Version++: Cryptocurrency Blockchain Handshaking With Software Assurance

Arijet Sarker[+], Simeon Wuthier[+], Jinoh Kim[*], Jonghyun Kim[§], and Sang-Yoon Chang[+]
[+]University of Colorado Colorado Springs
[*]Texas A&M University-Commerce
[§]Electronics and Telecommunications Research Institute

***Abstract*—Cryptocurrency software implements the cryptocurrency operations, including the distributed consensus protocol and the peer-to-peer networking. We design a software assurance scheme for cryptocurrency and advance the cryptocurrency handshaking protocol. Since we focus on Bitcoin (the most popular cryptocurrency) for implementation and integration, we call our scheme Version++, built on and advancing the current Bitcoin handshaking protocol based on the Version message. Our Version++ protocol providing software assurance is distinguishable from the previous research because it is permissionless, distributed, and lightweight to fit its cryptocurrency application. Our scheme is permissionless since it does not require a centralized trusted authority (unlike the remote software attestation techniques from trusted computing); it is distributed since the peer checks the software assurances of its own peer connections; and it is designed for efficiency/lightweight due to the dynamic nature of the peer connections and the large-scale broadcasting in cryptocurrency networking. Utilizing Merkle Tree for the efficiency of the proof verification, we implement and test Version++ on Bitcoin software and conduct experiments in an active Bitcoin node prototype connected to the Bitcoin Mainnet. Our prototype-based performance analyses demonstrate the lightweight design of Version++. The peer-specific verification grows logarithmically with the number of software files in processing time and in storage. In addition, the Version++ verification overhead is small compared to the overall handshaking process; our measured overhead of 2.22% with minimal networking latency between the virtual machines provides an upper bound in the real-world networking with greater handshaking duration, i.e., the relative Version++ overhead in the real world with physically separate machines will be smaller.**

***Index Terms*—Bitcoin, Software Assurance, Permissionless, Distributed, Merkle Tree, Bitcoin Core**

I. Introduction

Bitcoin, a distributed and permissionless cryptocurrency, has gained immense popularity since its inception with the white paper published by Satoshi Nakamoto in 2008 [13] and has a market cap exceeding hundreds of billion dollars (450.06 billion dollars as of August 2022) [5]. The underlying networking of Bitcoin is based on a peer-to-peer network (P2P). To join the Bitcoin network, a potential peer needs to download and install the Bitcoin software and involve in the peer discovery process. The peer starts establishing connections with the discovered peers including TCP handshaking protocol in the OSI network layer and Bitcoin handshaking protocol in the application layer. Once the connection is successfully established, the peer can exchange messages with other peers to participate in the Bitcoin network.

During the Bitcoin handshaking to establish the P2P connections, the two peers identify their current version of the Bitcoin software with each other by exchanging the Version messages. We design and advance such Bitcoin protocol to include the software assurance and call our scheme Version++. Version++, including the Proof generation and verification algorithms, is built on the standard cryptographic primitives and is permissionless, distributed, and lightweight by design to make it compatible with the Bitcoin P2P networking. In the Version++ protocol, Bitcoin software files and the Bitcoin peer's unique ID are used to compute the Proof using Merkle Tree [10], [11] to network it to other Bitcoin peers for software code assurance of the respective software version. Our Merkle Tree-based approach prevents an attacker from reusing the Proof of other Bitcoin peers (due to the use of ID as one of the inputs) and from generating the correct Proof without the necessary code files.

The permissionless and distributed Bitcoin requires the mechanisms for its P2P connection establishment to forgo the reliance on the centralized system, which can provide a challenge in the cryptocurrency networking in general [8], [7]. More specifically, in our work, a centralized architecture for software code assurance violates the permissionless structure in the Bitcoin network. Because of this reason, the remote code attestation based on a remote trusted server is prohibited in permissionless cryptocurrencies including Bitcoin. Moreover, Bitcoin network requires the large-scale broadcasting, which challenges the real-time networking overheads needed for the remote code attestation. Therefore, our work Version++ for achieving software assurance is distinguishable from the remote code attestation in its properties. In contrast to remote code attestation, *software assurance* is permissionless (no need for a centralized trusted execution environment/trusted computing for software code assurance), distributed (Proof generation and verification of code assurance are processed by Bitcoin peers instead of a trusted server) and lightweight (no real-time networking or hardware requirements). However, to achieve these properties, the software assurance achieved by Version++ is only capable of ensuring that a Bitcoin node/prover holds the software code files and has a trade-off with remote code attestation which offers stronger integrity/security assurance including code execution.

Our work is generally applicable and useful for any permissionless and distributed cryptocurrencies in principle. How-

ever, we focus on the Bitcoin cryptocurrency for two reasons. First, Bitcoin is the most popular cryptocurrency. Second, we implement our work on the concrete Bitcoin protocol to advance the Version-based handshake, hence the name of our scheme, Version++.

The rest of the paper is organized as follows. We discuss the background of the default Bitcoin handshaking protocol and the related work in Section II and Section III respectively. We describe our proposed designed scheme, Version++ including the requirements, Proof generation, and verification in Section IV. The implementation and analyses details are illustrated in Section V and Section VI respectively. We conclude the paper with Section VII.

II. Background

We describe the background information on Bitcoin networking and protocol in this section. If two potential peers want to communicate with each other directly they need to go through the Bitcoin handshaking process in the P2P network. New potential peers can start the peer discovery process using DNS query and existing peers can get information about other potential peers from the already connected peers [19]. Since the Bitcoin network architecture is not defined by geographical location the location of the peers are irrelevant and can be selected randomly.

We consider two potential peers - peer A and peer B to describe the Bitcoin handshaking protocol. Peer A establishes a TCP connection with peer B (using port 8333 generally) to start Bitcoin handshaking protocol with peer B [17]. After that peer A sends its own version message to peer B. Though there are 14 data fields in the Version message [3] our interest lies in the version and IP data field of the Version message which defines the highest Bitcoin software version number and the IP address of the transmitting node (i.e., peer A in this case). Peer B also sends its own Version message to peer A upon receiving the Version message from peer A. If peer B accepts the Version message of peer A then it sends a Verack message (an acknowledgment message of the received Version message and has no payload) to peer A. After that, peer B sets the lower of the two versions (comparing the version of peer A and peer B) to be used for further communication with peer A. Similarly peer A also sends the Verack message to peer B after receiving and accepting the Version message from peer B. However, it is worth mentioning that either of the peer (peer A or peer B) should not send a Verack message before receiving the Version message from the other peer. The connection between peer A and peer B is established once they send Verack messages to each other.

III. Related Work in Code Attestation and Merkle Tree

Version++ advances the Bitcoin handshaking protocol and adds the software-assurance functionality, while the current Bitcoin protocol provides no assurance other than the Version message itself identifying the software version. Software assurance is related to remote code attestation using hardware or software-based techniques in trusted computing. We intentionally call the functionality Version++ *assurance* (checking whether the peer has the software version) in order to distinguish from *attestation* (which additionally include real-time code execution). The authors in [16] propose a code attestation approach to attest the piece of code rather than the entire memory using a trustworthy secure kernel. Delegated attestation is proposed in [1] involving embedded devices with low computing power for code attestation. In [15], the authors propose a remote code attestation approach based on a challenge-response protocol using a trusted platform module (TPM). The authors in [6] discuss about obtaining fresh evidence from a running system for code attestation. [1], [9], [14] However, these previous works in remote code attestation require a trusted centralized environment (providing the root of trust and conducting the verification), which goes against the permissionless principle of cryptocurrency. The remote code attestation is also based on a real-time interactive protocol, which adds additional communication transmissions challenging its deployment in the global-scale broadcasting network of cryptocurrency.

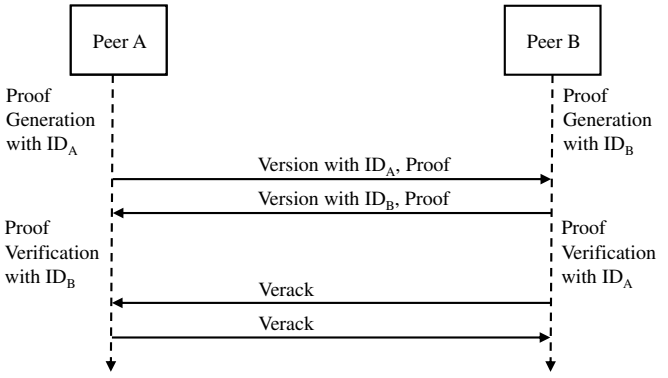

Fig. 1: Version++ handshaking built on the Bitcoin handshaking protocol. The only additions from the current handshaking protocol is the inclusion of the Version++ Proof in the Version message (networking) and the Proof verification on the local peers (computing). The two peers interchange their roles as prover and verifier in the handshaking.

Our scheme in the Version++ handshaking protocol uses the Merkle Tree to compute/generate and verify the Version++ Proof. Such use of the cryptographic hash function combined with Merkle Tree is found in other applications to ensure the integrity of the networking payload and locally stored data, including the block device integrity at kernel level (i.e., dmverity [2]) and transaction integrity on blockchains [18], [13].

IV. Version++ Scheme Protocol

A. Version++ Protocol Overview

Fig. 1 describes the Version++ handshaking protocol which builds on the Bitcoin handshaking based on the Version and Verack exchanges. The Version++ protocol includes the Proof for the software assurance, which is dependent on the peer ID

Version	Services	Timestamp	Receiver services	Receiver address	Receiver port	Transmitter services	Transmitter address	Transmitter port	Nonce	User agent	Start height	Relay	ID	Proof
4 Bytes	8 Bytes	8 Bytes	8 Bytes	16 Bytes	2 Bytes	8 Bytes	16 Bytes	2 Bytes	8 Bytes	Size varies	4 Bytes	1 Byte	8 Bytes	32 Bytes

Fig. 2: Version message data fields and the Version++ addition (highlighted and shaded in blue).

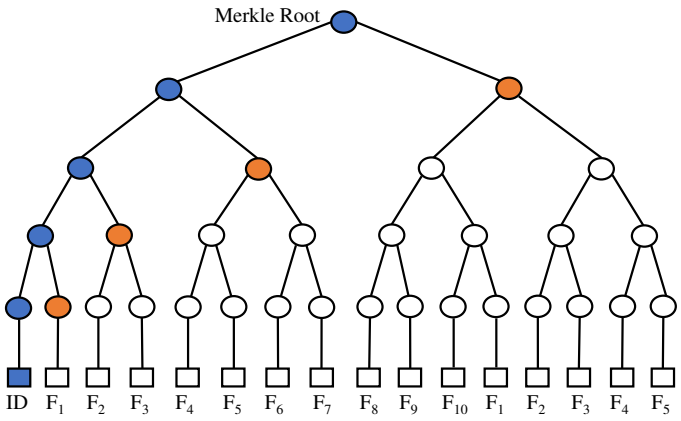

Fig. 3: Merkle Tree

Table I: Definition of each variable

Notation	**Notation Explanation**
n	A node (hash value) in the Merkle Tree
j	Vertical position of n
i	Horizontal position of n in same j
$n_{i,j}$	Position of node n at i,j
h	Vertical position of Merkle Root
H	Hash function
B	Boolean variable
m_r	Received Merkle Root from a Bitcoin potential peer
ID	ID of a Bitcoin potential peer

Algorithm 1 Proof Verification Algorithm

Input: ID, m_r
Output: B ▷ B is Boolean Variable

B = 0
i = 1
j = 1
$n_{1,1} \leftarrow H(\text{ID})$
while $j < h$ **do**
  $n_{i,j+1} \leftarrow H\ (\ n_{i,j} \parallel n_{i+1,j}\ )$
  $j \leftarrow j + 1$
end while
if $n_{i,j+1} == m_r$ **then**
  B = 1
else
  B = 0
end if
Return B

and thus unique to the peer; Fig. 2 shows the information/data fields of the Version message in Version++ protocol (including the data fields of the Version message in the default Bitcoin handshaking protocol) and highlight the Version++ addition of Proof. The two peers exchange the roles of prover and verifier (e.g., peer A acts as the prover when it sends its Proof, and peer B acts as the prover when it sends its Proof) because the Bitcoin handshaking is bidirectional. If the Proof verification preceding the Verack transmission fails, then the verifier has no assurance that the prover indeed has the Bitcoin software version. The rest of this section describes the Version++ protocol in greater details, including the Proof Generation and the Proof Verification, both of which use Merkle Tree.

B. Variables and Notations

In this section, we describe the variables and notations used in the paper. n defines a node (hash value) in the Merkle Tree (shown as a circle in Fig. 3). j defines the vertical position of node n whereas i defines the horizontal position of node n according to Fig. 3. Therefore, the subscripts i, j in $n_{i,j}$ indicate the position of a node in the Merkle Tree. For example, $n_{2,1}$ refers to the node whose horizontal and vertical position is 2 and 1 respectively (denoted by the first orange circle in Fig. 3 according to the bottom-up approach). h is the vertical position of a Merkle Root (the final hash computation of Merkle Tree). H and B denote the hash function and boolean variable respectively. On the other hand, m_r and ID represent the Merkle Root received from another potential peer for verification and ID (e.g., IP address) of a Bitcoin peer respectively.

C. Peers and their roles in Version++

We do not need any additional entity (i.e., a centralized verifier in case of remote code attestation) other than the peers in the Bitcoin network for Version++ handshaking to satisfy the distributed, permissionless structure of bitcoin. The peers generate and verify the Proof by themselves in Version++ handshaking for the software assurance. A peer becomes a *prover* when it needs to prove to other peers that it holds a specific Bitcoin software version. On the other hand, it becomes a *verifier* when it verifies whether its potential peer has the specific Bitcoin software version that the peer is trying to prove. When two potential peers (e.g. peer A and peer B) involve in the peer connection establishment process both the peers become the prover and verifier to

each other during the Version++ handshaking as shown in Fig. 1. A peer needs to generate the Proof using its ID for the Bitcoin software version before proceeding to the Version++ handshaking protocol with another peer. For example, peer A and peer B generate the proof for its Bitcoin software version with $\mathrm{ID_A}$ and $\mathrm{ID_B}$, respectively, before sending the Version message as depicted in Fig. 1. During the Version++ handshaking, both the potential peers exchange the Version message with its own ID and Proof for the Bitcoin software version it holds. After that, the peers check the version data field from the received Version message and regenerate the received Proof for that version with the receive ID for Proof verification. As shown in Fig. 1, peer A and peer B use the received ID ($\mathrm{ID_B}$ and $\mathrm{ID_A}$, respectively) from the Version message for Proof verification. If the verification is successful then the verifier sends the Verack message to the prover. Version++ handshaking succeeds when both the peers involved in the handshaking has successful Proof verification and sent the Verack message to each other.

D. Prerequisites and Approaches

This section describes the prerequisites and the approaches to fulfill them. In our scheme, a peer can generate and provide the Proof of a Bitcoin software version only if it has that software version. An attacker is defined as a peer who generates or provides the Proof without having the specific software version. We build our scheme on the cryptographic primitives, i.e., one-way and the collision-resistance properties of the cryptographic hash function. The cryptographic hash functions' weak collision-resistance property (also known as the secondary preimage resistance) enforces that the prover uses the correct/unmodified software code files and its ID. It specifically disables the following two threats by an attacker: first, an attacker who does not have the software codes or has the modified software codes cannot generate the Proof of the current software codes; second, an attacker cannot manipulate the ID to generate the Proof so that the ID changes while retaining the same Proof.

We employ the following two methods to provide security that prevents an attacker from reusing the Proof of other Bitcoin peers (for example, after eavesdropping). First, peer ID (e.g., IP address) is included as the input for Proof generation process to make the Proof unique to each Bitcoin peer and ID-dependent. Second, the Proof (Merkle Root) is only networked and all the other intermediate branch values of the Merkle Tree are processed and stayed in the local machine (similarly to a private key in public-key cryptography). To make the Proof consistent and the same across Bitcoin peers (which is required because the Proof generation and verification happen on separate Bitcoin peers), we establish a rule of ordering the files (as input to our scheme) according to the file size.

E. Proof Generation

In Version++, a prover uses code files of a Bitcoin software version and its ID (IP address) as input for Proof generation. The *Proof generation* involves creating a Merkle Tree using the code files and ID as shown in Fig. 3. The code files and hash values are represented as rectangles and circles respectively in Fig. 3. The Merkle Root generated from the Merkle tree is communicated to the verifier as the Proof, and other hash nodes/outputs within the Merkle tree remain on the local machine of the prover.

We denote F to be an ordered collection of the code files while appending the ID in the first element, and $\hat{\mathrm{F}}$ the cyclic extension of F. $\hat{\mathrm{F}}$ is required to make a balanced tree from F having the number of leaves to be a power of two, thus $|\hat{\mathrm{F}}| = 2^{\lceil \log_2 |\mathrm{F}| \rceil}$. For example, the bottom row in Fig. 3 shows the $\hat{\mathrm{F}}$ elements when there are ten code files and $|\mathrm{F}| = 11$. Therefore, we round this F to the next power of two using the previous files in cyclic order so that $|\hat{\mathrm{F}}| = 16$. The prover generates the complete Merkle Tree including the Merkle Root from $\hat{\mathrm{F}}$, where the computing progression moves upward in Fig. 3 using the one-way hash functions.

F. Proof Verification

Handshake w/ verification involves sending Version message between the peers, Proof verification and responding back with Verack message given that Proof verification is successful and does not include Proof generation. In this section, we describe about the Proof verification in-details. The verifier checks the Bitcoin software version number received from the Version message of the prover and recomputes the Proof for that version using the prover's ID. The locally stored nodes/hash outputs that are affected by the ID (marked as orange circular in Fig. 3) are known to the verifier if it has already generated the Merkle Tree and Merkle Root from the code files of that specific version. The verifier only needs to store these nodes (*Stored Hashes*) of a Bitcoin software version for Proof verification. However, the verifier regenerates the Merkle Root using the prover's ID with those Stored Hashes and only needs to update the nodes/hash outputs that are affected by the ID. Therefore *Proof verification* involves regenerating the Proof using prover's ID and comparing the regenerated Proof with the Proof received from the prover. The details description of the Proof verification algorithm are shown in Algorithm 1 which takes ID and m_r as input and outputs B. It computes H(ID) and puts it in $n_{1,1}$ position of the Merkle Tree. It keeps concatenating the hash value in $n_{i,j}$ and $n_{i+1,j}$ position, hashing the concatenated value and putting the hashed value in the $n_{i,j+1}$ position till j reaches to the height of the Merkle Root, h. If the computed Merkle Root in final $n_{i,j+1}$ position matches with m_r then Algorithm 1 returns B = 1 (successful verification) otherwise B = 0 (unsuccessful verification).

V. Implementation

We take a hybrid approach for our implementation and experimentation. For the Version++ prototype, we implement a prover and a verifier in two virtual machines in one physical machine. This experimental setup involves the minimum networking latency (the handshake protocol time duration will be greater in the real-world networking scenarios) and thus

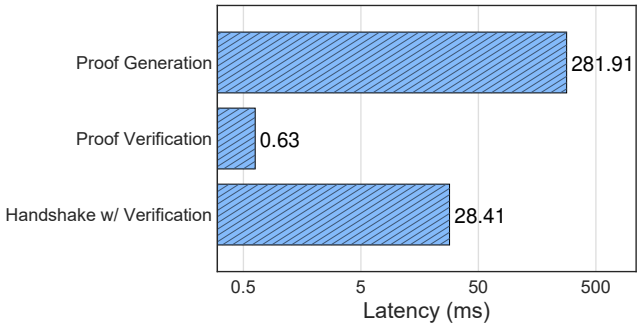

Fig. 4: The latency measurements for the different operations of Version++

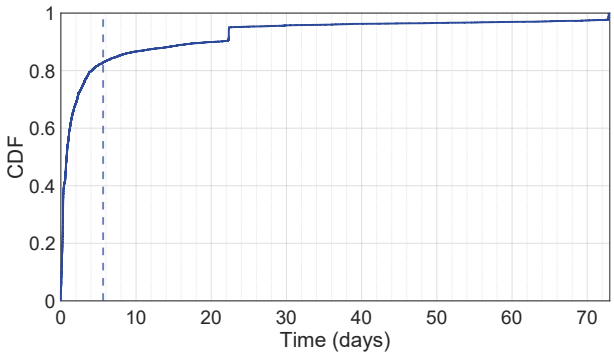

Fig. 5: The distribution/CDF of duration that peers stay connected using real-world Bitcoin data.

provide the worst-case overhead analyses for our paper (the real-world overhead for Version++ will be even smaller than our measurements when the networking latency increases the handshaking duration). We also connect a Bitcoin prototype to the Mainnet to measure the real-world P2P networking, including the peer connection duration and the current software version distributions providing us with references for the Version++ overheads.

The implementation inside of Bitcoin software is embedded within the connection manager (CONNMAN), utilizing Bitcoin's native SHA-256 hash generator and widespread implementation of a Merkle Tree [12]. Our software creates an instance of the Proof generation, Proof verification and Merkle Tree using C++, built-in *recursive_directory_iterator* and a regular expression to retrieve all relevant files in the *src* folder associated with C++, C, and shell[1]. For Version++ prototype, we use a network isolated from the internet and set two machines to run our Bitcoin Core instance. One machine uses Python 3.9 to repeatedly connect and disconnect from the other peer, with a 500ms delay between samples. Both machines have 16 GB DDR4 RAM with the AMD Ryzen Threadripper 3960X processor. We use *tcpdump* to capture the network traffic of Bitcoin, and extract the handshake information for each sample. Though we implement Version++ using Bitcoin

[1]For consistency of the Proof inputs across the peer nodes, we exclude the files that are dynamic across Bitcoin instances, e.g. configuration files and those depending on configurations.

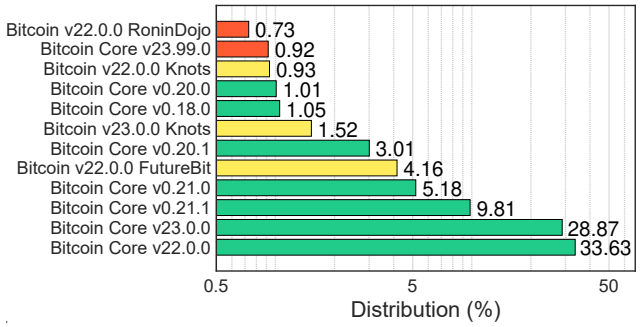

Fig. 6: The distribution (%) of Bitcoin software versions among the nodes (top 90%) in Bitcoin network. The horizontal axis is in logarithmic scale. Green, yellow and red color indicate the Bitcoin Core software versions, Bitcoin Core alternatives versions, and Bitcoin Core exceptions versions, respectively.

Core software other Bitcoin softwares (fork from the Bitcoin Core software) can also be used for the implementation given that they have the code files.

VI. ANALYSES

In this section, we provide latency analysis for Proof generation, verification and handshake with verification, comparison of Version++ handshaking duration with the average peer connection duration and storage overhead for Proof verification to demonstrate the overhead of implementing Version++ in the default Bitcoin handshaking protocol.

Latency We show the latency in Fig. 4 for Proof generation, Proof verification and handshake with verification which are 281.91 ms, 0.63 ms and 28.41 ms respectively. We also observe that Proof verification latency is $\frac{0.63}{28.41} = 2.22\%$ of handshake with verification latency. Since the implementation of Version++ is done in one virtual machine this overhead will be much smaller in the real-world considering the networking latency between two peers which are isolated by different geographical locations and machines.

Update Frequency We calculate the average Bitcoin Core software version update frequency considering the release date of the first version (Bitcoin Core v0.10.0), the latest version (Bitcoin Core v23.0.0), and the number of Bitcoin Core versions updated during this period. The Bitcoin Core software update frequency on average is $\frac{1650911904-1231526304}{38}$ = $\frac{419385600}{38}$ = 11036463.1579 seconds = 127.7368421 days or 127 days 17 hours 41 minutes 3.1579 seconds. Therefore, a peer needs to go through the Proof generation once taking 281.91 ms every 127.7368421 days on average.

Time Overhead We show the peer connection duration distribution in Fig. 5 by extracting the public node information from Bitnodes [4]. The average peer connection in the Bitcoin networking lasts for approximately 5.62 days (which is 485568000 millisecond) before it gets disconnected. Therefore, the verification overhead compared to the peer-connection duration only takes $\frac{310.32}{485568000} = 6.39 \times 10^{-7}$ of the average peer connection duration. The numerator values

Table II: The storage overhead of the top 90% distribution of Bitcoin software versions

Bitcoin Software	**Storage Overhead (MB)**
Bitcore Core	1.29
Bitcore Core alternatives	0.55
Bitcore Core exceptions	N/A

are in Fig. 4 while the denominator is from Fig. 5. Therefore, the Version++ handshaking overhead is significantly small compared to the peer connection duration.

Storage Overhead We show the distribution of top 90% of Bitcoin software versions used by the peers in the Bitcoin network in Fig. 6. The green color indicates the Bitcoin Core Software versions (*Bitcoin Core*), the yellow color indicates the software versions which create a fork from the default Bitcoin Core (*Bitcoin Core alternatives*) and the red color indicates the software versions (*Bitcoin Core exceptions*) which do not have the publicly available executables/code files for the implementation of Merkle Tree (i.e., Bitcoin v22.0.0 RoninDojo) or is still making updates (i.e., Bitcoin Core v23.99.0). We show the storage overhead for Proof verification of Bitcoin software versions (among the top 90% distribution) in Table II for Bitcoin Core, Bitcoin Core alternatives and Bitcoin Core exceptions. We do not calculate the storage overhead for Bitcoin Core exceptions since those versions do not have the publicly available executables/code files or still updating its files. Among the distribution of the top 90% of Bitcoin softwares, Bitcoin Core, Bitcoin Core alternatives and Bitcoin Core exceptions have 7 versions, 3 versions and 2 versions respectively. Bitcore Core softwares are used by approximately 83% peers in the Bitcoin network and storage overhead is 1.29 MB (Megabytes). Therefore, a Bitcoin node can verify approximately 83% of Bitcoin nodes by storing 1.29 MB of Merkle Tree. Bitcore Core alternatives is used by 6.61% peers and storage overhead is 0.55 MB. Therefore, a verifier can verify 89.61% of peers in the Bitcoin network by having a storage overhead of only 1.849 MB.

VII. Conclusion

Our proposed scheme, Version++ provides a distributed, permissionless and lightweight solution for software assurance in a Bitcoin P2P network. Version++ differs from the remote code attestation in terms of centralized execution, centralized verification, and extensive networking/hardware requirements to make it compatible with the distributed permissionless architecture of Bitcoin network. The analyses result for Version++ demonstrates that it can be integrated with the default Bitcoin handshaking protocol with very little overhead. For example, Version++ handshaking overhead is 6.39×10^{-7} times smaller than the average peer-connection duration.

Acknowledgement

This work was supported by National Science Foundation under Grant No. 1922410 and by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2021-0-02107, Collaborative research on element Technologies for 6G Security-by-Design and standardization-based International cooperation). Our software implementation of the Version++ protocol is available at https://github.com/bitcoin-version-plus-plus/bitcoin-version-plus-plus.

References

[1] M. Ammar, B. Crispo, I. De Oliveira Nunes, and G. Tsudik, "Delegated attestation: scalable remote attestation of commodity cps by blending proofs of execution with software attestation," in *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2021, pp. 37–47.

[2] M. Baines and W. Drewry, "Integrity-checked block devices with device mapper," in *Linux Security Symposium*, 2011.

[3] Bitcoindeveloper, "P2P Network," https://developer.bitcoin.org/reference/p2p_networking.html#version, 2022, [Online; accessed 20-August-2022].

[4] Bitnodes, "Global bitcoin nodes distribution," https://bitnodes.io/dashboard/, 2022, [Online; accessed 1-August-2022].

[5] CoinMarketCap, "trending cryptocurrencies," https://coinmarketcap.com/trending-cryptocurrencies/, 2022, [Online; accessed 20-August-2022].

[6] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," *International Journal of Information Security*, vol. 10, no. 2, pp. 63–81, 2011.

[7] W. Fan, H.-J. Hong, J. Kim, S. J. Wuthier, M. Nakashima, X. Zhou, E. Chow, and S.-Y. Chang, "Lightweight and identifier-oblivious engine for cryptocurrency networking anomaly detection," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[8] W. Fan, H.-J. Hong, S. Wuthier, X. Zhou, Y. Bai, and S.-Y. Chang, "Security analyses of misbehavior tracking in bitcoin network," in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2021, pp. 1–3.

[9] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth, "New results for timing-based attestation," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 239–253.

[10] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Conference on the theory and application of cryptographic techniques*. Springer, 1987, pp. 369–378.

[11] ——, "Protocols for public key cryptosystems," in *Secure communications and asymmetric cryptosystems*. Routledge, 2019, pp. 73–104.

[12] Microsoft, "Microsoft/merklecpp: A C++ library for creation and manipulation of Merkle Trees," https://github.com/microsoft/merklecpp, 2021, [Online; accessed 20-July-2022].

[13] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.

[14] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "{APEX}: A verified architecture for proofs of execution on remote devices under full software compromise," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 771–788.

[15] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn, "Design and implementation of a tcg-based integrity measurement architecture." in *USENIX Security symposium*, vol. 13, no. 2004, 2004, pp. 223–238.

[16] E. Shi, A. Perrig, and L. Van Doorn, "Bind: A fine-grained attestation service for secure distributed systems," in *2005 IEEE Symposium on Security and Privacy (S&P'05)*. IEEE, 2005, pp. 154–168.

[17] J. Tapsell, R. N. Akram, and K. Markantonakis, "An evaluation of the security of the bitcoin peer-to-peer network," in *2018 IEEE international conference on internet of things (IThings) and IEEE green computing and communications (GreenCom) and IEEE cyber, physical and social computing (CPSCom) and IEEE smart data (SmartData)*. IEEE, 2018, pp. 1057–1062.

[18] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[19] Y. Zhang, R. Tan, X. Kong, Q. Tan, and X. Liu, "Bitcoin node discovery: Large-scale empirical evaluation of network churn," in *International Conference on Artificial Intelligence and Security*. Springer, 2019, pp. 385–395.