminating a central point of control. The generic illustration of centralized and decentralized networks is shown in Figure 3. Blockchain functions as a decentralized network, eliminates the need for a central authority by distributing a shared ledger across a network of nodes. When a node wants to initiate a transaction, it generates a transaction that remains in the transaction pool. Before adding to the blockchain, it requires approval from the majority of nodes. Other nodes then retrieve the transaction from the pool, validates it, and includes it in a block, which is subsequently broadcast throughout the network. The participants in the blockchain network are referred to as miners and are responsible for verifying, organizing, and sequencing transactions into a block. The interlinked blocks form a chain, and the decentralized nature of the network ensures that no single entity has control over the entire network. It promotes security and trust in digital transactions.

Each block contains the hash of the preceding block, creating a chronological link that forms a blockchain. Figure 4 provides an overview of the blockchain structure, the blocks with their hash, the hash of the previous block, timestamp, nonce, and transaction records. The utilization of the hash from the previous block acts as a barrier to prevent unauthorized modifications to the block and the insertion of new blocks between existing ones. The decentralized blockchain consensus mechanisms, such as PoW or PoS, ensure agreement on the validity of transactions without centralized control. This architectural design guarantees that altering the contents of a particular block, such as transactions within it, would require modifications to the hash values of subsequent blocks in the chain [13].

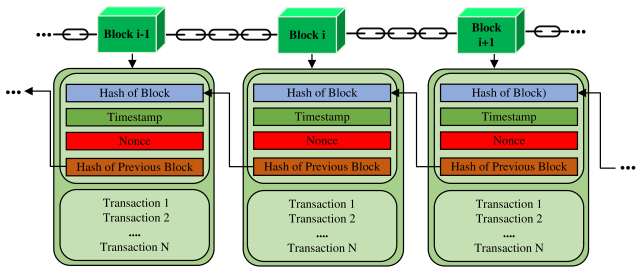


FIGURE 4. A generic architecture of blockchain.

B. BLOCKCHAIN-BASED CONSENSUS PROTOCOLS

A consensus mechanism refers to an approach used in blockchain networks to achieve agreement among nodes on the validity of transactions and the state of the distributed ledger. Two of the most used consensus techniques in the context of blockchain and cryptocurrencies are PoW and PoS. Achieving a flawless consensus protocol remains challenging due to the delicate trade-off between consistency, availability, and security [14]. PoW is the foundational blockchain protocol that requires a substantial amount of computational effort from nodes where each node has to be consistently active. In PoW, miners compete to solve complex mathematical puzzles. The participant who successfully solves the puzzle is granted the privilege to append a new block to the blockchain and is rewarded accordingly. PoW has been implemented in many cryptocurrencies such as Bitcoin, Litecoin, and Ethereum [15].

PoS is an alternative consensus mechanism where validators create new blocks and validate transactions based on the amount of cryptocurrency they have and are willing to stake as collateral. PoS differs from PoW as it selects nodes to create new blocks based on held stakes rather than computational power. PoS is regarded as more energy-efficient than PoW, as it doesn't require a high level computational power. Similar to PoW, PoS exhibits probabilistic finality. PoS has been employed in various cryptocurrencies, such as Nxt [16] and Ouroboros [17]. PPcoin introduced coin age as a factor in solving PoS puzzles [18]. Ethereum is strategically shifting from PoW to PoS. This shift is driven by the pursuit of enhanced scalability and environmental sustainability [19]. In the Practical Byzantine Fault Tolerance (PBFT) protocol, all participating nodes engage in a voting process to append the next block to the existing blockchain. When the majority of nodes reach an agreement to add a block to the chain, the selected block is incorporated into the chain. PBFT considers the majority as $2/3$ of the total participating nodes, enabling it to function properly even in the presence of up to $1/3$ malicious nodes. PBFT achieves consensus more swiftly and economically compared to PoW, making it particularly suitable for private blockchains like Hyperledger projects [20]. However, it is less recommended for public blockchains due to scalability limitations [21]. A comprehensive survey of blockchain based consensus protocols is presented in [22] and [23].

C. ETHEREUM

Ethereum was launched in 2015 by Buterin [24] and stands as a pioneering blockchain platform to expand the capabilities of blockchain technology beyond simple transactions [25]. The robustness and security of cryptocurrencies are based on the implementation of strong consensus algorithms [26]. Ethereum has emerged as a significant evolution, introducing features that set it apart. Ethereum is established on the decentralization principles and it differentiates itself from

Bitcoin by offering programmable and self-executing smart contracts, fostering a vibrant ecosystem.

The Ethereum 2.0 is introduced to address the scalability and energy efficiency challenges by transitioning from PoW to PoS. The Ethereum's evolution is illustrated in Figure 5. Ethereum 2.0 unfolds in a phased approach, with each stage strategically addressing key aspects of the network's evolution. Phase 0 is marked as a significant shift with the launch of the PoS Beacon Chain. The validators contribute to the network by staking Ethereum cryptocurrency in the PoS consensus mechanism. Phase 1 introduces the concept of sharding, by partitioning the Ethereum network into smaller and independent parts to improve the scalability and transaction processing capacity. Phase 1.5 is the subsequent step in Ethereum's evolution, often known as the merge. This involves eliminating the PoW mechanism and transitioning the Ethereum network to rely entirely on PoS for transaction validation and block creation. The final Phase 2.0 aims to empower existing shards with the ability to execute transactions natively, enhancing efficiency and functionality across the Ethereum network.

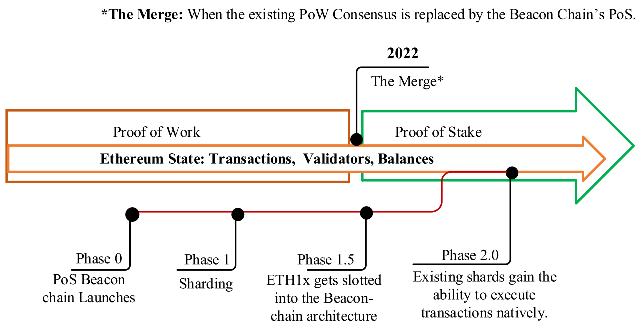

*The Merge: When the existing PoW Consensus is replaced by the Beacon Chain's PoS.
2022
The Merge*
Proof of Work
Proof of Stake
Ethereum State: Transactions, Validators, Balances
Phase 0
PoS Beacon chain Launches
Phase 1
Sharding
Phase 1.5
ETH1x gets slotted into the Beacon-chain architecture
Phase 2.0
Existing shards gain the ability to execute transactions natively.

FIGURE 5. The evolutionary stages of ethereum.

D. BEACON CHAIN

The Beacon Chain is a key component within the Ethereum 2.0, that is designed to address the scalability, security, and decentralization issues of Ethereum's original PoW consensus. During Ethereum's shift from PoW to PoS consensus mechanism, Beacon Chain serves as a separate blockchain for coordinating and managing the PoS protocol. Validators can participate in the network by staking a certain amount of cryptocurrency as collateral. The Beacon Chain employs a unique consensus mechanism, using the collective staked assets of validators to secure the network and reach an agreement on the state of the system. The validators on the Beacon Chain play a crucial role in proposing and attesting the blocks contributing to the decentralization and security of the network. It introduces the innovative concept of sharding, splitting the Ethereum network into smaller, more manageable parts called shards. Each shard independently stores its data and is responsible for handling transactions independently. Sharding facilitates parallel transaction processing, optimizing the Ethereum network's capacity. Through the implementation of sharding and PoS, Beacon Chain strives to enhance Ethereum's overall scalability, efficiency, and sustainability.

Ethereum's consensus protocol is a composite of two distinct consensus protocols, namely LMD GHOST and Casper FFG [4]. The combination of these two protocols is referred as Gasper [3]. This integration aims to achieve a balance between liveness and safety. The LMD GHOST ensures slot-by-slot liveness, sustaining the chain's continuous operation, while Casper FFG contributes safety measures protecting against extended reversions. The Casper FFG was introduced by Buterin and Griffith as a tool that defines the concepts of justification and finalization for the blockchain protocols. The Casper FFG serves as an extension to the LMD GHOST consensus protocol, adjusting its fork choice rule. The protocol defines block height, checkpoint blocks and attestations which are signed messages serving as votes to move from one checkpoint block to another. Casper introduces justification and finalization, analogous to PBFT's prepare and commits phases [27]. Casper being a finality gadget rather than a complete protocol, relies on the assumption that the underlying protocol has its fork-choice rule. In each epoch, all validators engage in running the fork-choice rule at some point to produce an attestation.

The LMD GHOST is a fork-choice rule derived from the Greediest Heaviest Observed SubTree (GHOST) [28]. When working with a tree of blocks and a set of votes, LMD GHOST determines the optimal block to be considered as the leading head of the chain. LMD GHOST serves as a fork choice rule, guiding nodes in selecting the most promising blockchain among competing branches. It relies on the latest messages from validators to determine the preferred chain. The algorithm also incorporates a weighted branch assignment, assigning weights to branches based on votes from active validators. It helps the nodes to prioritize branches with greater support and fosters consensus within the network. Fostering consensus involves the process of nodes achieving consensus on the validity of transactions and the overall state of the network. When more than 50% of nodes construct blocks on the pattern of the fork choice rule, the blockchain grows and the probability of reverting older blocks decreases exponentially. LMD GHOST doesn't provide finality, it supports a confirmation rule. This allows nodes to confirm the most promising chain based on the consensus of active validators, contributing to a more secure and reliable blockchain. The underlying concept is that honest validators construct their blocks on what they perceive as the best head, aligning their votes accordingly.

The Gasper protocol integrates the concepts of GHOST and Casper FFG. It introduces epoch boundary pairs represented as (B, E) serving as checkpoints in a chain where B denotes a block and E an epoch. Committees are formed in each epoch, assigning validators to specific slots, where each validator proposes one block per slot. Then every committee member will confirm what they observe as the leading block in the chain (ideally the recently proposed block) utilizing the fork-choice rule HLMD GHOST (a slight modification of LMD GHOST). The protocol employs the familiar concepts of justification and finalization from Casper FFG with the distinction that pairs are justified and finalized instead of individual checkpoint blocks. Justification is a process in the consensus protocol where a block is considered to be accepted or approved by the network. In Casper FFG, justification occurs when a supermajority (more than 2/3) of validators attest to the validity of a particular block. This collective agreement provides a level of confidence in the correctness of the block. Finalization goes a step further than justification. In the context of Casper FFG, a block is considered finalized when it is justified and there is a subsequent block that is also justified. Finalization implies a more robust level of agreement indicating that not only the current block but also its succession has received widespread approval from the network. Both justification and finalization are crucial for ensuring the security and integrity of the blockchain. These processes provide a mechanism for the network to collectively agree on the validity of blocks which helps to prevent the risk of malicious activities or long rollbacks.

E. FORMAL VERIFICATION

This section includes a brief explanation of the formal modeling and verification techniques. Formal methods are mathematical techniques for the specification, design, verification and analysis of software and hardware systems. These techniques aim to ensure the reliability and correctness of critical software systems through the mathematical modeling of their behavior and properties [29]. These techniques are distinguished by their precisely defined syntax and semantics. The application of formal methods is important to model, verify, and analyze mission-critical systems [30], [31]. Model checking stands out as a powerful formal methods-based technique, that systematically explores all possible states of a system to make sure that the specified properties are met. It enhances the software design reliability, plays a crucial role in preventing errors, and ensures correctness from the early stages of development. It is an automated technique that is valuable for uncovering design errors and it examines whether a given formal property holds for a finite state system model or not [32].

The model checking technique involves three phases: modeling, running and analysis that are graphically presented in Figure 6. In the modeling phase, the system is modeled unambiguously and accurately using the formal specification language of the model checker. In this work, we have used CSP# to describe the formal model. LTL is utilized to formalize the properties because it analyzes the individual paths. The running phase involves executing the model checker to validate the specified properties. The PAT model checker is used in this work. The analysis phase assesses the satisfaction or violation of the properties. If a violation occurs, the model checker generates a counter-example that

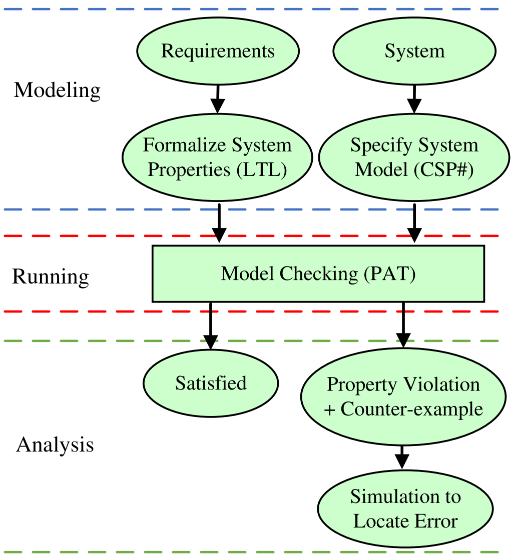

Modeling
Requirements
System
Formalize System Properties (LTL)
Specify System Model (CSP#)
Running
Model Checking (PAT)
Analysis
Satisfied
Property Violation + Counter-example
Simulation to Locate Error

FIGURE 6. Model checking approach via PAT model checker.

can be analyzed through simulation, leading to the refinement of the model or properties.

Various model checkers, including PAT [33], NuSMV [34], SPIN [35], and UPPAAL [36], are used for formal verification. In this study, the PAT model checker is used due to its efficiency in modeling, reasoning and simulating concurrent systems. CSP# is an enriched formal modeling language that is used in the PAT model checker. It integrates high-level modeling operators such as interleaving, parallel/sequential composition, and interrupts with low-level programming constructs in C# including arrays, variables, if-then-else statements, etc. The syntax of CSP# utilized in this context is defined below:

$$\begin{aligned} X_1 ::= &\ \mathrm{SKIP}(\text{termination}) \\ &\mid e \rightarrow X_1(\text{event prefixing}) \\ &\mid e\{program\} \rightarrow X_1(\text{operation prefixing}) \\ &\mid [g]\, X_1(\text{state guard}) \\ &\mid \text{if } g \text{ then } X_1 \text{ else } X_2(\text{boolean statement}) \\ &\mid c?a \rightarrow X_1(a)(\text{channel input}) \\ &\mid c!a \rightarrow X_1(\text{channel output}) \\ &\mid X_1;\ X_2(\text{sequential composition}) \\ &\mid X_1 \parallel X_2(\text{parallel composition}) \\ &\mid X_1 \ \Box\ X_2(\text{external choice}) \end{aligned}$$

here, X_1 and X_2 represent processes, g denotes a guard condition, e represents an event, program is a sequence of code statements executed atomically and c describes a synchronized communication channel.

The assertions are defined in the PAT model checker to query the system behavior e.g., deadlock-freeness, LTL and reachability. In this work, the deadlock-freeness and LTL assertions are used and are explained as follows:

Deadlock-freeness: The following assertion checks for the existence of deadlock in a process P.

#assert $P()$ deadlockfree;

Linear Temporal Logic (LTL): This assertion verifies whether a process P satisfies an LTL formula φ.

#assert $P() \models \varphi$;

The syntax of an LTL formula is expressed as:

$\varphi = e \mid p \mid \varphi_1 \wedge \varphi_2 \mid \neg\, \varphi \mid \Box\varphi \mid \Diamond\varphi \mid \bigcirc\varphi \mid \varphi_1 \cup \varphi_2$

here, e stands for an event, and p represents an atomic proposition. The symbols $\Box$, $\Diamond$, $\bigcirc$, and $\cup$ are interpreted as always, eventually, next, and until respectively.

LTL is used in this work because it is well-suited for specifying properties about sequences of states in a system along a single path. In contrast, Computation Tree Logic (CTL) is specifically designed for reasoning about branching time properties, where the system's behavior may branch into multiple possible future states at each step. It allows the specification of properties along different paths. The LTL formulas are often more intuitive and easier to understand as compared to CTL formulas, especially for users who are not familiar with formal logic or model checking techniques. By using LTL, it is ensured that specifications are clear and comprehensible to a wider audience. Many model checking tools provide robust support for LTL, offering features such as automatic translation of LTL formulas into the appropriate model checking algorithms.

III. LITERATURE REVIEW

This section provides a comprehensive overview of the blockchain research domain, including key areas, such as generic blockchain consensus protocols, Ethereum consensus protocols, and formal verification of consensus protocols and Beacon Chain.

A. POW AND POS CONSENSUS PROTOCOLS

The PoW consensus protocol implemented in the Bitcoin network has drawn much attention from the research community [12]. In the PoW consensus protocol, the network nodes have to solve a mathematical puzzle to produce a valid block. The consensus is achieved through the computational work performed by network nodes which are known as miners. The PoW consensus mechanism provides security and decentralization to blockchain networks, as it requires participants to invest computational resources and energy [37]. Despite its merits, it faces challenges including, tremendous energy consumption and limited scalability. It is prone to security attacks, such as selfish mining, 51%, and eclipse attacks. It follows the long chain rule for block finalization.

The Bitcoin backbone protocol is based on PoW and it is characterized by the common prefix and chain quality properties [38]. It relies on novel assumptions about the adversary's hashing power and network synchronicity. It ensures transaction liveness and persistence, facilitating the analysis of Bitcoin and a more complex Byzantine agreement protocol. Its security is maintained even under high network synchronicity and adversary hashing power of less than 1/2.

In contrast to PoW, PoS has emerged as a more energy efficient alternative, because the block creation process relies on possessing a sufficient stake in the system [16]. Ouroboros is based on PoS with robust security assurances [17]. It is based on secure coin flipping for randomness in leader selection. The protocol boasts security to protect against desynchronization attacks. The block finalization occurs at the end of each epoch. It incentivizes participants through a resilient Nash Equilibrium based reward mechanism. It mitigates potential attacks like selfish mining.

The PoS mechanism based on coin age, is presented in [18]. It is resistant against nothing-at-stake problem and blockchain forks. However, it is susceptible to vulnerabilities such as grinding and long-range attacks. Delegated Proof of Stake (DPoS) [39] is proposed to enhance the efficiency and security of blockchain. It introduces a downgrade mechanism to address the malicious behaviors among witnesses. It uses random voting to enhance fairness and decentralization in block generation/finalization. It implements a downgrade mechanism to swiftly handle malicious nodes, ensuring that the overall system operates effectively and securely.

B. HYBRID CONSENSUS PROTOCOLS

To address the limitations of the above-discussed consensus protocols, hybrid approaches were designed to unify the advantages and mitigate potential attacks. For instance, Proof of Elapsed Time (PoET) is introduced to tackle the energy consumption issue [40]. For a fair and efficient leader election process, it employs a trusted execution environment (TEE) and Intel Software Guard Extensions (SGX). Through TEE and SGX, a random timer value is generated and assigned to each node. The verifying nodes then sleep for a randomly generated waiting time, and the first node to complete this waiting time earns the right to create the next block. It is vulnerable to hijacking, even when a minority of nodes are compromised. The block finalization is probabilistic, and the incentive mechanism involves rewards and slashing.

Addressing the limitations of PoW, a novel consensus mechanism called Proof of Luck is proposed in [41], aiming to reduce the computational power required for transactions and to enhance throughput. Based on TEE, the algorithm is comprised of two key functions: pollround and pollmine. Each mined block is assigned a luck value, a random number between 0 and 1, with a cumulative luck value calculated through the summation of luck values within each block of the chain. Miners are prioritized with the highest luck value can append their blocks to the chain. A hybrid PoW-PoS cryptocurrency protocol to thwart 51% attacks, using a single thread system with strict time spacing for block generation is designed in [42]. This ensures a resilient consensus, fair profit distribution and prevents domination by a single user or a group.

C. BFT CONSENSUS PROTOCOL

Byzantine fault tolerance involves achieving consensus among nodes even in the presence of malicious nodes in a distributed network. Practical Byzantine fault tolerance (PBFT) is an example of BFT, which operates as a replication algorithm capable of withstanding Byzantine faults [43]. PBFT anticipates the presence of dishonest or faulty nodes and it is designed to be a high-performance consensus algorithm that relies on a set of trusted nodes within the network. In PBFT, nodes are arranged sequentially, with one designated as the leader and the remaining nodes acting as backups. PBFT offers a higher transaction throughput than PoW. It works better on a private network rather than on a public one as a result of the overhead communication. The PBFT networks are more vulnerable to Sybil attacks and are less decentralized due to the presence of central authority.

The NEO cryptocurrency platform introduces Delegated Byzantine Fault Tolerance (DBFT) as a solution to centralization challenges in PBFT [44]. In DBFT, the nodes are elected through a voting process. Among these nodes, one node is designated as a leader, and rest of the nodes act as backup nodes also called delegates. The participants in the network participate in the voting process, based on the cryptocurrency they hold. However, a potential drawback arises if all the participants vote for themselves, leading to the risk of communication problems. Despite resolving the centralization issues, DBFT still faces issues related to high communication overhead and susceptibility to sybil attacks. The idea of PoS and PBFT consensus methods serve as the foundation for the Tendermint protocol [45]. The validators are selected according to their stake, and a leader is chosen from a group of validators in a round-robin fashion. The consensus is reached with 2/3 of the network nodes, similar to PBFT. To mitigate communication overhead and uphold blockchain decentralization, the Ripple network incorporates the federated byzantine agreement protocol [46]. Unlike PBFT, FBA eliminates the need for a central authority to maintain a node list for transaction processing and validation, allowing any node to participate in consensus. The agreement of 80% of the nodes is necessary for transaction addition.

In our previous works, to address the issues of trustworthy leader and validators selection, we have proposed formally verified consensus protocols, i.e., secure and trustworthy blockchain-based crowdsourcing (STBC), trust-based blockchain crowdsourcing (TBC), and trust and transactions chain (TTC) consensus protocols [47], [48], [49]. These consensus protocols are designed based on various trust factors to increase their reliability.

D. ETHEREUM CONSENSUS PROTOCOLS

The recent Casper FFG protocol [4] is introduced as a hybrid of PoW and PoS, that aims to penalize malicious validators. In this model, PoW is used to propose blocks, while PoS is employed for finality, ensuring irreversible consensus. It achieves consensus through a staking mechanism where validators commit deposits, and finality is reached through a BFT like process. It exhibits energy efficiency and 33% fault tolerance but lacks full resistance against 51% attack. It includes security properties that penalize validators unable to finalize blocks. Block finalization occurs at the end of each epoch, and the incentive mechanism involves deposit rewards, block proposal and finalization rewards. In Casper, stakeholders operate as peer nodes who broadcast vote messages. For block finalization, it offers plausible liveness and accountable safety with tolerance up to 1/3 of the total voting power controlled by the Byzantine nodes.

The Gasper protocol [3], combines the LMD GHOST and Casper FFG to create a comprehensive PoS-based blockchain design. In Gasper, validators propose and attest to blocks based on their staked ether. This staking mechanism provides economic security and incentivizes validators to act honestly, as their stakes are at risk. Gasper eliminates the need for resource intensive PoW, making the consensus process more energy efficient and scalable in the Ethereum network. The design emphasizes a balance between simplicity, understandability, and practicality while considering safety and liveness.

Some preliminary work is done on the justification and finalization and validator's voluntary exit processes in the python reference implementation of Ethereum 2.0 Beacon Chain [9], [10]. In this work, we present a detailed analysis of the state initialization, justification and finalization of checkpoints in the Beacon Chain. We have verified these processes against specified properties including deadlock-free, safe initialization, justification, and finalization. For a comprehensive overview, we present the major properties of the consensus protocols discussed in Table 2.

E. FORMAL VERIFICATION OF PROTOCOLS

In the ever-evolving landscape of software engineering, formal verification is essential for ensuring the reliability and correctness of complex systems. Formal verification is a rigorous mathematical process for validating the correctness of a complex decentralized system's design. This section explores the recent progress of formal verification techniques utilized in blockchain consensus protocols. The formal modeling for the adversary scenarios on PoW blockchains by utilizing the BIP (Behavior, Interaction, Priority) framework is presented in [50]. By employing the statistical model checker SBIP, this approach examines diverse attacks, including DNS (Denial of Network services), double spending, and consensus delay attacks. It accounts for both stochastic and temporal interactions within blockchain systems.

The RedBelly blockchain's BFT consensus mechanism [51], is defined in threshold automata and validated by using ByMC. This approach highlights the gravity of the issue by identifying vulnerabilities in blockchain consensus along with counter-examples. The emphasis is on analyzing various properties like justification, consistency, and termination. The PBFT protocol is validated by the PRISM model checker and formalized using the Continuous-Time Markov Chain (CTMC) model to simulate its temporal response [52]. The

TABLE 2. The comparative analysis of blockchain consensus protocols.

Protocol	Main feature	Energy efficiency	Fault tolerance	Security properties	Vulnerabilities	Block finalization
PoW [12]	Solve difficult hash	No	50%	Resistant to sybil attack	51%, eclipse, selfish mining	Long-chain rule
Bitcoin backbone [38]	Analysis of common prefix and chain quality	No	50%	Resistant to selfish mining	Double-spending	–
PoS-based (Ouroboros) [17]	Secure coin-flipping for randomness	Yes	50%	Secure against 50% adversary	Desynchronization attack	At the end of each epoch
PoS-based (PPCoin) [18]	Based on coin age	Yes	50%	Resistant to sybil and 51% attack	Nothing-at-stake, collusion attack, stake grinding, and long-range	Probabilistic
DPoS [39]	Effectiveness in PoS	Yes	50%	–	Sybil, 51% and DoS	Based on votes
PoET [40]	Analysis of sawtooth lake scheme	Yes	50%	Partially secure	Hijacking/vulnerable even a minority of nodes compromised	Probabilistic
PoL [41]	Block design based on TEEs	Partial	50%	Liveness, persistence, resistant to 51%, and double-spending attack	Sybil attack	Maximum luck value
Hybrid PoW-PoS [42]	Strict timestamp spacing constraint	Yes	50%	Resistant to 51%, and double-spending attack	–	Fork-free/when majority of node agree/no validation mechanism
PBFT [43]	Tolerate faulty node	Yes	33%	Fairness	Sybil	Multi-phase commit process
DPBFT [44]	Mitigation of centralization in PBFT	Yes	33%	Resistant to sybil attack	Double-spending and 51%	Multi-phase commit process
Tendermint [45]	Tolerate malfunctioning nodes	Yes	33%	Liveness, fairness	Eclipse, sybil and DoS attack	2/3 nodes agree on block validation
FBA [46]	Use of unique node list to address communication challenges	Yes	20%	Resistant to sybil attack	Double-spending and 51% attack	Maximum nodes agree on block i.e 80%
STBC [47]	Selection of trustworthy leader and validators	Yes	33%	Safety, resistant to selfish mining	–	Based on majority of votes
TBC [48]	Selection of leader and validators based on trust	Yes	33%	Liveness and persistence	–	Based on majority of votes
TTC [49]	Verification of trustworthy selection of leaders and validators via trust chain consensus	Yes	33%	Fraud-resilience and resistant to eclipse, sybil attacks	–	Based on majority of votes
Casper FFG [4]	Hybrid of PoW and PoS, penalize malicious validators	Yes	33%	Fairness, decentralization	Lack full resistance against a 51% attack	At the end of each Epoch, 2/3 votes
Gasper [3]	Combination of Casper FFG and LMD GHOST	Yes	33%	Safety, plausible and probabilistic liveness	Long-term censorship	2/3 supermajority votes
Justification and finalization in Ethereum 2.0 [9]	Verification of justification and finalization of checkpoints	Yes	33%	Safety	–	2/3 supermajority votes
Voluntary exit of validator in Ethereum 2.0 [10]	Verification of voluntary exit process of validator	Yes	33%	Safety and liveness	–	2/3 supermajority votes
Formal verification of justification and finalization in Beacon Chain (this paper)	Verification of state transition, justification and finalization of checkpoints	Yes	33%	Deadlock-free, safe initialization, justification and finalization	–	2/3 supermajority votes

analysis considers factors like the delay of primary and replica nodes; however, it neglects to include a discussion on potential node failures. The likelihood of the entire process is influenced by internet speed, while replica nodes have a minimum effect. The streamlined version of HotStuff is formalized using TLA+, and its safety is established through the TLA Proof System (TLAPS) [53]. Furthermore, the safety properties of HotStuff undergo verification using the Ivy tool that reveal the complexity of verification through both TLAPS and Ivy. The analysis demonstrate that the protocol diverges from the conventional view instance model, opting instead for a tree model to address a fundamental issue.

TABLE 3. A comparison of various properties of formally verified blockchain based consensus protocols.

Protocol	Properties	Formal specification language	Verification tool
PoW [50]	Secure against DNS and double-spending	BIP	SBIP
BFT [51]	Consistency, justification and termination	Threshold automata	ByMC
PBFT [52]	Primary and replica nodes delay	CTMC	PRISM
HotStuff [53]	Safety	Ivy/ TLA+	Ivy/ TLAp
Tendermint [54]	Deadlock-free consensus	CSP#	PAT
Tendermint [55]	Termination and synchronization	TLA+	TLAPS
Casper [6]	Safety and non-triviality	Coq	Coq proof assistant
Finality mechanism of Gasper [7]	Accountable safety, plausible liveness and slashing lower bound	Coq	Coq proof assistant
Deposit Contract [56]	Liveness	K framework	Vyper
CBC Casper [57]	Safety and liveness	HOL	Isabelle/HOL proof assistant
SCP (FBA-Based) [58]	Safety and liveness	Timed automata	UPPAAL
Algorand [59]	Safety	Coq	Coq proof assistant
STBC [47]	Safety, fault-tolerance	CSP#	PAT
TBC [48]	Liveness and persistence	CSP#	PAT
TTC [49]	Fraud resilience	CSP#	PAT
Executable K-model of Ethereum 2.0 [5]	-	K framework	K tool
State transition (Gasper FFG) [8]	Safety	Dafny	SMT Solver Z3
Justification and finalization of checkpoints [9]	Safety	PROMELA	SPIN
Voluntary exit of validator in Ethereum 2.0 [10]	Safety and liveness	PROMELA	SPIN
Formal verification of justification and finalization in Beacon Chain (this paper)	Deadlock-free, safe initialization, justification and finalization	CSP#	PAT

The formal definition of the Tendermint consensus protocol is defined using CSP# and its verification is conducted through the PAT model checker [54]. The results indicate that the protocol is free from deadlocks and can achieve consensus with the agreement of 2/3 of the network nodes. Moreover, the verification reveals that if minority nodes surpass 1/3 of the network nodes, the network can be disrupted. However, the Tendermint protocol prioritizes consistency over liveness. The formal specification of Tendermint is conducted using TLA+, and its verification is carried out through TLAPS [55]. The Apalache model checker is employed to assess safety properties, specifically termination and synchronization. The formalization process enhances the comprehension of liveness properties, uncovering various issues that can arise from the unexpected behavior of faulty peers. The analysis emphasizes the correlation between timeout duration, end-to-end delays of messages, and block generation rate. The CBC Casper constitutes a partial specification for a family of consensus protocols. It relies on five framework parameters to define specific protocols, including participating nodes, validator weights, fault tolerance threshold, consensus values, and an estimator function. These parameters shape protocol states and messages. CBC Casper protocols exhibit Byzantine fault tolerance, specifically concerning equivocation, where nodes cannot send conflicting messages. The Ethereum consensus protocol CBC Casper is formalized using Coq proof assistant [6]. It proves the safety and non-triviality properties and uses Coq's type classes for abstraction, reveal protocol assumptions, and express these in mathematical terms.

The Coq formalization of the Gasper protocol is presented in [7]. It aligns closely with the Gasper protocol's abstraction level. This Coq formalization offers a rigorous representation, explicitly specifying assumptions crucial for plausible liveness. The key assumptions include the non-emptiness of the supermajority quorums, proper justification links, consideration of votes from the validator set of the target block, and the production of honest votes by the validators of a supermajority set. The Ethereum 2.0 security assessment investigates the deposit smart contract, provides critical insights into bytecode correctness, and reveals potential vulnerabilities [56]. The subtle bugs are identified originating from the compiler, emphasizing the importance of addressing these issues for the broader research community. The verification process focuses on a complex incremental Merkle tree algorithm, uncovering challenges such as algorithm complexity, bytecode size, and security concerns. The noteworthy findings include, critical bugs in both the contract and compiler. The trust relies on bytecode specification correctness and the KEVM verifier, supported by a partially mechanized proof of algorithm correctness.

The refinement of the CBC Casper is investigated in [57]. It emphasizes on the PoS protocol security and aims to achieve correctness particularly addressing safety and liveness in the presence of Byzantine behavior. It extensively investigates an instance called Casper, The Friendly GHOST (TFG), which is systematically formalized, and the verification process utilizes the Isabelle/HOL proof assistant. The significant contributions include, the establishment and proof of blockchain safety, adjustments to properties like governing messages and state transitions, and the proposal of an improved clique oracle for effective decision making among validators. The formal model of Stellar Consensus Protocol (SCP) for federated Byzantine agreement is presented in [58]. It highlights the significance of configuring quorums to ensure the liveness and safety of SCP, employing timed automata for modeling and the UPPAAL model checker for verification. This approach enables pre-execution verification of quorum configurations to ascertain consensus assurance before deployment on real networks. However, due to the large state space, the model undergoes abstraction, resulting in processing delays and packet overheads as validators broadcast and analyze quorum information for each blockchain block.

The Algorand blockchain employs a secure PoS consensus protocol, ensuring a single certified block per round to prevent forks. Its formal verification is done using Coq addressing the timing complexities and adversarial actions to enhance realism [59].The achieved asynchronous safety proof demonstrates that even with full control of message delivery, two different blocks cannot be certified in the same round. The model is deemed general, with the potential for extending verification to other critical properties like liveness for the Algorand protocol. In our previous works, we have proposed and formally verified the STBC, TBC, and TTC consensus protocols using the PAT model checker [47], [48], [49]. The STBC protocol verifies the safety and fault-tolerance properties, the TBC protocol verifies the persistence and liveness properties, and the TTC protocol verifies the fraud-resilience properties. In this work, our focus is on the formal verification of Ethereum 2.0 Beacon Chain by employing the model checking technique.

F. FORMAL VERIFICATION OF BEACON CHAIN

Beacon Chain serves as the backbone of Ethereum 2.0 and consists of crucial elements such as the state transition function, validator's related functions, justification and finalization of checkpoints, and various others. There are some fundamental projects that are supported by the Ethereum Foundation on the formal verification of Beacon Chain.

The first project was undertaken by the Runtime Verification Inc. to provide executable and formal semantics to the reference implementation in the K framework [5]. The model presents executable interpretations of the state transition of the Beacon Chain, facilitating direct testing against standard test suites. The use of K framework for the formal verification of Beacon Chain encounters challenges due to its complexity, particularly in modeling Python language features. The Runtime Verification Inc. also works on the formal verification of the Casper FFG and Gasper protocol using Coq [6], [7]. The work presented in [6] involves only conceptual exploration of the processes guided by specific assumptions and lacks thorough modeling of these processes. The work described in [7] utilizes the Coq proof assistant to model and verify the Gasper's finality mechanism ensuring properties at an abstract level. The findings are validated using an abstract model of quorums on a message set without explicitly specifying honest behavior and communication. The state transition function of the Beacon Chain is verified using deductive verification technique of theorem proving. All its processes are formally specified in Dafny specification language and verified through the SMT solver Z3 model checker [8]. The verification results encountered several issues including runtime errors, division by zero, underflow/overflow, and array out of range. However, it faces limitations in terms of scalability when applied to large-scale systems. Further, it is difficult to interpret the verifier feedback in case of verification failure.

All the previous projects of formal verification of Ethereum 2.0 Beacon Chain that are described above were based on theorem proving technique. This project employs the model checking technique that offers a more comprehensive and automated verification process as compared to theorem proving. Theorem proving requires significant manual effort to construct the proofs, particularly when dealing with the complexities of large-scale systems. Therefore in this research, we use model checking due to its automated nature which allows the thorough analyses of system states, ensuring that no critical property is overlooked. Some introductory work is done on the justification and finalization and the validator's voluntary exit processes using the SPIN model checker [9], [10]. In both of the works, the specifications are written in PROMELA specification language. The safety properties are defined using LTL formulas that are evaluated through the SPIN model checker. The preliminary study offers limited depth and lacks comprehensive modeling of the described processes.

This work is specifically focused on the formal verification of state initialization, justification, and finalization of the Beacon Chain in detail using the PAT model checker. To achieve this, CSP# is utilized to formally specify these processes and the properties are described using LTL formulas and are verified through the PAT model checker. The PAT model checker is used in this research work to automate

TABLE 4. A comparison of formal verification of beacon chain.

Protocol/Framework	Verification technique	Verification tool	Primary focus	Verified properties	Limitations
Executable K-model of Ethereum 2.0 [5]	Theorem proving	K-tool	State transition	-	Complexity in modeling Python language features into K, fine-grained analysis struggles with if statement constructs.
Casper [6]	Theorem proving	Coq proof assistant	Formalization of Casper at abstract level and formal proofs of properties	Safety and non-triviality	Conceptual exploration based on certain assumptions, without thorough modeling of the processes.
Finality mechanism of Gasper [7]	Theorem proving	Coq proof assistant	Finality mechanism of Gasper with dynamic set of validator	Accountable safety, plausible liveness and slashing lower bound	Abstract model without explicit definition of honest behavior and communication.
State transition (Gasper FFG) [8]	Theorem proving (Deductive verification)	SMT solver Z3	State transition (Over/underflows, array-out-of-bounds, division-by-zero)	Safety	Deductive verification is a time-consuming process and it is difficult to interpret the verifier feedback in case of verification failure.
Justification and finalization of checkpoints [9]	Model checking	SPIN	Justification and finalization	Safety	Preliminary work with limited depth and it lacks comprehensive modeling of the justification and finalization processes.
Voluntary exit of validator in Ethereum 2.0 [10]	Model checking	SPIN	Voluntary exit of validator	Safety and liveness	Initial work and lacks details.
Formal verification of justification and finalization in Beacon Chain (this paper)	Model checking	PAT	State initialization, justification and finalization	Deadlock-free, safe initialization, justification and finalization	The aspects of rewards and penalties and the complete state transition are not analyzed.

the aspects of the verification process including model generation, property specification, and verification which reduce the need of human intervention. It provides a unique combination of user-friendliness and automation in formal verification which makes it a valuable tool for analyzing and verifying concurrent systems. It provides visualization capabilities that helps to understand and analyze the behavior of the systems, facilitating the interpretation of verification results. Table 4 presents a comparative analysis of the work on formal verification of Beacon Chain, highlighting the properties, languages, tools, and limitations.

IV. SYSTEM MODEL OF JUSTIFICATION AND FINALIZATION

This section provides the basic definitions, system model, and algorithms of initialization of beacon state from Eth1, justification and finalization.

A. DEFINITIONS

At the first, some terminologies to understand the system are defined.

In a view V_i, $J(V_i)$ represents the set of justified checkpoints, and $F(V_i)$ represents the set of finalized checkpoints. Here E represents the current epoch and E' denotes the previous epoch.

Definition 1 (Supermajority Link): A supermajority link refers to a connection between two pairs (B_1, E') and (B_2, E). The link is established when attestations with the checkpoint edge $(B_1, E') \xrightarrow{V} (B_2, E)$ are more than $\frac{2}{3}$ of the total validator stake.

We can write this as $(B_1, E') \xrightarrow{J} (B_2, E)$

Definition 2 Justification: In the view V_i, the set $J(V_i)$ of justified pairs is defined using the following criteria:

- $(B_{\text{genesis}}, 0) \in J(V_i)$;
- If $(B_1, E') \in J(V_i)$ and $(B_1, E') \xrightarrow{J} (B_2, E)$, then $(B_2, E) \in J(V_i)$ as well.

If $(B_2, E) \in J(V_i)$, it means that the checkpoint B_2 is considered as justified in V_i during an epoch E.

Definition 3 (Finalization): In the view V_i, the block (B_0, E) is finalized (particularly k-finalized) if $(B_0, E) = (B_{\text{genesis}}, 0)$ or there is an integer $k \geq 1$ and blocks $(B_1, \ldots, B_k) \in \text{view}(V_i)$ in such a way that the following criteria are met.

- $(B_0, 0), (B_1, 1), \ldots, (B_k, k)$ are consecutive epoch boundary pairs in chain(B_k);
- $(B_0, 0), (B_1, 1), \ldots, (B_{k-1}, k+1)$ belongs to $J(V_i)$.

We say that a block B is finalized for an epoch E if $(B, E) \in F(V_i)$ for some epoch E.

Now we describe the model of Beacon state initialization, justification, and finalization.

B. INITIALIZATION OF BEACON STATE FROM ETH1

Initializing the beacon state from Ethereum 1.0 (Eth1) is a crucial step in the launch of the next evolution Ethereum 2.0. The beacon state represents the current state of the Ethereum 2.0 Beacon Chain which coordinates and manages the overall network. The initialization process ensures that the Beacon Chain begins with a secure and valid state incorporating information from the Eth1 chain, and setting up the necessary structures for the Ethereum 2.0 consensus mechanism and validator system. This transition from Eth1 to Ethereum 2.0 marks a significant milestone in the evolution of the Ethereum network.

The visual representation of the initialization process for the state of Beacon Chain from Eth1 data is provided in the Figure 7. The flow diagram emphasizes the sequential execution of steps and the dependencies.

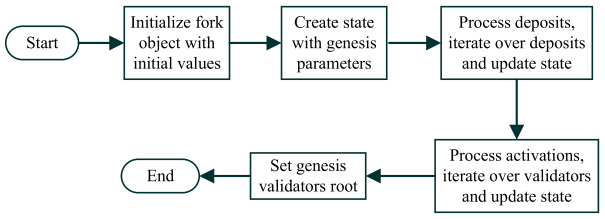

Start
Initialize fork object with initial values
Create state with genesis parameters
Process deposits, iterate over deposits and update state
Process activations, iterate over validators and update state
Set genesis validators root
End

FIGURE 7. Initialization process of beacon state.

The process starts with the initialization of the fork object and the creation of a new beacon state instance. The state is configured with parameters such as genesis time and eth1 data. The function then processes deposits, iteratively updating the state's deposit-related information and invoking the process deposit function. Subsequently, activations are processed by iterating over validators and updating their effective balances. If the effective balance of the validator is maximum then its activation epoch is set as genesis. The process concludes with the setting of validators root as genesis validators root in the state for domain separation and chain versioning.

The purpose of initializing the state is to set up the blockchain securely, consistently, and in a well defined manner. It establishes the foundation for network operation and ensures a smooth transition from the genesis state to the ongoing state of the Beacon Chain. The decisions made during initialization, such as the activation of validators and the determination of effective balances, have a lasting impact on the ongoing security and finality of the blockchain. In summary, the initialization of the state establishes the starting point for the Beacon Chain.

C. JUSTIFICATION AND FINALIZATION

In various contexts, justification and finalization can have different meanings. However, within distributed systems, particularly in the Beacon Chain Eth2 consensus algorithm, the terms are often used as follows:

In the Beacon Chain, justification is the process of providing evidence and valid reasons for accepting a specific block into the chain. It demonstrates the correct and valid addition of the block according to established consensus rules. This integral aspect of the consensus algorithm safeguards the integrity and validity of the entire Beacon Chain. Epoch justification extends this principle to a specific epoch, requiring evidence that attestations and blocks within that time frame comply with Casper FFG rules. A justified checkpoint typically occurs at the start of an epoch. The justification process is explained through the Figure 8. It illustrates that when a node observes a 2/3 supermajority of votes for attestation from the justified checkpoint C_{N+1} to checkpoint C_{N+2}, then C_{N+2} is also marked as justified.

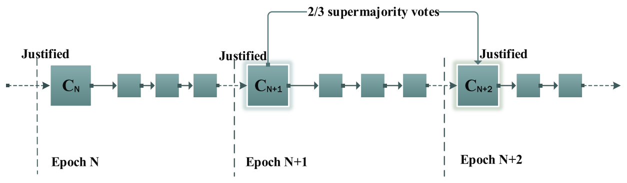


FIGURE 8. An illustration of justification process.

The process of finalization is represented as an important point in the Beacon Chain consensus. It denotes that a block or specific epoch's content becomes irreversible and is permanently added to the chain. It represents a collective agreement within the network, signifying that altering the affirmed state would require a substantial portion of the network's resources. This process improves Beacon Chain security and immutability by establishing an unchangeable transaction history. Epoch checkpoint finalization ensures the irreversibility of blocks and attestations within an epoch. A finalized checkpoint further strengthens this by confirming the status of a justified checkpoint, requiring strong agreement among validators.

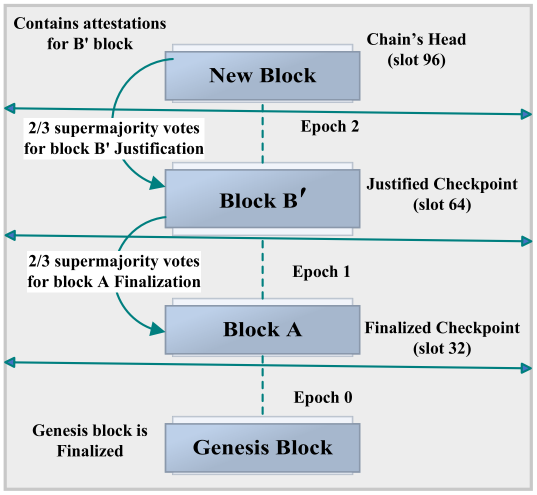

Contains attestations for B' block
Chain's Head (slot 96)
New Block
Epoch 2
2/3 supermajority votes for block B' Justification
Block B′
Justified Checkpoint (slot 64)
2/3 supermajority votes for block A Finalization
Epoch 1
Block A
Finalized Checkpoint (slot 32)
Epoch 0
Genesis block is Finalized
Genesis Block

FIGURE 9. Process of finalization of checkpoints.

In Figure 9, it is clarified that a checkpoint at block A is considered finalized if it is justified and the next block B' is also justified. For this purpose, it is verified that if

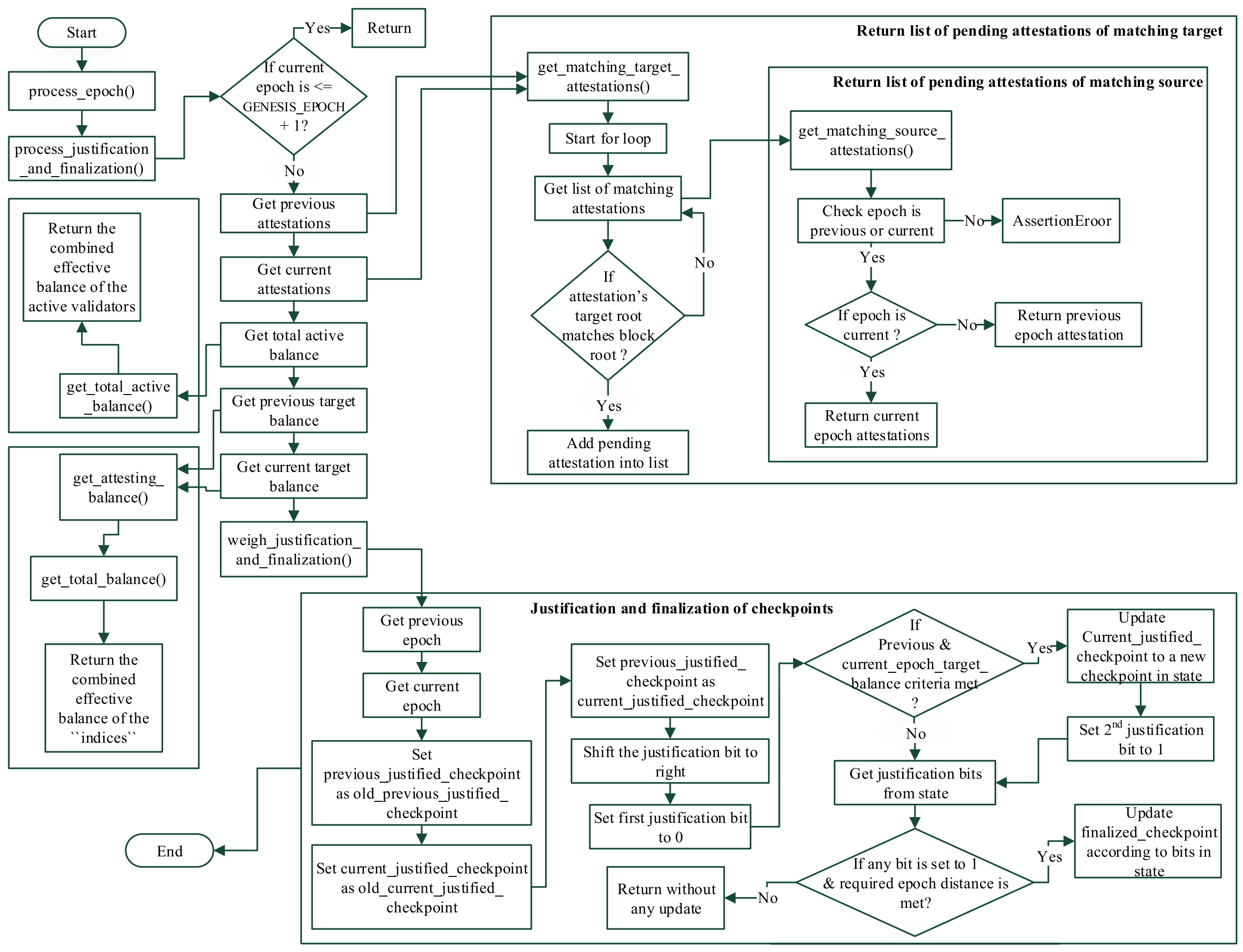

Start
process_epoch()
process_justification_and_finalization()
If current epoch is <= GENESIS_EPOCH + 1?
Yes
Return
No
Get previous attestations
Get current attestations
Get total active balance
Get previous target balance
Get current target balance
weigh_justification_and_finalization()
Return the combined effective balance of the active validators
get_total_active_balance()
get_attesting_balance()
get_total_balance()
Return the combined effective balance of the ``indices``
End
Return list of pending attestations of matching target
get_matching_target_attestations()
Start for loop
Get list of matching attestations
If attestation's target root matches block root ?
No
Yes
Add pending attestation into list
Return list of pending attestations of matching source
get_matching_source_attestations()
Check epoch is previous or current
No
AssertionEroor
Yes
If epoch is current ?
No
Return previous epoch attestation
Yes
Return current epoch attestations
Justification and finalization of checkpoints
Get previous epoch
Get current epoch
Set previous_justified_checkpoint as old_previous_justified_checkpoint
Set current_justified_checkpoint as old_current_justified_checkpoint
Set previous_justified_checkpoint as current_justified_checkpoint
Shift the justification bit to right
Set first justification bit to 0
If Previous & current_epoch_target_balance criteria met ?
Yes
Update Current_justified_checkpoint to a new checkpoint in state
Set 2nd justification bit to 1
No
Get justification bits from state
If any bit is set to 1 & required epoch distance is met?
Yes
Update finalized_checkpoint according to bits in state
No
Return without any update

FIGURE 10. Flow diagram for the process of justification and finalization in the Ethereum 2.0 Beacon Chain.

the block B' of epoch 2 has received attestations from 2/3 of the validators. If this condition is met, the block B' at slot 64, along with the preceding blocks in epoch 1, attains justification. Once the block at slot 64 is justified, all blocks that were previously justified at slot 32, become finalized. The process continues throughout the chain whenever a block gets enough attestations it gets justified and then later finalized. The genesis block is always considered as finalized according to the network's rules. Overall, these processes contribute to the security and stability of the Beacon Chain, and the associated state transitions are considered permanent.

The core components of Beacon Chain specification are its state and the state transition function. Each node within the network upholds a state reflecting its perspective on the current state of the world. The processing of the state is carried out by the Beacon Chain state transition function. When the new beacon block is given for processing, this function updates the older state to a newer state. It includes epoch-transition processing and subsequently handling the blocks' contents. The state transition function of the beacon chain consists of three components. The slot processing is executed for every slot irrespective of concurrent activities. The epoch processing is conducted at every start slot of the epoch regardless of the ongoing tasks. The block processing exclusively takes place during slots when a beacon block has been received.

The epoch processing occurs at the beginning slot of the subsequent epoch during the slot processing. The concept of epochs helps to organize the progression of time and activities within a Beacon chain, making it easier to coordinate and manage consensus processes. The *process_epoch* function is called from the *process_slots* function whenever the state slot is the start slot of each epoch. This is the main function in which all the other functions are called while processing an epoch. The initial step in epoch processing involves justifying and finalizing checkpoints. The justification and finalization ensure that the chosen checkpoints are accepted by the network, promoting a robust and trustworthy consensus mechanism.

The overview of the justification and finalization function is described in Figure 10. The process begins by cautiously checking whether the current epoch is genesis or if the current epoch is within the first two epochs after the genesis, the process exits early to avoid potential issues. After this initial check, the next step is to fetch the attestations for both the previous and current epochs from the state through *get_matching_target_attestations* which retrieves attestations relevant to a specified epoch. It selectively retrieves attestations from the *get_matching_source_attestations*, and it checks whether the specified epoch is the previous or current epoch. Based on this, it returns the corresponding attestations, but only those whose target root matches with the block root of the given epoch. These functions essentially filter out the attestations that are pertinent to the specified epoch, target, and source, providing a more refined set of information for further processing. These attestations serve as critical pieces of information for the subsequent steps. After collecting this historical data, it calculates the total active balance of validators in the system, a sum of all the staked balances of active validators. Then, get the precise balances associated with attestations from the previous and current epochs.

The function passes the gathered data; the total active balance, the balances from the previous and current epochs, and the current state, to the *weigh_justification_and_finalization* function which is responsible for justifying and finalizing checkpoints. It carefully determines the *previous_epoch* and *current_epoch* based on the current state. The previous and current justified checkpoints are stored as old previous and old current justified checkpoints respectively. The justification bits are shifted to compute new justified checkpoints. The target balances are calculated from the previous and current epochs, evaluating whether they met the threshold criteria. If a balance exceeds three times the total active balance, then update the state with a new justified checkpoint with its epoch and root. The corresponding bit in the justification bits is set to 1.

For the process of finalization, the justification bits are examined. If the specified bits are all set as 1 and the required epoch distance is met, then set the finalized checkpoint to the relevant justified checkpoint, otherwise return without any update. This process ensures that the state is appropriately justified and finalized based on the calculated balances, epochs, and justification bits.

D. ALGORITHMS

This section provides a description of algorithms specifically designed for the state initialization and the process of justification and finalization.

The process of initializing the state is described in Algorithm 1. It begins with obtaining information from the Ethereum 1.0 chain, such as the hash of the latest block, the timestamp of that block, and the deposits list (line 1-2). A *Fork* object is created, specifying the initial and current versions of the Ethereum 2.0 protocol. This object helps manage protocol upgrades and changes. A beacon state object is created and initialized with various genesis values (lines 3-4). The deposits made by the validators on the Ethereum 1.0 chain are processed. The deposit data is collected, and a merkle tree structure is created to represent the deposits. The merkle root is stored in the *eth1_data* field of the beacon state and calls the process deposit to further update the state based on the deposit information (lines 6-8). After processing deposits, validators are processed for activation and their effective balances are calculated based on the provided balance and the effective balance increment constant. If a validator's effective balance is equal to the maximum effective balance, they become eligible for activation. Then the activation eligibility epoch and activation epoch are set to the genesis epoch. (lines 10-14). The root of the validators list is calculated by the hash tree root function and the state's genesis validators root is updated (line 15). This root is crucial for domain separation and chain versioning. The initialized beacon state is returned, representing the genesis state of the Ethereum 2.0 beacon chain (line 16).

Algorithm 1 Pseudocode of Initialization of Beacon State.

```
1  InitializeBeaconState($State$, $BlockHash_{eth1}$,
2                        $Timestamp_{eth1}$, $List_{deposits}$)
3    $Fork \leftarrow Fork_{object}$ with initial values
4    $State \leftarrow State_{object}$ with genesis values
5  // Process deposits
6    $Leaves \leftarrow []$
7    for $deposit \in List_{deposits}$ do
8        $leaves$.append($deposit.data$)
         $List_{depositdata} \leftarrow List[DepositData](*leaves)$
         $State.deproot_{eth1data} \leftarrow Root_{hashtree}(List_{depositdata})$
         ProcessDeposit($State, deposit$)
9  // Process activations
10 foreach $i, validator \in State.list_{validators}$ do
11     $List_{balance} \leftarrow State.balances[i]$
12     $Effectivebal_{validator} \leftarrow \min(bal - bal\%EFFECTIVE\_BALANCE\_INCREMENT, MAX_{EFFECTIVE\_BALANCE})$
13     if $Effectivebal_{validator} = MAX_{EFFECTIVE\_BALANCE}$ then
14         $ActivElgEpoch_{validator} \leftarrow E_{genesis}$
15         $ActiveEpoch_{validator} \leftarrow E_{genesis}$
16 $State.root_{genesis\_validators} \leftarrow Root_{hashtree}(State.list_{validators})$
17 return $State$
```

The process of deposit is explained in Algorithm 2, it takes the current state and deposit containing information about deposits in Eth1 as input (line 1). The merkle branch is verified for the deposit data of Eth1 to ensure that it is not possible to fake a deposit (lines 2-3). If the merkle branch is valid, it calls the *ApplyDeposit* function to update the state with the information from the deposit (lines 4-6). If the merkle branch check fails then an *AssertionError* is raised indicating that the entire block is invalid (lines 7-8).

Algorithm 2 Pseudocode of Process Deposit.

```
ProcessDeposit($State$, $deposit$)
  $merkle_{branch}$ ← isValidMerkleBranch($leaf$, $branch$,
                                        $depth$, $index$, $root$)
  if $merkle_{branch}$ = $True$ then
      ApplyDeposit($State$, $key_{public}$, $withdraw_{credentials}$,
                                        $amount$, $sign$)
  else
      raiseError
  endif
```

The *ApplyDeposit* process is defined in the Algorithm 3, it updates the state based on the provided deposit information. It takes five parameters: current state, validator's public key, withdrawal credentials, the amount of the deposit, and the signature (lines 1-2). Firstly, it extracts a list of public keys of all the existing validators in the current state (line 3). Then it is checked that if the provided public key does not belong to the list of existing validators then a *DepositMessage* is created using the provided public key, withdrawal credentials, and amount for depositing validator in the network (lines 4-6). The domain and the signing root are calculated for the deposit message (lines 7-8). Then it is verified that if the deposit message is valid against the validator's signature. If the verification fails, then an assertion error is raised indicating that the deposit is not valid. If the condition is true, then the deposit is considered valid and the validator is added to the registry (lines 9-14). Otherwise, if the public key is already in the list of existing validators, then increase the balance of the corresponding validator by the deposit amount (lines 15-17).

Algorithm 3 Pseudocode of Apply Deposit.

```
ApplyDeposit($State$, $key_{public}$, $withdraw_{credentials}$,
                                    $amount$, $sign$)
  $pubkeyslist_{validators}$ ← $State.validators$
  if $key_{public} \notin pubkeyslist_{validators}$ then
      $msg_{deposit}$ ← DepositMessage($key_{public}$,
                              $withdraw_{credentials}$, $amount$)
      $domain$ ← ComputeDomain(DOMAIN_DEPOSIT)
      $root_{signing}$ ← ComputeSigningRoot($msg_{deposit}$,
  $domain$)
        if $Verify(key_{public}, root_{signing}, sign)$ = $False$ then
          raiseError
        else
          AddValidatorToRegistry($State$, $key_{public}$,
                              $withdraw_{credentials}$, $amount$)
        endif
  else
      IncreaseBalance($state$, $index$, $amount$)
  endif
```

The process of justification and finalization is described in Algorithm 4 (line 1). It is checked that if the epoch is genesis or within the first two epochs after the genesis then it returns (lines 2-3). If the condition is false it means that the epoch is not genesis, then it gets the list of previous and current attestations for the same target from *MatchTarAttest* Algorithm 5 (lines 4-6). The balance of the active validators is calculated from the function *getTotalActiveBalance*, it takes the state as input and returns the total balance (line 7). The *getAttestingBalance* is called here which takes the state and previous attestations as input and returns the total amount of those validators who made these previous target attestations (line 8). Similarly, the total balance is calculated for the current attestations and is stored as the current target balance (line 9). Then *WeiJustAndFin* is called to perform further processing related to justification and finalization (line 10 Algorithm 7).

Algorithm 4 Pseudocode of Justification and Finalization.

```
JustAndFin($State$)
  if ($E_{cur}$ <= $E_{genesis}$ + 1) then
    exit
  else
    $List_{PreAttest}$ ← MatchTarAttest($State$, $E_{pre}$)
    $List_{CurAttest}$ ← MatchTarAttest($State$, $E_{cur}$)
    $Total_{activebal}$ ← $getTotalActiveBalance(state)$
    $Pre_{targetbal}$ ← $getAttestingBalance(state, List_{PreAttest})$
    $Cur_{targetbal}$ ← $getAttestingBalance(state, List_{CurAttest})$
  WeiJustAndFin($State$,$Total_{activebal}$,$Pre_{targetbal}$,$List_{CurAttest}$)
```

The Algorithm 5 named *MatchTarAttest* is designed to get the matching target attestations based on the state and epoch (line 1). To obtain the source attestations list, it calls the function *MatchSourceAttest* with parameters state and epoch. It likely retrieves attestations relevant to the specified epoch (line 2). An empty list named *TargetAttest* is initialized to store the pending attestations of matching target. A loop is initiated that iterates through each attestation *a* in the *SourceAttest* list. Within the loop, there is a conditional statement that checks whether the *target.root* of the attestation matches the block root at the specified epoch. If the condition is true, it means that the target matches, the attestation *a* is added to the *TargetAttest* list (lines 3-6). After the loop completes, the algorithm returns the list of matching target attestations (line 7). This algorithm is designed to selectively retrieve pending attestations based on the matching condition. It aims to ensure that only relevant target attestations are included in the final result.

The Algorithm 6 describes *MatchSourceAttest* process that returns match source attestations based on the given state and epoch (line 1). At first, it checks that the epoch is within the range of previous and current epoch. This is asserted to ensure that the correct epoch is provided (line 2). An empty list called *SourceAttest* is initialized to store the matched source attestations (line 3). If the epoch *E* is equal

Algorithm 5 Pseudocode of Matching Target Attestations.

```
1 MatchTarAttest($State, E$)
2   $SourceAttest \leftarrow$ MatchSourceAttest($State, E$)
3   $TargetAttest \leftarrow []$
4   foreach $a \in SourceAttest$ do
5       if $a.data.target.root ==$ GetBlockRoot($State, E$)
6           $TargetAttest.Add(a)$
7 return $TargetAttest$
```

to current epoch, it adds the current epoch's attestations to the *SourceAttest* list (lines 4-5). Otherwise, it adds the previous epoch's attestations to the *SourceAttest* list (lines 6-7). It returns the *SourceAttest* list, which contains the relevant source attestations based on the provided epoch (line 8).

Algorithm 6 Pseudocode of Matching Source Attestations.

```
1 MatchSourceAttest($State, E$)
2   assert $E \in E_{pre}$ and $E_{cur}$
3   $SourceAttest \leftarrow []$
4   if $E = E_{cur}$ then
5       $SourceAttest.Add \leftarrow State.current\_epoch\_attest$
6   else
7       $SourceAttest.Add \leftarrow State.previous\_epoch\_attest$
8 return $SourceAttest$
```

This Algorithm 7 defines an overview for weighing justification and finalization. It calls the *WeiJustification* Algorithm 8 with parameters state, total active balance, previous and current epoch target balances to obtain a set of justified checkpoints. Then it calls the *WeiFinalization* Algorithm 9 with the outputs of *WeiJustification*, including the state, old previous and current justified checkpoints, and the current epoch as input to obtain finalized checkpoints.

Algorithm 7 Pseudocode of Weigh Justification and Finalization.

```
1 WeiJustAndFin($State, Total_{activebal}, Pre_{targetbal}, Cur_{targetbal}$)
2   $justified_{checkpoints} \leftarrow$ WeiJustification($State, Total_{activebal},$
3                                   $Pre_{targetbal}, Cur_{targetbal}$)
4   $finalized_{checkpoints} \leftarrow$ WeiFinalization($State, OldPJ_{checkpoint},$
5                                   $OldCJ_{checkpoint}, E_{cur}$)
```

The weigh justification process is explained in Algorithm 8 (line 1). It gets the previous and current epoch values and stores in the variables (lines 2-3). It accesses the previous justified checkpoint from the state and store it as an old previous justified checkpoint (line 4). Similarly, it accesses the current justified checkpoint from the state and stores it as the old current justified checkpoint (line 5). It updates the current justified checkpoint as previous justified checkpoint in the state (line 6). The justification bits represents the string of bits that are used for justification. The method used here shifts the bits in the state's justification bits array to the right. There is an array slicing operation that extracts the first *JUSTIFICATION_BITS_LENGTH* - 1 element from the justification bits array, then sets the first bit to 0 (lines 7-9). The process of justification starts with a condition, it checks if the total balance of the validators targeted during the previous epoch is at least 2/3 times of the total active balance. If this condition is true, it implies that a sufficient proportion of the active validators supported some specific target during the previous epoch (line 10). Then it updates the current justified checkpoint in the state with the new checkpoint object having previous epoch and block root. It sets the second bit in the justification bits to 1, this indicates that the previous epoch is justified as current checkpoint (lines 11-12). In the next condition, it again checks if the total balance of the validators targeted during the current epoch is at least 2/3 of the total active balance then the current justified checkpoint is updated in the state. It sets the first bit of the justification bits to 1, which indicates that the current epoch is justified (lines 14-16). This conditional justification scenario is explained through Figure 11.

Algorithm 8 Pseudocode of Weigh Justification.

```
1  WeiJustification($State, Total_{activebal}, Pre_{targetbal}, Cur_{targetbal}$)
2    $E_{pre} \leftarrow$ getPreviousEpoch($State$)
3    $E_{cur} \leftarrow$ getCurrentEpoch($State$)
4    $OldPJ_{checkpoint} \leftarrow State.PJ_{checkpoint}$
5    $OldCJ_{checkpoint} \leftarrow State.CJ_{checkpoint}$
6    $State.PJ_{checkpoint} \leftarrow State.CJ_{checkpoint}$
7    for $i \leftarrow 1$ to $JUSTIFICATION\_BITS\_LENGTH - 1$ do
8        $State.bits_{justification}[i] \leftarrow State.bits_{justification}[i-1]$
9    $State.bits_{justification}[0] \leftarrow 0$
10   if $Pre_{targetbal} * 3 \geq Total_{activebal} * 2$ then
11       $State.CJ_{checkpoint} \leftarrow Checkpoint(E_{pre}, root_{block})$
12       $State.bits_{justification}[1] \leftarrow 1$
13   else if $Cur_{targetbal} * 3 \geq Total_{activebal} * 2$ then
14       $State.CJ_{checkpoint} \leftarrow Checkpoint(E_{cur}, root_{block})$
15       $State.bits_{justification}[0] \leftarrow 1$
16   end if
```

After justification, the process of finalization starts, which is described in Algorithm 9. It takes the state, previous and current justified checkpoints, and epoch as input and updates the finalized checkpoint in the state (line 1). A local variable named *bits* is created and the value of the justification bits is assigned to it (line 2). It represents the status of recently justified checkpoints. A condition checks if all the bits from index 1 to index 3 are set to 1, indicating that the 2nd, 3rd, and 4th most recent epochs are justified, and if the epoch of the old previous justified checkpoint incremented by 3 equals the current epoch then the old previous justified checkpoint in the state is updated as finalized checkpoint (lines 3-4). The next condition checks for justification in the 2nd and 3rd most recent epochs. It also checks if the epoch

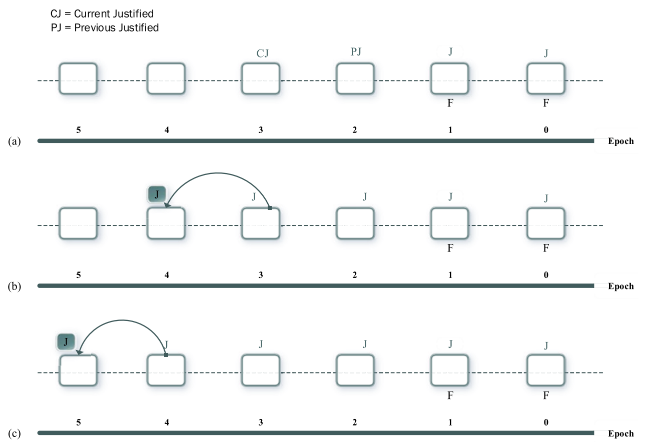


FIGURE 11. The scenario of justification of checkpoints. (a) the checkpoint at epoch 2 is previously justified and 3 is currently justified (b) the epoch 4 checkpoint is justified when justification bits are: 0111 (c) the epoch 5 is justified when justification bits are: 1111.

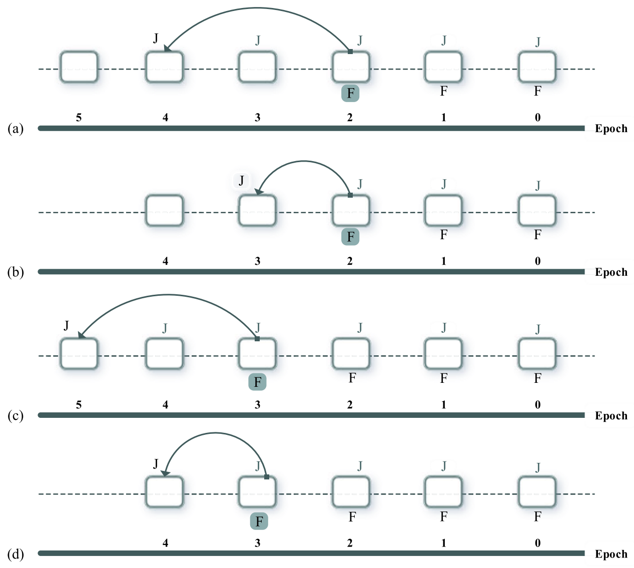


FIGURE 12. Conditional scenario of finalizing checkpoints. (a) the checkpoint at epoch 2 is finalized using the justified checkpoint at epoch 4 (b) the epoch 2 checkpoint is finalized using the justified checkpoint at epoch 3 (c) the epoch 3 checkpoint is finalized using the justified checkpoint at epoch 5 (d) the checkpoint at epoch 3 is finalized using the justified checkpoint at epoch 4.

of the old previous justified checkpoint adding 3 becomes the current epoch then the old previous justified checkpoint is set as finalized (lines 5-6). Similar to the previous conditions, it checks for justification in the 1st, 2nd, and 3rd most recent epochs. If the epoch of the old current justified checkpoint plus 2 becomes the current epoch then the old current justified checkpoint becomes finalized (lines 7-8). Similarly, the following conditional block checks specific patterns of justified epochs and their relative epochs to determine if a finalization condition is met. If a condition is true, the finalized checkpoint in the state is updated accordingly (lines 9-10). The scenario explaining the finalization of the checkpoint is described in Figure 12.

Algorithm 9 Pseudocode of Weigh Finalization.

```
1  WeiFinalization($State$, $OldPJ_{checkpoint}$, $OldCJ_{checkpoint}$, $E_{cur}$)
2    $bits \leftarrow State.bits_{justification}$
3    **if** all($bits[1:4] = 1$) **and** $OldPJ_{checkpoint}.E + 3 = E_{cur}$ **then**
4        $State.finalized_{checkpoint} \leftarrow OldPJ_{checkpoint}$
5    **else if** all($bits[1:3] = 1$) $textbf{and}$ $OldPJ_{checkpoint}.E + 2 = E_{cur}$ **then**
6        $State.finalized_{checkpoint} \leftarrow OldPJ_{checkpoint}$
7    **else if** all($bits[0:3] = 1$) **and** $OldCJ_{checkpoint}.E + 2 = E_{cur}$ **then**
8        $State.finalized_{checkpoint} \leftarrow OldCJ_{checkpoint}$
9    **else if** all($bits[0:2] = 1$) **and** $OldCJ_{checkpoint}.E + 1 = E_{cur}$ **then**
10       $State.finalized_{checkpoint} \leftarrow OldCJ_{checkpoint}$
11   **endif**
```

V. FORMAL SPECIFICATION OF JUSTIFICATION AND FINALIZATION IN BEACON CHAIN

This section describes the formal model of the python reference implementation of the state initialization and justification and finalization process using CSP#. The functions of the reference implementation are described as processes. To ensure the correctness of the specified processes, the properties of the safe initialization, justification and finalization are described using LTL that are verified through the PAT model checker.

A. INITIALIZATION OF BEACON STATE

The formal specification of the beacon state initialization involves creating fork, state initialization, process deposits, process activations, and setting of genesis validators root (line 1).

1. $initialize\_beacon\_state\_from\_eth1() = CreateFork(); StateIni(); ProcessDeposits(); ProcessActivations(); GenesisValRoot();$

The fork creation is described as a process *CreateFork* (line 2). The first and second parameters represent previous version and current version that are set to genesis fork version with null bytes (line 2.1). The epoch is specified as *GENESIS_EPOCH* that is set to zero.

2. $CreateFork() = cfork\{$
   $nullbytes4 = new\ Bytes4(null4);$
   $GENESIS\_FORK\_VERSION = new\ Version(nullbytes4);$
2.1. $fork = new\_Fork(GENESIS\_FORK\_VERSION, GENESIS\_FORK\_VERSION, GENESIS\_EPOCH);$
   $\} \rightarrow Skip;$

TABLE 5. Description of notations.

Notations	Description
$JustAndFin$	Process of Justification and Finalization
$MatchTarAttest$	Get Matching Target Attestations
$MatchSourceAttest$	Get Matching Source Attestations
$WeiJustAndFin$	Weigh Justification and Finalization
$deproot_{eth1data}$	Deposit root of Eth1 data obtained from previous state
$Root_{hashtree}$	Hash tree root function calculates the hash of specific data
E	Epoch
$E_{genesis}$	Genesis Epoch
E_{pre}	Previous Epoch
E_{cur}	Current Epoch
$List_{preAttest}$	List of Previous Attestation
$List_{curAttest}$	List of Current Attestation
$Total_{activebal}$	Total Active balance list of validators
$Pre_{targetbal}$	Previous target balance list
$Cur_{targetbal}$	Current target balance list
$SourceAttest$	List of Matching Source Attestation
$TargetAttest$	This list is obtained through source attestations
$getPreviousEpoch$	It calculates previous epoch based on current epoch
$getCurrentEpoch$	It calculates current epoch according to slot of current state.
$OldPJ_{checkpoint}$	Old previous justified checkpoint
$OldCJ_{checkpoint}$	Old current justified checkpoint
$PJ_{checkpoint}$	Previous justified checkpoint stored in state
$CJ_{checkpoint}$	Current justified checkpoint stored in state
$bits_{justification}$	Justification bits of attestations provided by validators
$finalized_{checkpoint}$	Finalized checkpoint of state
$ActivElgEpoch_{validator}$	Activation Eligibility Epoch of validator
$ActiveEpoch_{validator}$	Activation Epoch of validator
$Effectivebal_{validator}$	Effective Balance of validator

The beacon state initialization is specified as a process *StateIni* that includes five parameters (line 3). The first parameter is *genesis_time*, calculated as the sum of Ethereum 1.0 timestamp and the *GENESIS_DELAY*. The second parameter *Fork* is described earlier. The third parameter *eth1_data* includes information about the Ethereum 1.0 chain and the number of deposits. The fourth parameter describes the header of the latest block on the Beacon Chain. The fifth parameter *randao_mixes* initializes the list of random seeds with the Ethereum 1.0 block hash.

```
3. $StateIni() = sinit\{$
      $genesis\_time = eth1\_timestamp + GENESIS\_DELAY;$
      $eth1\_block\_hash =$
       $new\,Hash32(new\,Bytes32(thirtytwobytes));$
      $eth1\_data =$
       $new\,Eth1Data(eth1\_block\_hash, deposits.GetCount());$
      $latest\_block\_header = new\,BeaconBlockHeader($
       $new\,Root(new\,Bytes32(sszObject.HashTreeRoot($
       $new\,BeaconBlockBody())))));$
      $randao\_mixes = new\,RandaoMixes(eth1\_block\_hash.$
       $Get(), EPOCHS\_PER\_HISTORICAL\_VECTOR);$
3.1. $beaconState = new\,BeaconState(genesis\_time, fork,$
       $eth1\_data, latest\_block\_header, randao\_mixes);$
```

The third part of the beacon state initialization includes the processing of deposits specified as *ProcessDeposits* (line 4). Firstly, the list of deposit data *leaves* is created by extracting the data attribute from each *Deposit* in the deposits list and it is stored in the *leaves* variable (line 4.1). A sub-list of leaves is created by iterating through the deposits list (line 4.2). The *deposit_root* of the *eth1_data* in the state is updated by taking the hash tree root of the *deposit_data_list*.

```
4. $ProcessDeposits() = procdeposits\{$

 $//Process\ deposits$
4.1 $leaves = beaconState.GetLeaves(deposits);$
      $var\ index = 0;$
4.2 $while\ (index < deposits.GetCount())\{$
4.2.1 $deposit\_data\_list.Set(leaves.Get(index), index);$
      $eth1\_data = beaconState.GetEth1Data();$
4.2.2 $eth1\_data.SetDepositRoot(new\,Root(new\,Bytes32($
4.2.3 $sszObject.HashTreeRoot(deposit\_data\_list))));$
```

It is verified if the merkle branch is valid for the deposit data then the eth1 deposit index is incremented in the state, indicating that the deposit has been processed (line 4.3).

```
  $//process\_deposit$
      $leaf = new\,Bytes32(sszObject.HashTreeRoot($
       $deposit.GetDepositData()));$
      $proofs = deposit.GetProofs();$
      $branch = proofs.Get();$
      $branch = new\,SeqBytes32();$
      $beaconState.GetEth1DepositIndex();$
      $root = eth1\_data.GetDepositRoot();$
      $root.Set(new\,Bytes32(rootvalue));$
4.3. $if(beaconState.is\_valid\_merkle\_branch(leaf,$
       $branch, DEPOSIT\_CONTRACT\_TREE\_DEPTH$
       $+1, beaconState.GetEth1DepositIndex(), root) ==$
       $true)\{$
        $beaconState.IncEth1DepositIndex();$
```

The list of public keys of all the existing validators in the current state is then extracted (line 4.4).

```
 $//apply\ deposit$
        $var\ index1 = 0;$
        $statevalidators = beaconState.GetValidators();$
4.4     $while\ (index1 < statevalidators.GetCount())\{$
          $validator = statevalidators.Get(index1);$
          $listpubkeys.Set(validator.GetPubKey(), index1);$
          $index1 = index1 + 1;\ \}$
```

If a public key does not exist in the computed list, then certain verification for the deposit is performed before adding it in the existing validators list (line 4.5). It computes the signing root for the deposit message and the domain (line 4.5.1). After performing verification of the deposit, the validator is added in the validators' registry. If the public key already exists in the computed public key list, then the balance of the corresponding validator is increased (line 4.6).

```
        $var\, data = deposit.GetDepositData();$
        $pubkey = data.GetBLSPubkey();$
        $var\, withdrawal\_credentials =$
        $data.GetWithdrawalCredentials();$
        $var\, amount = data.GetAmount();$
        $var\, signature = data.GetBLSSignature();$

4.5     $if\ (listpubkeys.Contains(pubkey) == false)\{$

          $deposit\_message = new\, DepositMessage(pubkey,$

          $withdrawal\_credentials, amount);$

          $var\, DOMAIN\_DEPOSIT = new\, DomainType($
          $new\, Bytes4(Domain\_Deposit\_Values));$
          $var\, domain = beaconState.compute\_domain($
          $DOMAIN\_DEPOSIT, fork.GetCurVer(),$
          $\_forkData.GetGenValRoot());$

4.5.1     $var\, signing\_root =$
          $beaconState.compute\_signing\_root($
          $new\, Root(new\, Bytes32(sszObject.HashTreeRoot($
          $deposit\_message))), domain);$

        $//\, add\_validator\_to\_registry$
          $//\, get\_validator\_from\_deposit$
          $var\, effective\_balance = helper.Min(amount -$
          $amount\,\%\, EFFECTIVE\_BALANCE\_INCREMENT,$
          $MAX\_EFFECTIVE\_BALANCE);$
          $var\, activation\_eligibility\_epoch =$
          $FAR\_FUTURE\_EPOCH;$
          $var\, activation\_epoch = FAR\_FUTURE\_EPOCH;$
          $var\, exit\_epoch = FAR\_FUTURE\_EPOCH;$
          $var\, withdrawable\_epoch =$
          $FAR\_FUTURE\_EPOCH;$

          $validator = new\, Validator(pubkey, withdrawal\_$
          $credentials, activation\_eligibility\_epoch,$
          $activation\_epoch, exit\_epoch, withdrawable\_$
          $epoch, new\, Gwei(effective\_balance));$

4.5.2     $statevalidators.Add(validator);$
          $balances = beaconState.GetBalances();$
          $balances.Add(new\, Gwei(amount));\ \}$
4.6     $else\,\{$
          $var\, indexp = listpubkeys.GetIndex(pubkey);$

          $beaconState.increase\_balance(beaconState,$
          $indexp, new\, Gwei(amount));$
        $\};\ \}$
      $index = index + 1;$
    $\}\} \rightarrow Skip;$
```

The fourth step in initializing the beacon state is the processing of activations described as *ProcessActivations* (line 5). The effective balance for each validator is calculated and it is ensured that it is within the allowed range (line 5.1.1). If it is equal to *MAX_EFFECTIVE_BALANCE*, then the validator activation eligibility and activation epochs are set to *GENESIS_EPOCH* (line 5.1.2).

```
5. $ProcessActivations() = procactiv\,\{$
     $//Process\ activations$
     $var\, index1 = 0;$
5.1  $while\ (index1 < statevalidators.GetCount())\{$
       $balances = beaconState.GetBalances();$
       $balance = balances.Get(index1);$

       $validator = validators.Get(index1);$
5.1.1  $validator.SetEffectiveBalance(new\, Gwei(helper.Min($
       $balance.Get() - balance.Get()\,\%$
       $EFFECTIVE\_BALANCE\_INCREMENT,$
       $MAX\_EFFECTIVE\_BALANCE)));$
5.1.2  $if\ (validator.GetEffectiveBalance() ==$
         $MAX\_EFFECTIVE\_BALANCE)\{$
         $validator.SetActEligEpoch(GENESIS\_EPOCH);$
         $validator.SetActEpoch(GENESIS\_EPOCH);$
       $\}$
       $index1 = index1 + 1;$
     $\}$
     $\} \rightarrow Skip;$
```

In the end of the beacon state inialization, the process *GenesisValRoot* is specified (line 6). In this process, the *genesis_validators_root* in the state is set by taking hash tree root of the state validators.

```
6. $GenesisValRoot() = \{$
       $beaconState.SetGenesisValidatorsRoot(new\, Root($
       $sszObject.HashTreeRoot(beaconState.$
       $GetValidators())));\ \} \rightarrow Skip;$
```

B. JUSTIFICATION AND FINALIZATION

This section is focused on the formal specification of justification and finalization that is one of the process in processing of epoch (line 6). At this level, we have specified the processes of genesis epoch, justification and finalization and then the next epoch is called.

```
6. $process\_epoch() = Genesis\_epoch();$
     $process\_justification\_and\_finalization();$
     $call\_next\_epoch();$
```

In the process *Genesis_epoch*, the beacon state slot is set as one which means the current epoch is zero.

```
7. $Genesis\_epoch() = \{beaconState.SetSlot(1);\ \} \rightarrow Skip;$
```

The formal specification of justification and finalization is described by the *process_justification_and_finalization*. For the first three epochs, the process *genesis_one_two_justification_and_finalization* process is specified and for the other epochs, the process *weigh_justification_and_finalization* is described.

```
8. $process\_justification\_and\_finalization() =$
      $[beaconState.get\_current\_epoch(beaconState) <=$
      $GENESIS\_EPOCH + 2]$
       $genesis\_one\_two\_justification\_and\_finalization()$ □
      $[beaconState.get\_current\_epoch(beaconState) >$
      $GENESIS\_EPOCH + 2]$
       $weigh\_justification\_and\_finalization(beaconState,$
       $beaconState.get\_total\_active\_balance(beaconState),$
       $beaconState.get\_attesting\_balance(beaconState,$
       $beaconState.get\_matching\_target\_attestations($
       $beaconState, beaconState.get\_previous\_epoch($
       $beaconState))), beaconState.get\_attesting\_balance($
       $beaconState, beaconState.get\_matching\_target\_$
       $attestations(beaconState, beaconState.get\_current\_$
       $epoch(beaconState))));$
```

The process *genesis_one_two_justification_and_finalization* (at line 9) describes the justification of genesis, first, and second epoch, and the finalization of genesis and first epoch (lines 9.1-9.3). The state slot is set to update the current epoch as three. The checkpoint one is set as previous justified checkpoint and the checkpoint two is set as current justified checkpoint.

```
9. $genesis\_one\_two\_justification\_and\_finalization() =$
      $gontjustfin.stateslot\{$
9.1 $genesis = new\ Checkpoint(0);$
      $justified.Add(genesis, -1);$
      $finalized.Add(genesis, -1);$
9.2 $c1 = new\ Checkpoint(1);$
      $justified.Add(c1, 0);$
      $finalized.Add(c1, 0);$
9.3 $c2 = new\ Checkpoint(2);$
      $justified.Add(c1, 1);$
9.4 $beaconState.SetSlot(129);$
9.5 $previous\_justified\_checkpoint = new\ Checkpoint();$
      $previous\_justified\_checkpoint.SetEpoch(1);$
      $beaconState.SetPreviousJustifiedCheckpoint($
       $previous\_justified\_checkpoint);$
9.6 $current\_justified\_checkpoint = new\ Checkpoint();$
      $current\_justified\_checkpoint.SetEpoch(2);$
      $beaconState.SetCurrentJustifiedCheckpoint($
       $current\_justified\_checkpoint);$
      $\} \rightarrow Skip;$
```

The process *weigh_justification_and_finalization* is defined for the epochs greater than two to update the checkpoints (line 10). It takes the state, total active balance, previous epoch target balance, and current epoch target balance as input and justify and finalize the checkpoints accordingly. The previous justified checkpoint is made as current justified checkpoint (line 10.1). The justification bits represent the string of bits (total four) that are used for justification. The justification bits are shift right and zero is pad to the right of bits to justify the current checkpoint (line 10.2).

```
10. $weigh\_justification\_and\_finalization(state, total\_$
      $active\_balance, previous\_epoch\_target\_balance,$
      $current\_epoch\_target\_balance) = wjustfin.stateslot\{$
     $var\ previous\_epoch =$
       $beaconState.get\_previous\_epoch(state);$
     $var\ current\_epoch =$
       $beaconState.get\_current\_epoch(state);$
     $var\ old\_previous\_justified\_checkpoint =$
       $beaconState.GetPreviousJustifiedCheckpoint();$
     $old\_current\_justified\_checkpoint =$
       $beaconState.GetCurrentJustifiedCheckpoint();$
  $//\ Process\ justifications$
  10.1 $beaconState.SetPreviousJustifiedCheckpoint($
      $beaconState.GetCurrentJustifiedCheckpoint());$
  10.2 $beaconState.RShiftJustifBitsPad0(beaconState.$
      $GetJustificationBits(), 1);$
```

If the previous checkpoint is not justified then to justify it, it is checked that if its attestations have total weight more than 2/3 of the total validator stake, then it is set as current justified checkpoint (line 10.3). The justification bit at index 1 is set as one to represent that the previous epoch checkpoint is justified and it is added in the list of justified checkpoints. Here, we have made improvement in the reference implementation by the addition of justified list to record justified checkpoints. It is ensured that if a justified checkpoint already exists in the list then it will not be added in that list. Further, this list is added to verify the justified checkpoints. The same procedure is repeated to justify a current checkpoint (line 10.4). Here the justification bit at index zero is set as one.

```
10.3 $if\ (previous\_epoch\_target\_balance.Get() * 3 >=$
        $total\_active\_balance.Get() * 2)\{$
        $beaconState.SetCurrentJustifiedCheckpoint($
        $new\ Checkpoint(previous\_epoch, beaconState.$
        $get\_block\_root(state, previous\_epoch)));$
        $beaconState.ChangeJustificationBit(beaconState.$
        $GetJustificationBits(), 1, 1);$
        $justified.Add((new\ Checkpoint(previous\_epoch,$
        $beaconState.get\_block\_root(beaconState,$
        $previous\_epoch))), old\_current\_justified\_$
        $checkpoint.GetEpoch());\ \}$
10.4 $if\ (current\_epoch\_target\_balance.Get() * 3 >=$
        $total\_active\_balance.Get() * 2)\{$
        $beaconState.SetCurrentJustifiedCheckpoint(new$
        $Checkpoint(current\_epoch, beaconState.$
        $get\_block\_root(state, current\_epoch)));$
        $beaconState.ChangeJustificationBit(beaconState.$
        $GetJustificationBits(), 0, 1);$
        $justified.Add((new\ Checkpoint(current\_epoch,$
        $beaconState.get\_block\_root(beaconState,$
        $current\_epoch))), old\_current\_justified\_$
        $checkpoint.GetEpoch());\ \}$
```

After justification, the process of finalization is explained. If the justification bits from index one to index three are one, i.e., second, third, and fourth most recent epochs are justified, second most epoch is justified using the fourth epoch as source, and *old_previous_justified_checkpoint* plus three equals the current epoch then the finalized checkpoint in the state is updated as *old_previous_justified_checkpoint*. Here, we have made improvement in the reference implementation by the addition of finalized list to record finalized checkpoints. It is assured that if a finalized checkpoint already exists in the list then it will not be added in that list. Further, this list is added to verify the finalized checkpoints.

```
// Process finalizations
    if(beaconState.CheckBits(beaconState.
      GetJustificationBits(), 1, 3, 111) &&

      (old_previous_justified_checkpoint.GetEpoch() + 3
      == current_epoch)){
      beaconState.SetFinalizedCheckpoint(
      old_previous_justified_checkpoint);
      var FP = finalized.Get(finalized.GetCount() − 1);
      finalized.Add(old_previous_justified_checkpoint,
      FP.GetEpoch()); }
```

If the justification bits from index one to index two are one, i.e., second and third most recent epochs are justified, second most recent epoch is justified using the third epoch as source, and *old_previous_justified_checkpoint* plus two equals the current epoch then the finalized checkpoint in the state is set as *old_previous_justified_checkpoint*.

```
    if(beaconState.CheckBits(beaconState.GetJustification
      Bits(), 1, 2, 11) && (old_previous_justified_
      checkpoint.GetEpoch() + 2 == current_epoch)){
      beaconState.SetFinalizedCheckpoint(
      old_previous_justified_checkpoint);
      var FP = finalized.Get(finalized.GetCount() − 1);
      finalized.Add(old_previous_justified_checkpoint,
      FP.GetEpoch()); }
```

If the justification bits from index zero to index two are one, i.e., first, second, and third most recent epochs are justified, first most recent epoch is justified using the third epoch as source, and *old_current_justified_checkpoint* plus two equals the current epoch then *old_current_justified_checkpoint* is set as the finalized checkpoint in the state.

```
    if(beaconState.CheckBits(beaconState.
      GetJustificationBits(), 0, 3, 111) && (
      old_current_justified_checkpoint.GetEpoch() + 2
      == current_epoch)){
      beaconState.SetFinalizedCheckpoint(
      old_current_justified_checkpoint);
      var FP = finalized.Get(finalized.GetCount() − 1);
      finalized.Add(old_current_justified_checkpoint,
      FP.GetEpoch()); }
```

If the justification bits from index zero to index one are one, i.e., first and second most recent epochs are justified, first most recent epoch is justified using the second epoch as source, and *old_current_justified_checkpoint* plus one equals the current epoch then *old_current_justified_checkpoint* is updated as the finalized checkpoint in the state.

```
    if(beaconState.CheckBits(beaconState.
      GetJustificationBits(), 0, 2, 11) &&
      (old_current_justified_checkpoint.GetEpoch() + 1
      == current_epoch)){
      beaconState.SetFinalizedCheckpoint(
      old_current_justified_checkpoint);
      var FP = finalized.Get(finalized.GetCount() − 1);
      finalized.Add(old_current_justified_checkpoint,

      FP.GetEpoch()); }
    var LFP = finalized.Get(finalized.GetCount() − 1);
    k = LFP.GetEpoch(); } → Skip;
```

The process *call_next_epoch* is defined to execute the next epoch. The procedure explained above is repeated in every epoch. The state slot is adjusted to move to the next epoch.

```
11. call_next_epoch() =
      process_justification_and_finalization(); NextEpoch();
      call_next_epoch();

12. NextEpoch() = [stateslot < 129] nepoch.stateslot {
      beaconState.SetSlot(stateslot + 32);
      stateslot = stateslot + 32;
      } → call_next_epoch();
```

VI. RESULTS AND ANALYSIS

The formal specification of the state initialization and the process of justification and finalization is verified through the PAT model checker. The specification is analyzed to be deadlock-free and safe initialization is ensured. Further, the properties of justification and finalization are specified using LTL that are verified through the PAT model checker.

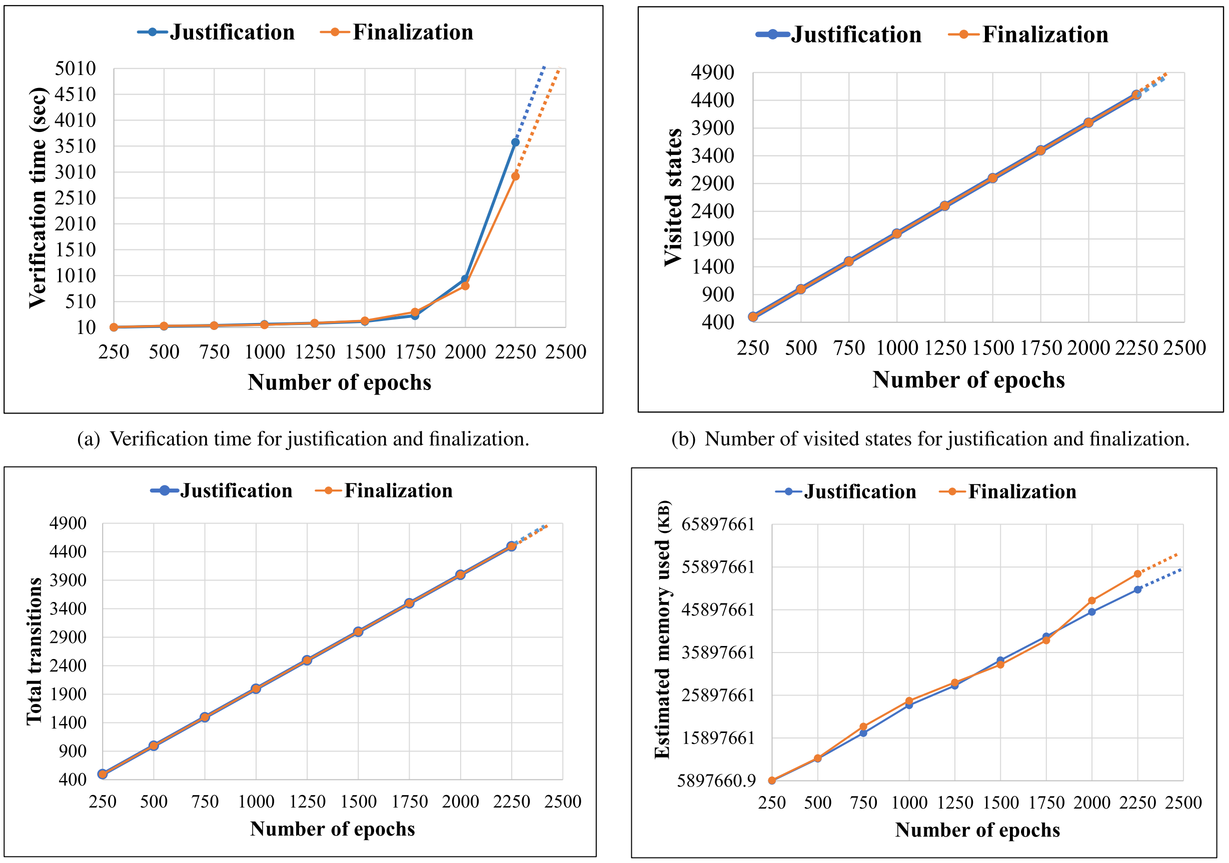
A. DEADLOCK-FREE

A deadlock-free formal model is described as at any moment, no process should wait for another process to proceed. The formal model presented in this work is deadlock-free. The Table 6 shows that the developed formal model is deadlock-free. The assertions $Q1$ and $Q2$ specified below for the state initialization and processing epoch are verified to be deadlock-free.

```
Q1. #assert initialize_beacon_state_from_eth1()
      deadlockfree;
Q2. #assert process_epoch() deadlockfree;
```

B. SAFE INITIALIZATION

In the initialization of beacon state, the safe initialization is ensured when the genesis time becomes equal to *eth1_timestamp* and *GENESIS_DELAY*, the fork is created successfully, the balances and state validators count become



(a) Verification time for justification and finalization.

(b) Number of visited states for justification and finalization.

(c) Total transitions for justification and finalization.

(d) Memory used for justification and finalization.

FIGURE 13. Analysis of number of epochs versus verification time, visited states, total transitions, and memory used for justification and finalization properties.

greater than zero, and the validators are activated. The LTL formula $Q3$ is specified to verify safe initialization. It can be seen in Table 6 that the state initialization is safe.

$\#define\ safeInitialization\ ((beaconState.GetGenesisTime() == eth1\_timestamp + GENESIS\_DELAY)\ \&\&\ (beaconState.GetFork() == fork)\ \&\&\ (balances.GetCount() > 0)\ \&\&\ (statevalidators.GetCount() > 0)\ \&\&\ (statevalidators.areActivated() == true));$

$Q3.\ \#assert\ initialize\_beacon\_state\_from\_eth1() \models \Diamond\ safeInitialization;$

C. JUSTIFICATION

The justification property is divided into four parts. In the first part, it is defined that the genesis epoch checkpoint is justified. In the second part, the previous justified checkpoint is defined to be the previous epoch checkpoint or the checkpoint before the previous epoch. The third part is the super majority link, i.e., if a checkpoint is not justified then to justify it, it is verified that if its attestations have total weight more than $\frac{2}{3}$ of the total validator stake. The fourth part describes that either the previous epoch or the current epoch checkpoint is justified. The LTL formula $Q4$ for justification specifies that eventually always will be the case that the genesis epoch checkpoint is justified and if there is a previous justified checkpoint and there is a super majority link then the current checkpoint is justified. In Table 6, it can be seen that the assertion $Q4$ is verified as true.

$\#define\ GenesisJustified\ justified.GetIndex(genesis) == 0;$

$\#define\ PrevJustifiedCP\ ((justified.GetIndex(old\_current\_justified\_checkpoint) == beaconState.get\_previous\_epoch(beaconState))\ \|\ (justified.GetIndex(old\_current\_justified\_checkpoint) == beaconState.get\_previous\_epoch(beaconState)) - 1);$

$\#define\ SuperMajLink\ ((previousEpochTargetBalance.Get() * 3 >= totalActiveBalance.Get() * 2)\ \|\ (currentEpochTargetBalance.Get() * 3 >= totalActiveBalance.Get() * 2));$

$\#define\ CurJustified\ ((justified.GetIndex(beaconState.GetCurrentJustifiedCheckpoint()) == beaconState.get\_previous\_epoch(beaconState))\ \|\ (justified.GetIndex(beaconState.GetCurrentJustifiedCheckpoint()) == (beaconState.get\_current\_epoch(beaconState))));$

$Q4.\ \#assert\ process\_epoch() \models \Diamond\Box\ (((GenesisJustified)\ \&\&\ (PrevJustifiedCP\ \&\&\ SuperMajLink)) \rightarrow CurJustified);$

TABLE 6. Results for verification of properties against the formal model.

Assertions	Deadlock-free	$\models \Diamond$ safeInitialization	$\models \Diamond\Box$ Justification	$\models \Diamond\Box$ Finalization
#assert initialize_ beacon_ state_ from_ eth1()	Q1: Yes	Q3: Yes	–	–
#assert process_ epoch()	Q2: Yes	–	Q4: Yes	Q5: Yes

Yes : Evaluated true   – : Not analyzed

D. FINALIZATION

The finalization property for checkpoint K consists of three parts. The first part describes that K + 1 checkpoints should be justified. The second part defines that K + 1 justified checkpoints should be adjacent. The third part specifies that the finalized checkpoint list should include K^{th} finalized checkpoint. The finalization property is specified by an LTL formula $Q5$ as eventually always there will be the case that if there are K + 1 justified checkpoints that are also adjacent then K^{th} checkpoint can be finalized. The Table 6, represents that the assertion $Q5$ is verified as true in our formal model.

$$\#define\ Kp1Justified\ (justified.GetCount()\ >=\ k+1);$$
$$\#define\ Kp1JAdjacent\ (justified.Adjacent(k+1));$$
$$\#define\ Kfinalized\ (finalized.Contains(finalized.Get(k))\ ==\ true);$$
$$Q5.\ \#assert\ process\_epoch()\ \models\ \Diamond\Box\ ((Kp1Justified\ \&\&\ Kp1JAdjacent)\ \rightarrow\ Kfinalized);$$

E. ANALYSIS OF THE PROPERTIES

We have carried out the formal verification on an Intel i7-87000 3.20 GHz CPU with 40 GB of memory, running Windows 10 and PAT version 3.5.1. The analysis of the justification and finalization properties is shown in Figure 13 (a–d). The horizontal axis denotes the number of epochs and the vertical axis represents the verification time in seconds, visited states, total transitions, and memory used for the justification and finalization properties. The verification time of both the properties increases drastically when the number of epochs become sufficiently large. This is because the time complexity of the automata-based model checking algorithm for LTL is linear in the size of the transition system and exponential in the length of the formula [32]. We have to verify each property/LTL formula for every epoch. As the time complexity of the model checking increases exponentially with the length of formula, therefore, with the increase of number of epochs, the length of formula also increases, and hence the verification time increases exponentially. Further, the state explosion problem is detected at the epoch 2500 in both the justification and finalization properties. The linear growth of visited states, total transitions, and estimated memory used is observed for both the properties.

VII. CONCLUSION AND FUTURE WORK

The revolutionary advent of blockchain technology has brought a new era of decentralized and transparent systems. Blockchain's immutable nature and distributed ledger capabilities have transformed industries, offering unprecedented security and trust in the digital transactions. Beacon Chain is the backbone of Ethereum 2.0 that introduces a PoS consensus mechanism, addressing scalability concerns associated with other blockchains. This innovation not only improves the sustainability of blockchain networks but also paves the way for enhanced security.

Because of the importance and critical nature of Beacon Chain, it is important to ensure its correctness. Formal methods are the most effective mathematical techniques to ensure the correctness of safety and mission-critical systems. Model checking stands out as the most important and effective formal verification technique. Therefore, it is used in this work, to formally verify the justification and finalization of checkpoints in the Ethereum 2.0 Beacon Chain. The justification and finalization are the significant aspects of Beacon Chain that needs to be verified to increase the reliability of Beacon Chain.

Hence, the processes of state initialization, justification, and finalization are specified using a formal specification language, i.e., CSP#. The properties of safe initialization, justification, and finalization are described through LTL. The formal model and the specified properties are provided as input to the PAT model checker to perform their verification. In the experiments, it is analyzed that the verification time increases drastically as the number of epochs become sufficiently large while linear growth is observed for visited states, total transitions, and estimated memory.

In future, we aim to formally model and verify further processes involved in epoch processing, as well as the entire state transition function in the Beacon Chain. Our research will also involve the formal modeling and verification of the fork choice rule and the behavior of honest validators, contributing to a deeper understanding of the system's resilience and reliability. Additionally, we will consider the formalization of the safety and liveness properties in the Beacon Chain protocol, ensuring both the absence of undesirable states and facilitating the timely progression of the system.

ACKNOWLEDGMENT

The authors would like to thank Sheeza Batool for her contribution to this work including proof-reading of the article.

REFERENCES

[1] J. A. Jaoude and R. G. Saade, "Blockchain applications—Usage in different domains," *IEEE Access*, vol. 7, pp. 45360–45381, 2019.

[2] Consensys. (2020). *The Ethereum 2.0 Beacon Chain Explained*. [Online]. Available: https://consensys.io/blog/the-ethereum-2-0-beacon-chain-explained

[3] V. Buterin, D. Hernandez, T. Kamphefner, K. Pham, Z. Qiao, D. Ryan, J. Sin, Y. Wang, and Y. X. Zhang, "Combining GHOST and Casper," 2020, *arXiv:2003.03052*.

[4] V. Buterin and V. Griffith, "Casper the friendly finality gadget," 2017, *arXiv:1710.09437*.

[5] M. A. Alturki, D. Bogdanas, C. Hathhorn, D. Park, and G. Roşyu, "An executable K model of Ethereum 2.0 beacon chain phase 0 specification," *Project Rep.*, pp. 1–14, 2020.

[6] E. Li, T. Serbanuta, D. Diaconescu, V. Zamfir, and G. Rosu, "Formalizing correct-by-construction Casper in Coq," in *Proc. IEEE Int. Conf. Blockchain Cryptocurrency (ICBC)*, May 2020, pp. 1–3.

[7] M. A. Alturki, E. Li, D. Park, B. Moore, K. Palmskog, L. Pena, and G. Roşu, "Verifying gasper with dynamic validator sets in Coq," Runtime Verification Inc., Chicago, IL, USA, Tech. Rep., 2020, pp. 1–11.

[8] F. Cassez, J. Fuller, and A. Asgaonkar, "Formal verification of the Ethereum 2.0 Beacon chain," in *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Germany: Springer Int. Publishing, 2022, pp. 167–182.

[9] M. Rashid, I. Rasool, N. A. Zafar, and H. Afzaal, "Formal modeling and verification of justification and finalization of checkpoints in Ethereum 2.0 beacon chain," in *Proc. IEEE 1st Karachi Sect. Humanitarian Technol. Conf. (KHI-HTC)*, Jan. 2024, pp. 1–6.

[10] M. Rashid, I. Rasool, N. A. Zafar, and H. Afzaal, "Formal modeling and verification of validator voluntarily exit in Ethereum 2.0 beacon chain," in *Proc. 2nd Int. Conf. Emerg. Trends Electr., Control, Telecommun. Eng. (ETECTE)*, Nov. 2023, pp. 1–6.

[11] Ethereum Foundation. (2020). *Beacon Chain Specifications*. [Online]. Available: https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/beacon-chain.md

[12] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," in *Proc. Decent. Bus. Rev.*, 2008, pp. 1–9.

[13] S. S. Sarmah, "Understanding blockchain technology," *Comput. Sci. Eng.*, vol. 8, no. 2, pp. 23–29, Aug. 2018.

[14] S. Zhang and J.-H. Lee, "Analysis of the main consensus protocols of blockchain," *ICT Exp.*, vol. 6, no. 2, pp. 93–97, Jun. 2020.

[15] V. Buterin, "Ethereum white paper," *GitHub Repository*, vol. 1, pp. 22–23, Jun. 2013.

[16] W. Zhao, "On nxt proof of stake algorithm: A simulation study," *IEEE Trans. Depend. Secure Comput.*, vol. 20, no. 4, pp. 3546–3557, Jul. 2023.

[17] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Proc. Annu. Int. Cryptol. Conf.*, Jul. 2017, pp. 357–388.

[18] S. King and S. Nadal, "PPCoin: Peer-to-peer crypto-currency with proof-of-stake," *Self-Published Paper*, vol. 19, no. 1, pp. 1–6, Aug. 2012.

[19] F. Saleh, "Blockchain without waste: Proof-of-stake," *Rev. Financial Stud.*, vol. 34, no. 3, pp. 1156–1190, Feb. 2021.

[20] S. Aggarwal and N. Kumar, "Hyperledger," in *Advances in computers*, vol. 121. Amsterdam, The Netherlands: Elsevier, 2021, pp. 323–343.

[21] S. Verma, D. Yadav, and G. Chandra, "Introduction of formal methods in blockchain consensus mechanism and its associated protocols," *IEEE Access*, vol. 10, pp. 66611–66624, 2022.

[22] B. Lashkari and P. Musilek, "A comprehensive review of blockchain consensus mechanisms," *IEEE Access*, vol. 9, pp. 43620–43652, 2021.

[23] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, "A survey of distributed consensus protocols for blockchain networks," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 2, pp. 1432–1465, 2nd Quart., 2020.

[24] V. Buterin, "Ethereum: Platform review," in *Opportunities Challenges for Private Consortium Blockchains*, vol. 45. R3's Website, 2016, pp. 1–45.

[25] J. Rosa-Bilbao and J. Boubeta-Puig, "Ethereum blockchain platform," in *Distributed Computing to Blockchain*, R. Pandey, S. Goundar, and S. Fatima, Eds. New York, NY, USA: Academic, 2023, ch. 15, pp. 267–282

[26] U. Mukhopadhyay, A. Skjellum, O. Hambolu, J. Oakley, L. Yu, and R. Brooks, "A brief survey of cryptocurrency systems," in *Proc. 14th Annu. Conf. Privacy, Secur. Trust (PST)*, Dec. 2016, pp. 745–752.

[27] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. 3rd USENIX Symp. Operating Syst. Design Implement.*, 1999, pp. 173–186.

[28] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in Bitcoin," in *Proc. 19th Int. Conf. Financial Cryptogr. Data Secur.*, 2015, pp. 507–527.

[29] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surv. (CSUR)*, vol. 41, no. 4, pp. 1–36, 2009.

[30] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Comput. Surv.*, vol. 28, no. 4, pp. 626–643, Dec. 1996.

[31] J. P. Bowen and M. Hinchey, "Ten commandments of formal methods…ten years on," in *Conquering Complexity*. Cham, Switzerland: Springer, 2012, pp. 237–251.

[32] C. Baier and J.-P. Katoen, *Principles of Model Checking*. Cambridge, MA, USA: MIT Press, 2008.

[33] N. S. of Computing. (2023). *Pat Model Checker Online Documentation*. [Online]. Available: https://www.comp.nus.edu.sg/~pat/OnlineHelp/index.htm

[34] N. D. Team. (2023). *Nusmv User Manual*. [Online]. Available: https://nusmv.fbk.eu/userman/v24/nusmv.pdf

[35] A. Sharma, "End to end verification and validation with SPIN," 2013, *arXiv:1302.4796*.

[36] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *Int. J. Softw. Tools Technol. Transf.*, vol. 1, nos. 1–2, pp. 134–152, Dec. 1997. https://api.semanticscholar.org/CorpusID:648658

[37] C. Gupta and A. Mahajan, "Evaluation of proof-of-work consensus algorithm for blockchain networks," in *Proc. 11th Int. Conf. Comput., Commun. Netw. Technol. (ICCCNT)*, Jul. 2020, pp. 1–7.

[38] J. Garay, A. Kiayias, and N. Leonardos, "The Bitcoin backbone protocol: Analysis and applications," in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn.*, vol. 9057. Cham, Switzerland: Springer, 2015, pp. 281–310.

[39] F. Yang, W. Zhou, Q. Wu, R. Long, N. N. Xiong, and M. Zhou, "Delegated proof of stake with downgrade: A secure and efficient blockchain consensus algorithm with downgrade mechanism," *IEEE Access*, vol. 7, pp. 118541–118555, 2019.

[40] L. Chen, L. Xu, N. Shah, Z. Gao, Y. Lu, and W. Shi, "On security analysis of proof-of-elapsed-time (PoET)," in *Proc. Int. Symp. Stabil., Saf., Security Distrib. Syst. (SSS)*, 2017, pp. 282–297.

[41] M. Milutinovic, W. He, H. Wu, and M. Kanwal, "Proof of luck: An efficient blockchain consensus protocol," in *Proc. 1st Workshop Syst. Softw. Trusted Execution*, Dec. 2016, pp. 1–6.

[42] A. Rahman, "A hybrid pow-pos implementation against 51% attack in cryptocurrency system," Ph.D. dissertation, Dept. Comput. Sci. Eng., United International Univ., Dhaka, Bangladesh, 2019.

[43] C. Cachin, "Architecture of the hyperledger blockchain fabric," in *Proc. Workshop Distrib. Cryptocurrencies Consensus Ledgers*, vol. 310, IL, USA, 2016, pp. 1–4.

[44] Q. Wang, J. Yu, Z. Peng, V. C. Bui, S. Chen, Y. Ding, and Y. Xiang, "Security analysis on dBFT protocol of NEO," in *Financial Cryptography Data Security*. Kota Kinabalu, Malaysia. Cham, Switzerland: Springer, Feb. 2020, pp. 20–31.

[45] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, Dept. Eng. Syst. Comput., Univ. Guelph, Guelph, ON, Canada, 2016.

[46] D. Schwartz, N. Youngs, and A. Britto, "The ripple protocol consensus algorithm," *Ripple Labs Inc White Paper*, vol. 5, no. 8, pp. 151–176, 2014.

[47] H. Afzaal, M. Imran, M. U. Janjua, and S. P. Gochhayat, "Formal modeling and verification of a blockchain-based crowdsourcing consensus protocol," *IEEE Access*, vol. 10, pp. 8163–8183, 2022.

[48] H. Afzaal, M. Imran, and M. U. Janjua, "Formal verification of persistence and liveness in the trust-based blockchain crowdsourcing consensus protocol," *Comput. Commun.*, vol. 192, pp. 384–401, Aug. 2022.

[49] H. Afzaal, M. Imran, and M. U. Janjua, "Formal verification of fraud-resilience in a crowdsourcing consensus protocol," *Comput. Secur.*, vol. 131, Aug. 2023, Art. no. 103290.

[50] I. Fedotov and A. Khritankov, "Statistical model checking of common attack scenarios on blockchain," 2021, *arXiv:2109.02803*.

[51] P. Tholoniat and V. Gramoli, "Formal verification of blockchain Byzantine fault tolerance," in *Handbook on Blockchain*. Cham, Switzerland: Springer, 2022, pp. 389–412.

[52] K. Zheng, Y. Liu, C. Dai, Y. Duan, and X. Huang, "Model checking PBFT consensus mechanism in healthcare blockchain network," in *Proc. 9th Int. Conf. Inf. Technol. Med. Educ. (ITME)*, Oct. 2018, pp. 877–881.

[53] L. Jehl, "Formal verification of hotstuff," in *Proc. Int. Conf. Formal Techn. Distrib. Objects, Compon., Syst.*, vol. 12719. Cham, Switzerland: Springer, 2021, pp. 197–204.

[54] W. Y. M. M. Thin, N. Dong, G. Bai, and J. S. Dong, "Formal analysis of a proof-of-stake blockchain," in *Proc. 23rd Int. Conf. Eng. Complex Comput. Syst. (ICECCS)*, 2018, pp. 197–200.
[55] S. Braithwaite, E. Buchman, I. Konnov, Z. Milosevic, I. Stoilkovska, J. Widder, and A. Zamfir, "Formal specification and model checking of the tendermint blockchain synchronization protocol (short paper)," in *Proc. 2nd Workshop Formal Methods Blockchains*, vol. 84, 2020, pp. 1–10.
[56] D. Park, Y. Zhang, and G. Rosu, "End-to-end formal verification of Ethereum 2.0 deposit smart contract," in *Proc. Int. Conf. Comput. Aided Verification*, Los Angeles, CA, USA. Cham, Switzerland: Springers, 2020, pp. 151–164.
[57] R. Nakamura, T. Jimba, and D. Harz, "Refinement and verification of CBC Casper," in *Proc. Crypto Valley Conf. Blockchain Technol. (CVCBT)*, Jun. 2019, pp. 26–38.
[58] J. Yoo, Y. Jung, D. Shin, M. Bae, and E. Jee, "Formal modeling and verification of a federated Byzantine agreement algorithm for blockchain platforms," in *Proc. IEEE Int. Workshop Blockchain Oriented Softw. Eng. (IWBOSE)*, Feb. 2019, pp. 11–21.
[59] M. A. Alturki, J. Chen, V. Luchangco, B. Moore, K. Palmskog, L. Peña, and G. Roşu, "Towards a verified model of the Algorand consensus protocol in Coq," in *Proc. Formal Methods. FM Int. Workshops*. Porto, Portugal. Cham, Switzerland: Springer, Oct. 2019, pp. 362–367.

HAMRA AFZAAL received the B.S. and M.S. degrees in computer science from COMSATS University Islamabad (CUI), Sahiwal Campus, Pakistan. She is currently pursuing the Ph.D. degree with Information Technology University, Lahore, Pakistan. After graduating, she joined CUI, Sahiwal Campus, as a Lecturer, in 2016. She is a PI of the Project titled "Formally verified Ethereum 2.0 Beacon Chain." Her research interests include blockchain, wireless networks, and formal modeling and verification.

NAZIR AHMAD ZAFAR was born in Pakistan, in 1969. He received the M.Sc. and M.Phil. degrees in mathematics and the M.Sc. degree in nuclear engineering from Quaid-i-Azam University, Pakistan, in 1991, 1993, and 1994, respectively, and the Ph.D. degree in computer science from Kyushu University, Japan, in 2004. He has served at various universities and well-reputed scientific organizations, from 1994 to 2007. He was the Dean/a Professor with the Faculty of Information Technology, University of Central Punjab, Lahore, Pakistan, from 2007 to 2010. He was an Associate Professor with the College of Computer Sciences and Information Technology (CCSIT), King Faisal University (KFU), Al-Ahsa, Saudi Arabia, from 2010 to 2014. In 2014, he returned to Pakistan and joined the COMSATS Institute of Information Technology (CIIT), Sahiwal Campus, as a Professor, and got the opportunity to work as the Head of the Department of Computer Science and the Head of Academics. He was a Professor, the Dean, and the Director of graduate studies at the Khwaja Fareed University of Engineering and Information Technology, Rahim Yar Khan, Pakistan, on leave from CIIT (2017–2018). Since 2021, he has been the Director of COMSATS University Islamabad, Sahiwal Campus, Pakistan. He is the Consultant of the Project: "Formally verified Ethereum 2.0 Beacon Chain." His research interests include the modeling of systems, model checking, formal specification and verification, integration of approaches, safety-critical systems, and blockchain. He is an active member of various academic bodies, selection boards, and societies. He has contributed for various scientific and technical committees, including organizing conferences and curriculum development in the capacity of a member and the Chairperson.

AQSA TEHSEEN received the B.S. degree in computer science from Bahauddin Zakariya University, Multan, Sahiwal, Pakistan, and the M.S. degree in computer science from COMSATS University Islamabad, Sahiwal, in 2021. She is currently a Research Associate dedicated to the Project "Formally Verified Ethereum 2.0 Beacon Chain" with the Information Technology University, Lahore, Pakistan. Her research interests include the Internet of Things, blockchain, formal verification.

SHAHEEN KOUSAR received the B.S. degree in information technology from Bahauddin Zakariya University Multan, Sahiwal, Pakistan, in 2019, and the M.S. degree in computer science from COMSATS University Islamabad, Sahiwal, in 2022. Currently, she is a Research Associate with the Information Technology University, Lahore, Pakistan, focusing on the Project "Formally Verified Ethereum 2.0 Beacon Chain.". Her research interests include the Internet of Things, blockchain, and formal verification.

MUHAMMAD IMRAN (Member, IEEE) received the Ph.D. degree in information technology from University Teknologi PETRONAS, Malaysia, in 2011. He has 15 years of tertiary-level teaching experience in Pakistan, Malaysia, Saudi Arabia, and Australia. He is currently a Senior Lecturer with the School of Science, Engineering and Information Technology, Federation University Australia. Previously, he was with King Saud University (KSU), Saudi Arabia, as an Assistant Professor and an Associate Professor, from 2011 to 2021. He has taught diverse courses, including computer networks, network protocols and algorithms, data science, information security, E-business, data structures, and operating systems. He is also the Founding Leader of the Wireless Networks and Security (WINS) Research Group, KSU, from 2013 to 2021. His research is financially supported by several national and international grants. He has completed several international collaborative research projects with reputable universities. He has published more than 300 research articles in peer-reviewed, highly-reputable international conferences (90), journals (198), editorials (15), book chapter (one), and two edited books. Many of his research articles are among the most highly cited and most downloaded. He has been listed among the top 100 researchers by Thomson Reuters (Web of Science) based on the number of citations earned in the last five years in the computer science and information systems category. His research has been cited more than 9,0000 with an H-index of 47, and an i-10 index of 156 (Google Scholar). His research interests include mobile and wireless networks, the Internet of Things, big data analytics, cloud/edge computing, and information security. He has received several awards and fellowships. He has been involved in about 100 peer-reviewed international conferences and workshops in various capacities, such as the chair, the co-chair, and a technical program committee member. He has been consecutively awarded with the Outstanding Associate Editor of IEEE Access, in 2018 and 2019, among many others. He is the Co-PI of the Project: "Formally Verified Ethereum 2.0 Beacon Chain." He served as an Editor-in-Chief for *European Alliance for Innovation (EAI) Transactions on Pervasive Health and Technology* and an Associate Editor for *IEEE Communications Magazine*. He is serving as an Associate Editor for top-ranked international journals, such as *IEEE Network*, IEEE Future Generation Computer Systems, and IEEE Access. He served/serving as a Guest Editor for about two dozen special issues in journals, such as *IEEE Communications Magazine*, *IEEE Wireless Communications Magazine*, IEEE Future Generation Computer Systems, IEEE Access, and *IEEE Computer Networks*.