

Article

Blockchain Handshaking with Software Assurance: Version++ Protocol for Bitcoin Cryptocurrency [†]

Arijet Sarker ^{1,*} , Simeon Wuthier ² , Jinoh Kim ³ , Jonghyun Kim ⁴ and Sang-Yoon Chang ^{2,*} ¹ Computer Science Department, Florida Polytechnic University, Lakeland, FL 33805, USA² Department of Computer Science, University of Colorado Colorado Springs, Colorado Springs, CO 80918, USA; swuthier@uccs.edu³ Computer Science and Information Systems, Texas A&M University–Commerce, Commerce, TX 75428, USA; jinoh.kim@tamuc.edu⁴ Electronics and Telecommunications Research Institute, Daejeon 34129, Republic of Korea; jhk@etri.re.kr

* Correspondence: asarker@floridapoly.edu (A.S.); schang2@uccs.edu (S.-Y.C.)

[†] This paper is an extended version of our paper published in Sarker, A.; Wuthier, S.; Kim, J.; Kim, J.; Chang, S.Y. Version++: Cryptocurrency blockchain handshaking with software assurance. In Proceedings of the 2023 IEEE 20th Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, USA, 8–11 January 2023; pp. 804–809; and Sarker, A.; Wuthier, S.; Kim, J.; Kim, J.; Chang, S.Y., Version++ Protocol Demonstration for Cryptocurrency Blockchain Handshaking with Software Assurance. In Proceedings of the 2023 IEEE 20th Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, USA, 8–11 January 2023; pp. 915–916.

Abstract: Cryptocurrency software implements cryptocurrency operations (including the distributed consensus protocol and peer-to-peer networking) and often involves the open-source community. We design a software assurance scheme for cryptocurrency and advance the cryptocurrency handshaking protocol by providing the verification capability of the Bitcoin software by peers and preventing any potential peer from establishing a connection with modified Bitcoin software. 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 previous research because it is permissionless, distributed, and lightweight for 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 to support the dynamic nature of the peer connections and 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. Furthermore, the Version++ verification overhead is small compared to the version-verack handshaking process; we measure the overhead to be 0.524% in our local networking environment between virtual machines and between 0.057% and 0.282% (depending on the peer location) in our more realistic cloud-based experiments with remote peer machines.

Keywords: Bitcoin; software assurance; permissionless; distributed; Merkle Tree; Bitcoin Core

Citation: Sarker, A.; Wuthier, S.; Kim, J.; Kim, J.; Chang, S.-Y. Blockchain Handshaking with Software Assurance: Version++ Protocol for Bitcoin Cryptocurrency. *Electronics* **2024**, *13*, 3857. <https://doi.org/10.3390/electronics13193857>

Academic Editor: Domenico Ursino

Received: 19 August 2024

Revised: 19 September 2024

Accepted: 27 September 2024

Published: 29 September 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Bitcoin, a distributed and permissionless cryptocurrency, has gained immense popularity since its inception with the white paper published by Satoshi Nakamoto in 2008 [1] and has a market cap exceeding hundreds of billions of USD (USD 450.06 as of August 2022) [2]. 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

get involved in the peer discovery process. The peer starts establishing connections with the discovered peers, including the 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 a 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 [3,4] 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 cryptocurrency networking in general [5,6]. 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 large-scale broadcasting, which challenges the real-time networking overheads needed for the remote code attestation. Therefore, our 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 tradeoff 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. However, 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 related work and the background of the default Bitcoin handshaking protocol in Section 2 and Section 3, respectively. We describe our proposed designed scheme, Version++, including the requirements, Proof generation, and verification in Section 4. The implementation and analysis details are illustrated in Section 5 and Section 6, respectively. We conclude the paper with Section 7.

2. 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 includes real-time code execution).

Previous research studied remote code attestation in software, hardware, and hybrid. The software-based code attestation technique does not rely on any dedicated hardware [7] and is suitable for resource-constrained embedded devices, e.g., wireless sensor nodes, IoT

devices [8]. However, Bitcoin software installation typically requires computing devices with higher configurations than those embedded devices in terms of memory, storage, processing power, etc. This attestation technique involves real-time challenge-and-response protocol, tight time synchronization, e.g., refs. [9–12] and filling the empty memory space of the prover's computing device to prevent the attacker from injecting malicious code during attestation [13,14].

Hardware-based code attestation involves hardware security, e.g., trusted platform module (TPM) [15], software guard extensions (SGX) [16], ARM TrustZone [17]. These provide greater security than software-based code attestation because of the ability to perform tamper-resistant cryptographic operations (e.g., protecting cryptographic keys, signing attestation reports) and implement a secure execution environment by the dedicated hardware and hardware-based root-of-trust. These attestation techniques involve a timing-based challenge-response protocol using TPM [18,19], fine-grained attestation instead of coarse-grained attestation (verification of the specific components instead of the whole system) [20,21], SGX-based isolated execution environment for verification [22,23], and verification of kernel [24] or domain integrity (checking expected behavior by predefined rules and constraints) for frequently changing/dynamic systems [25].

More recent literature takes a hybrid approach to incorporate both the aforementioned software- and hardware-based code attestation approaches [26–31] while reducing the hardware costs using, e.g., Field-Programmable Gate Array (FPGA), Memory Protection Unit (MPU), Micro-Controller Unit (MCU), or Read-Only Memory (ROM) [29,32–34].

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 decentralized and permissionless principles of cryptocurrency. Remote code attestation requires more computational overload than our scheme since it is based on a real-time interactive protocol, which adds additional communication transmissions, challenging its deployment in the global-scale broadcasting network of cryptocurrency. Moreover, the requirement to know the exact configuration of the prover's computing device and check the entire/partial memory space during attestation [35–37] make this technique infeasible to implement in the Bitcoin P2P network.

Also related to our research are those using Merkle Tree. Our Version++ handshaking protocol uses the Merkle Tree to compute/generate and verify the Version++ Proof. While the cryptographic hash function provides the integrity of the networking payload and stored data, the Merkle Tree structure built on the hash function computations provide verification efficiency. Similarly to our approach, the use of Merkle Tree and hash function can be found in other applications, including the block device integrity at the OS kernel level, e.g., dm-verity [38], mobile/wireless networking software assurance [39], and transaction integrity on cryptocurrency blockchains (Merkle tree is used for the data efficiency for ledger storage and networking) [1]. Our work differs from aspect-oriented programming (AOP) and aspect-oriented software development (AOSD) because those works are related to improving the security of a single version of software systems (e.g., consensus mechanisms, smart contracts, and transaction validation of a blockchain software system) [40], software development life cycles [41], and cloud services [42]. On the contrary, our work provides software assurance for multiple software versions of Bitcoin cryptocurrency software.

3. Background: Bitcoin Version Handshake Protocol

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 obtain information about other potential peers from those already connected peers [43].

We consider two peers (peer A and peer B) to describe the Bitcoin handshaking protocol. Peer A establishes a TCP connection with peer B (using port 8333) to start the Bitcoin handshaking protocol with peer B [44]. Afterward, peer A sends its own Version message to peer B. Though there are 13 data fields in the Version message [45], our interest

lies in the version and IP data field of the Version message, which defines the most recent Bitcoin software version number and the IP address of the transmitting node (i.e., the IP address of peer A in this case), respectively. Peer A also sends *Wtxidrelay* and *Sendaddrv2* messages after sending the Version message. The *Wtxidrelay* message indicates that the potential peer supports relaying the transaction witnesses (data used to sign and verify transactions) using transaction ID instead of a separate witness transaction ID [46], and the *Sendaddrv2* message specifies the capability of a potential peer to understand the new version of the addr message (supports more address types, e.g., v3 Tor hidden service addresses than addr message) [47]. After receiving peer A's Version message, peer B also sends its own Version, *Wtxidrelay*, and *Sendaddrv2* messages 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 that 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. The Verack message is the acknowledgment of Version and is only sent when receiving the corresponding Version.

Figure 1 describes the Version handshake protocol in black color (while also including our Version++ additions), and we divide the handshake between those involved in Version and Verack (including *Wtxidrelay* and *Sendaddrv2*, which occur in between) vs. the complete handshake. We refer to all these message transmissions (from Version to Verack messages) between peer A and peer B as *Version-Verack Handshake* or *V.V. Handshake*. However, both peers also send two additional messages—*Sendheaders* and *Sendcmpct*—to each other during the handshaking. The *Sendheaders* message shows the preference of the potential peer to receive block header instead of the block hash [48], and the *Sendcmpct* message indicates that the potential peer supports compact block relay (protocol to reduce bandwidth and latency to transfer a block from one peer to another peer) [49]. Although there are no strict guidelines, Bitcoin implementations allow peers to send *Sendheaders* and *Sendcmpct* messages before or after the Verack message. However, we observe that those messages are sent after the Verack message during capture of the packets of Bitcoin handshaking messages. We refer to the *Sendheaders* and *Sendcmpct* message transmission between peer A and peer B, including the Version-Verack Handshake, as *Complete Handshake* or *C. Handshake*.

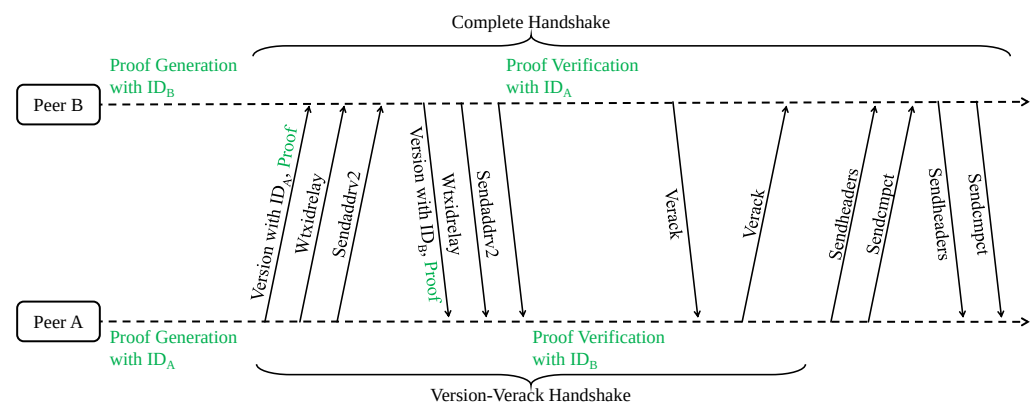


Figure 1. Version++ handshaking built on the Bitcoin handshaking protocol, where the two peers interchange their roles as prover and verifier in the handshaking. We highlight the Version++ additions to the existing Version in green color. The only additions from the current handshaking protocol are the inclusion of the Version++ Proof in the Version message (increasing the networking size) and the Proof generation and verification on the local peers (adding computing). Proof generation occurs offline, i.e., before initiating the Version++ handshaking protocol.

4. Version++ Scheme Protocol

In this section, we discuss the Version++ protocol in-detail. At first, we discuss the overview of the protocol followed by the variables and notations used in the paper along with the participants and prerequisites in the Version++ protocol. We then discuss the Proof Generation and Proof Verification process in the Version++ scheme.

4.1. Version++ Protocol Overview

Figure 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 and thus unique to the peer; Figure 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 highlights the Version++ addition of Proof. As shown in Figure 1, 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. While the existing Version-based protocol transmits the Verack upon receiving the Version, our Version++ protocol requires an additional verification of the proof before sending the Verack. If the Proof verification preceding the Verack transmission fails, then the verifier has no assurance that the prover indeed has the Bitcoin software version and can either not transmit the Verack (discontinuing the connection establishment) or transmit the Verack without the assurance (reduced to the existing Version-based protocol without assurance). In our Version++ implementation, we discontinue the connection establishment when the Proof fails in verification. The rest of this section describes the Version++ protocol in greater detail, including the Proof generation and the Proof verification, both of which use Merkle Tree.

Version	Services	Timestamp	Receiver Services	Receiver Address	Receiver Port	Sender Services
4 Bytes	8 Bytes	8 Bytes	8 Bytes	16 Bytes	2 Bytes	8 Bytes
Sender Address 	Sender Port	Nonce	User Agent	Start Height	Relay	Proof
16 Bytes	2 Bytes	8 Bytes	Size Varies	4 Bytes	1 Byte	32 Bytes

Figure 2. Version message data fields and the Version++ fields (highlighted and shaded in green).

4.2. 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 Figure 3). j defines the vertical position of node n , whereas i defines the horizontal position of node n according to Figure 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 positions are 2 and 1, respectively (denoted by the first orange circle in Figure 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. Table 1 lists the variables we use in this paper.

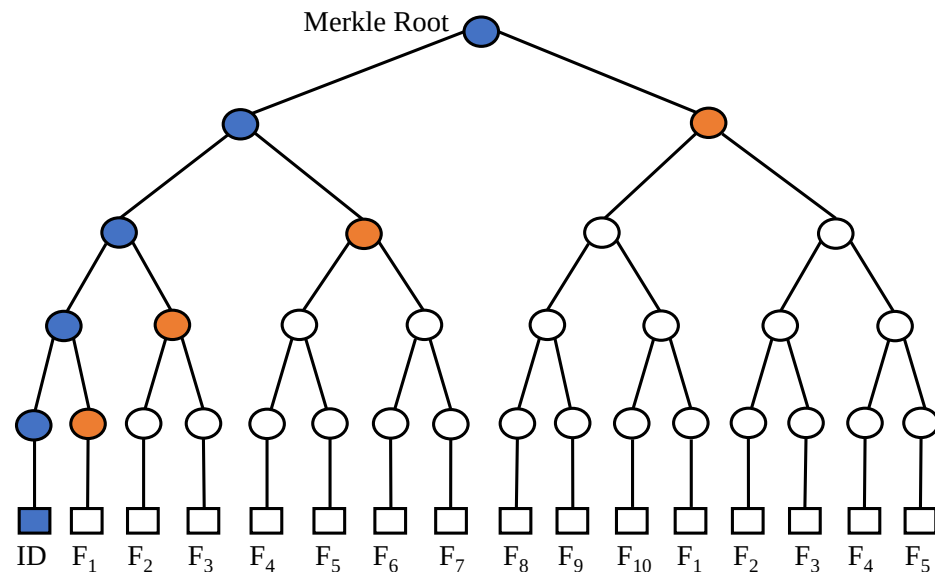


Figure 3. The binary Merkle tree structure of the Proof given the code files, F_1 to F_{10} .

Table 1. 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

4.3. 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) are involved in the peer connection establishment process, the peers become both the prover and verifier (interchanging the roles) during the Version++ handshaking because the Bitcoin peer connections are bidirectional, as shown in Figure 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 ID_A and ID_B , respectively, before sending the Version message, as depicted in Figure 1. During the Version++ handshaking, both of the potential peers exchange the Version message with its own ID and Proof for the Bitcoin software version it holds. *Version++ transmission* involves sending the Version message with Proof, while *Version transmission* does not have any Proof in the Version message. 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 Figure 1, peer A and peer B use the received ID (ID_B and 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 of the peers involved in the handshaking have successful Proof verification and have sent the Verack message to each other.

Version++ adds to the existing Bitcoin Version handshake protocol. We refer to the Version-Verack Handshake and Complete Handshake for the default Bitcoin handshaking protocol as *Version Version-Verack Handshake (Version V.V. Handshake)* and *Version Complete Handshake (Version C. Handshake)*, and for the Version++ handshaking protocol as *Version++ Version-Verack Handshake (Version++ V.V. Handshake)* and *Version++ Complete Handshake (Version++ C. Handshake)*.

4.4. 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 collision-resistant properties of the cryptographic hash function. The cryptographic hash functions' weak collision-resistant property (also known as the secondary preimage resistance) enforces that the prover use 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 the 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 remain 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.

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

4.6. Proof Verification

Version++ V.V. Handshake involves sending Version message with Wtxidrelay and Sendaddrv2 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 the Proof verification in detail. 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 circles in Figure 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 the prover's ID and comparing the regenerated Proof with the Proof received from the prover. A detailed description of the Proof verification algorithm is shown in Algorithm 1, which takes ID and m_r as input and outputs B . It computes $H(\text{ID})$ and puts it in the $n_{1,1}$ position of the Merkle Tree. It keeps concatenating the hash value in the $n_{i,j}$ and $n_{i+1,j}$ position, hashing the concatenated value and putting the hashed value in the $n_{i,j+1}$ position until j reaches 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).

Algorithm 1 Proof Verification

Input: ID, m_r
Output: B

```

 $B = 0$ 
 $i = 1$ 
 $j = 1$ 
 $n_{1,1} \leftarrow H(\text{ID})$ 
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  $B$ 
  
```

5. Implementation

We implement our scheme Version++ on active Bitcoin nodes in Testnet and in Mainnet. In Testnet, composed only of the nodes that we control, we implement Version and Version++ scheme in five cloud servers—Iowa, Oregon, Virginia, London, and Singapore—using Bitcoin Core software version 23.1 for this cloud-based implementation. The configuration of the cloud servers is e2-standard-2 machine type in Google Cloud Service with Ubuntu 20.04 server version 64 bits, 2 vCPUs (Intel Xeon Scalable Processor 1st generation 2 GHz), and 8 GB memory. We also implement Version and Version++ schemes for all the Bitcoin Core software versions (0.10.0–24.0.1) to date in our local machine with the configuration of 16 GB DDR4 RAM and AMD Ryzen Threadripper 3960X processor. We use *ping* to take the round-trip time (RTT) of the five cloud servers from our local machine.

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 [50]. 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 (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). For Version++ prototype implemented in our local machines, we use a network isolated from the internet and set two machines to run our Bitcoin Core instance. For Version++ cloud-based measurement, we use our local machine to connect it with the cloud servers. One machine uses a script based on Python 3.9 to repeatedly connect to and disconnect from the other peer, with a 500 ms delay between samples. We use *tcpdump* to capture the network traffic of Bitcoin, and extract the handshake information for each sample. Though we implement Version++ using Bitcoin 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. We have made our implementation available at <https://github.com/bitcoin-version-plus-plus/bitcoin-version-plus-plus> (accessed on 12 January 2024).

6. Experimental Analyses

In this section, we provide the analyses of Version++ with Version based on the measurements on local networking and cloud networking. The local networking measurement focuses on analyzing the overhead of different operations of Version++, including the comparison between Version and Version++, and the cloud networking measurement focuses on analyzing the Version++ and Version time duration, varying the geographical location of the peer.

6.1. Experiments in Local Networking

This section provides the description of the analyses performed based on our local machine. We provide latency analysis for Proof generation, Proof verification, Version transmission and Version++ transmission, the time duration of Proof generation, Proof verification and the distribution of code files of all the Bitcoin Core software versions to date, the number of files and size difference percentage across all the versions of Bitcoin Core software to date, comparison of Version++ V.V. handshake 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 Figure 4 for Proof generation, Proof verification, Version and Version++ transmission: 181.9 ms, 0.11 ms, 0.50 ms, and 1.70 ms, respectively. We also observe that Proof verification latency is $\frac{0.11}{20.97} = 0.524\%$ of V.V. Handshake latency (20.97 ms). Since the implementation of Version++ is carried out 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 (shown in Section 6.2). Also, Version++ transmission takes only 1.2 ms more than Version transmission. Therefore, the Version++ V.V. Handshake overhead is $\frac{1.31}{20.974} = 6.246\%$ in V.V. Handshake.

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 v24.0.1), and the number of Bitcoin Core versions updated during this period. The Bitcoin Core software update frequency on average is $\frac{1.671 \times 10^9 - 1.232 \times 10^9}{40} = \frac{4.394 \times 10^8}{40} = 1.098 \times 10^7 \text{ s} = 127.1 \text{ days}$ or 127 days 3 h 14 min 9.775 s. Therefore, a peer needs to go through the Proof generation once, taking 181.9 ms every 127.1 days on average.

Time Overhead: We show the peer connection duration distribution in Figure 5 by extracting the public node information from Bitnodes [51]. The average peer connection in the Bitcoin networking lasts for approximately 5.62 days (which is 4.856×10^8 millisecond) before it gets disconnected. Therefore, the verification overhead compared to the peer connection duration only takes $\frac{202.9}{4.856 \times 10^8} = 4.178 \times 10^{-7}$ of the average peer connection duration. The numerator values are in Figure 4, while the denominator is

from Figure 5. Therefore, the Version++ handshaking overhead is significantly smaller compared to the peer connection duration.

Storage Overhead: We show the distribution of the top 95% of Bitcoin software versions used by the peers in the Bitcoin network in Figure 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 code files for the implementation of Merkle Tree (i.e., Bitcoin vv1.17.0 RoninDojo and Bitcoin v1.18.1 RoninDojo) or are still making updates (i.e., Bitcoin Core v24.99.0). We show the storage overhead for Proof verification of Bitcoin software versions (among the top 95% distribution) in Table 2 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 code files or are still updating its files. Among the distribution of the top 95% of Bitcoin softwares, Bitcoin Core, Bitcoin Core alternatives, and Bitcoin Core exceptions have thirteen versions, two versions, and three versions, respectively. Bitcoin Core softwares are used by approximately 86.57% of peers in the Bitcoin network and storage overhead is 2.581 MB (Megabytes). Therefore, a Bitcoin node can verify approximately 86.57% of Bitcoin nodes by storing 2.581 MB of Merkle Tree. Bitcoin Core alternatives are used by 5.6% of peers and storage overhead is 0.367 MB. Therefore, a verifier can verify 92.17% of peers in the Bitcoin network by having a storage overhead of only 2.948 MB.

Proof Verification Overhead: Version++ is capable of identifying any modified version of the Bitcoin software in all cases. We know $|\hat{F}| = 2^{\lceil \log_2 |F| \rceil}$ from Section 4.5. Therefore, the depth of the Merkle tree is $\log_2 |\hat{F}|$. The number of files for bitcoin Core versions is as follows. From Bitcoin Core v0.10.0 to Bitcoin Core v0.14.3, from Bitcoin Core v0.15.0 to Bitcoin Core v23.1 and Bitcoin Core v24.0.1, they are in the range of 391 files to 504 files, 520 files to 988 files and 1055, respectively, according to Figure 7. Hence, $|\hat{F}|$ becomes 512, 1024, and 2048, resulting in a depth of the Merkle tree of 9, 10, and 11 for Bitcoin Core v0.10.0, Bitcoin Core v0.14.3, Bitcoin Core v0.15.0, Bitcoin Core v23.1, and Bitcoin Core v24.0.1, respectively. Therefore, we can observe from Figures 7 and 8 that Proof Verification latency only increases when the depth of the Merkle tree increases.

Proof Generation Overhead: Figure 9 shows the Proof generation time duration for all the Bitcoin Core version releases to date. As the number of files increases (evident from Figure 7), we can see that the Proof generation also increases accordingly. Figure 10 also shows the difference in percentage among the number of files and code file size for each of the Bitcoin Core version releases to date.

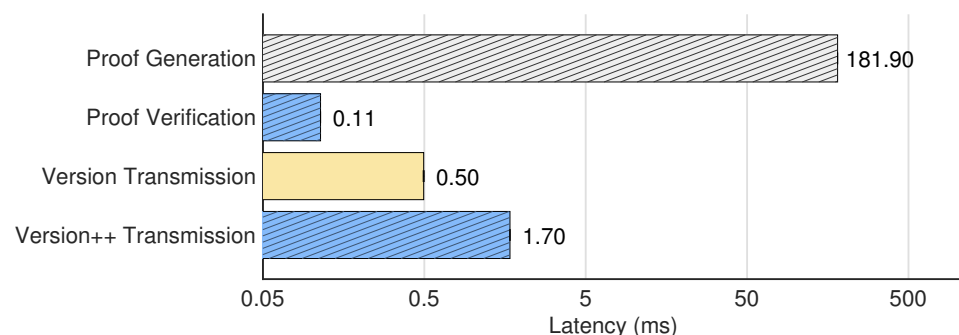
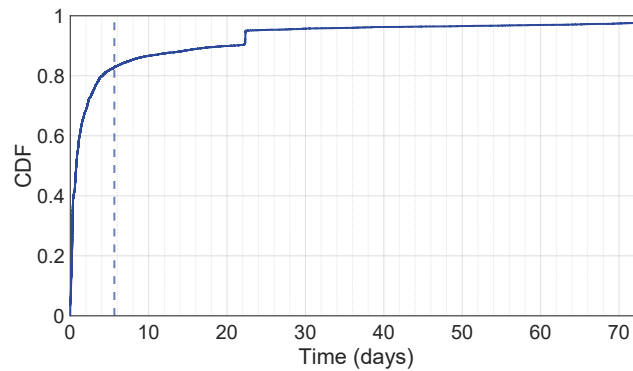
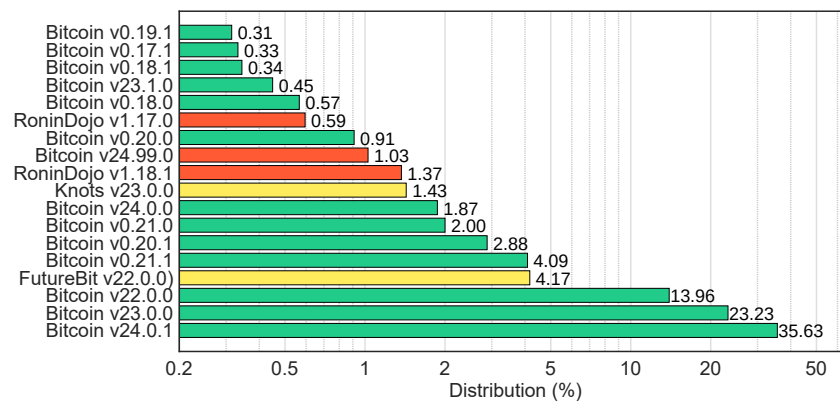
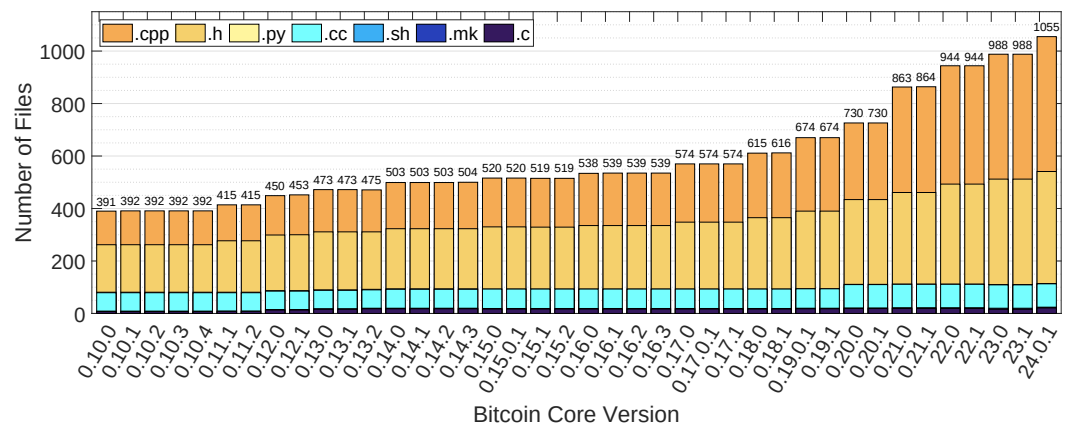


Figure 4. The latency measurements for the different operations of Version++ (our scheme) and the existing Bitcoin protocol (based on Version message). Gray bar represents the offline operation of Version++, blue bar represents online operation of Version++, and yellow bar represents the default Version scheme. The 95% confidence intervals are included but not visible.

Table 2. The storage overhead of the top 95% distribution of Bitcoin software versions.

Bitcoin Software	Storage Overhead (MB)
Bitcore Core	2.581
Bitcore Core alternatives	0.367
Bitcore Core exceptions	N/A

**Figure 5.** The distribution/CDF of duration that peers stay connected using real-world Bitcoin data.**Figure 6.** The distribution (%) of Bitcoin software versions among the nodes (top 95%) in Bitcoin network as of December, 2023. 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.**Figure 7.** The distribution of code files by file extension across the different Bitcoin Core versions.

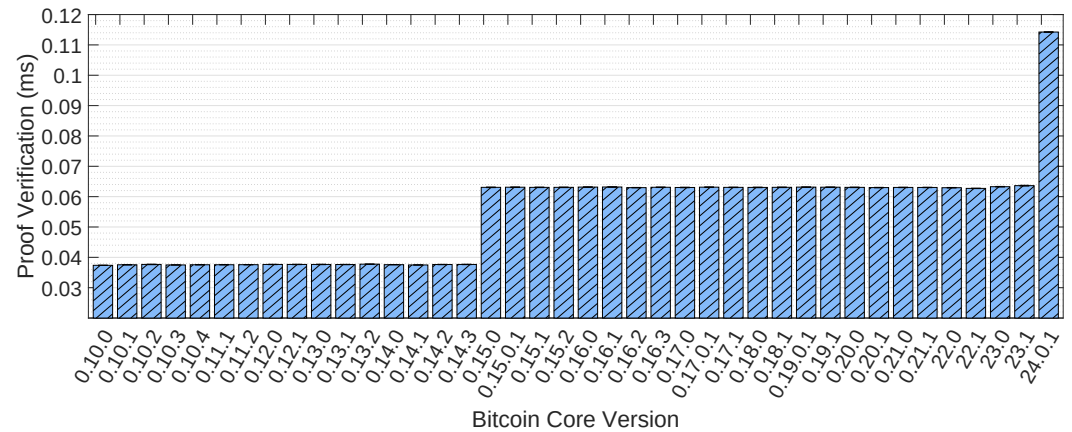


Figure 8. The Proof verification time duration for each Bitcoin Core version release. The 95% confidence intervals are included.

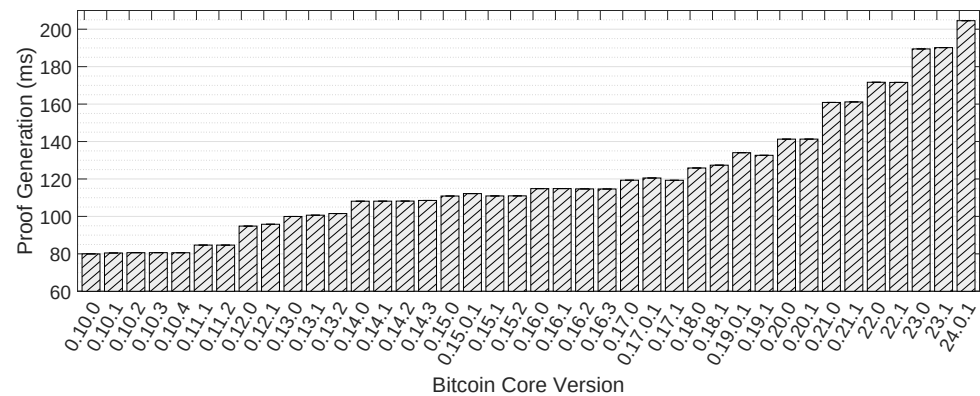


Figure 9. The Proof generation time duration for each Bitcoin Core version release. The 95% confidence intervals are included.

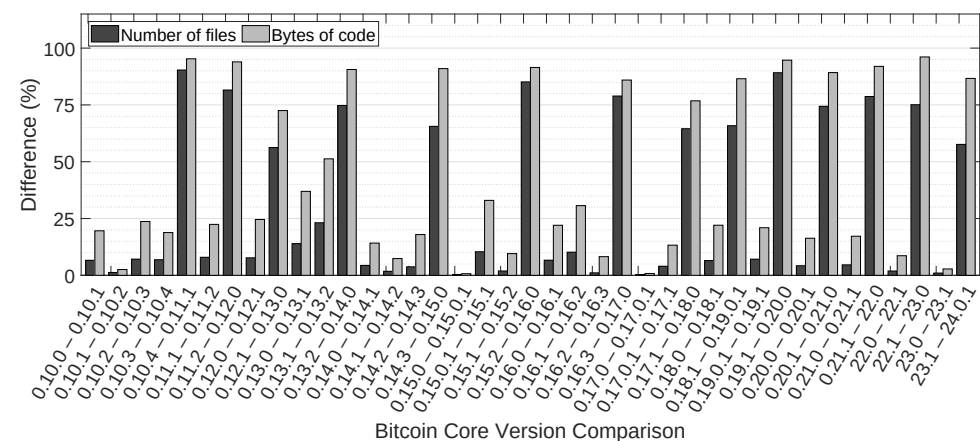


Figure 10. The number of files and size difference percentage across the Bitcoin Core version releases.

6.2. Experiments in Cloud Networking

In this section, we experiment with Version and Version++ scheme using cloud servers. We implement peer A in the local machine and peer B in the cloud server for each cloud server and then take the latency measurements for the Version and Version++ scheme. We also measure RTT using the five cloud servers and our local machine. We collect 1000 samples for each measurement.

Latency: Figure 11 shows the latency for RTT, Version V.V. Handshake, Version++ V.V. Handshake, Version C. Handshake and Version++ C. Handshake for each server.

The average latency for Version V.V. Handshake and Version++ V.V. Handshake is 36.29 ms, 49.07 ms, 58.52 ms, 126.7 ms, 189.5 ms, and 38.97 ms, 53.35 ms, 59.46 ms, 133.9 ms, 192.1 ms for Iowa, Oregon, Virginia, London, and Singapore respectively. The average latency between Version V.V. Handshake and Version++ V.V. Handshake differs only from 2 ms to 7 ms considering all servers. The average latency for Version C. Handshake and Version++ C. Handshake is 80.71 ms, 99.21 ms, 124.5 ms, 269.8 ms, 396.9 ms and 91.62 ms, 113.1 ms, 139.7 ms, 271.5 ms, 417.4 ms for Iowa, Oregon, Virginia, London, and Singapore, respectively. The average latency between Version C. Handshake and Version++ C. Handshake differs only from 2 ms to 21 ms considering all servers. Therefore, the Version++ V.V. Handshake overhead is between 1.336% and 8.722% compared to Version V.V. Handshake, and the Version++ C. Handshake overhead is between 0.663% and 13.998% compared to Version C. Handshake. Most importantly, Proof verification overhead is between 0.057% and 0.282% compared to the V.V. Handshake process.

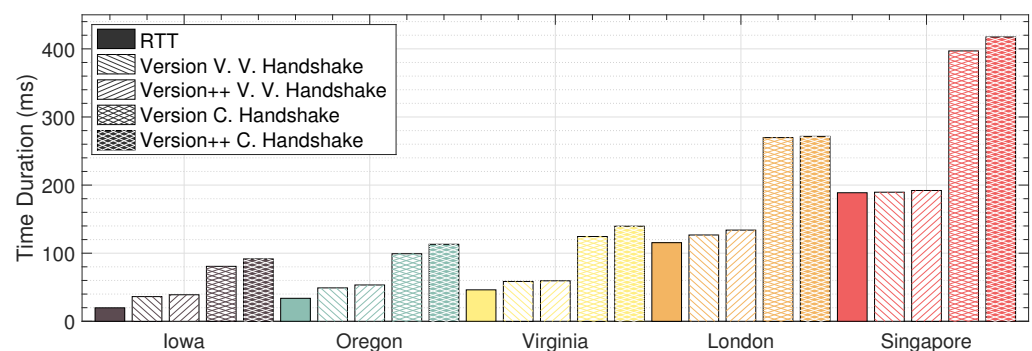


Figure 11. Comparison of the round trip time (RTT), the latency of Version-Verack Handshake (V.V. Handshake) and Complete Handshake (C. Handshake) for both Version and Version++, varying the geographical location.

The average latency for RTT is 19.86 ms, 33.79 ms, 46.23 ms, 115.5 ms and 188.8 ms for Iowa, Oregon, Virginia, London, and Singapore, respectively. Therefore, the average latency of Version++ V.V. Handshake and Version++ C. Handshake is 1.02 to 2 times and 2.21 to 4.78 times higher than RTT considering all servers. This difference is because of the protocol and implementation differences between RTT and Version++ scheme. We use *ping* to calculate RTT, which is the latency of sending packets using internet control message control (ICMP). On the other hand, we implement Version++ scheme using transmission control protocol (TCP) and the Version protocol. ICMP requires minimal overhead, as it is a connectionless protocol for testing the network, in contrast to Version++, which uses connection-oriented TCP protocol and Version protocol [52]. Moreover, the number of message transmissions between the peers to calculate each sample of Version++ (two for RTT and twelve for Version++) is also higher than RTT. Moreover, the Version++ scheme includes the Proof verification overhead.

7. Conclusions

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 the Bitcoin network. The analysis 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 4.178×10^{-7} times smaller than the average peer connection duration. Our future work includes applying Version++ to other cryptocurrency software, e.g., Ethereum, Litecoin, Cardano, Zcash, etc.

Author Contributions: Conceptualization, A.S., J.K. (Jinoh Kim), J.K. (Jonghyun Kim) and S.-Y.C.; Methodology, A.S. and S.W.; Software, A.S. and S.W.; Validation, A.S. and S.W.; Formal analysis, A.S.; Investigation, A.S. and S.-Y.C.; Resources, A.S. and S.-Y.C.; Data curation, A.S. and S.W.; Writing—original draft, A.S. and S.-Y.C.; Writing—review & editing, A.S. and S.-Y.C.; Visualization, A.S.; Supervision, J.K. (Jinoh Kim), J.K. (Jonghyun Kim) and S.-Y.C.; Project administration, J.K. (Jinoh Kim), J.K. (Jonghyun Kim) and S.-Y.C.; Funding acquisition, J.K. (Jinoh Kim), J.K. (Jonghyun Kim) and S.-Y.C. All authors have read and agreed to the published version of the manuscript.

Funding: 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).

Data Availability Statement: Our software implementation of the Version++ protocol is available at <https://github.com/bitcoin-version-plus-plus/bitcoin-version-plus-plus>. All obtained data in this work can be shared upon request.

Acknowledgments: This work builds on our previous conference paper [53] and the demo presentation [54], both at IEEE Consumer Communications and Networking Conference 2023. We significantly improve and extend from our previous conference paper for this journal manuscript. These extensions and additions include a more comprehensive and rigorous analysis of the Bitcoin handshake protocol (e.g., analyzing the Comprehensive Handshake beyond Version and Verack), the implementation and experimental analyses in cloud environments involving remote peer nodes (to simulate more realistic networking environments), the applications of our scheme Version++ to other Bitcoin Core software versions (to demonstrate the general applicability and enable the Version++ handshaking to greater peer nodes with different Bitcoin Core software). Our software implementation of the Version++ protocol is available at <https://github.com/bitcoin-version-plus-plus/bitcoin-version-plus-plus> (accessed on 12 January 2024).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. *Decentralized Bus. Rev.* **2008**, 21260. Available online: <https://bitcoin.org/bitcoin.pdf> (accessed on 26 September 2024).
2. CoinMarketCap. Trending Cryptocurrencies. 2022. Available online: <https://coinmarketcap.com/trending-cryptocurrencies/> (accessed on 20 August 2022).
3. Merkle, R.C. A digital signature based on a conventional encryption function. In Proceedings of the Conference on the Theory and Application of Cryptographic Techniques, Santa Barbara, CA, USA, 16–20 August 1987; Springer: Berlin/Heidelberg, Germany, 1987; pp. 369–378.
4. Merkle, R.C. Protocols for public key cryptosystems. In *Secure Communications and Asymmetric Cryptosystems*; Routledge: London, UK, 2019; pp. 73–104.
5. Fan, W.; Hong, H.J.; Wuthier, S.; Zhou, X.; Bai, Y.; Chang, S.Y. Security Analyses of Misbehavior Tracking in Bitcoin Network. In Proceedings of the 2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), Sydney, Australia, 3–6 May 2021; pp. 1–3.
6. Fan, W.; Hong, H.J.; Kim, J.; Wuthier, S.J.; Nakashima, M.; Zhou, X.; Chow, E.; Chang, S.Y. Lightweight and Identifier-Oblivious Engine for Cryptocurrency Networking Anomaly Detection. *IEEE Trans. Dependable Secur. Comput.* **2022**, *20*, 1302–1318. [CrossRef]
7. Armknecht, F.; Sadeghi, A.R.; Schulz, S.; Wachsmann, C. A security framework for the analysis and design of software attestation. In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, Berlin, Germany, 4–8 November 2013; pp. 1–12.
8. Castelluccia, C.; Francillon, A.; Perito, D.; Soriente, C. On the difficulty of software-based attestation of embedded devices. In Proceedings of the 16th ACM Conference on Computer and Communications Security, Chicago, IL, USA, 9–13 November 2009; pp. 400–409.
9. Kovah, X.; Kallenberg, C.; Weathers, C.; Herzog, A.; Albin, M.; Butterworth, J. New results for timing-based attestation. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20–23 May 2012; pp. 239–253.
10. Yang, Y.; Wang, X.; Zhu, S.; Cao, G. Distributed software-based attestation for node compromise detection in sensor networks. In Proceedings of the 2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007), Beijing, China, 10–12 October 2007; pp. 219–230.
11. Seshadri, A.; Luk, M.; Perrig, A.; Van Doorn, L.; Khosla, P. SCUBA: Secure code update by attestation in sensor networks. In Proceedings of the 5th ACM Workshop on Wireless Security, Los Angeles, CA, USA, 29 September 2006; pp. 85–94.

12. Seshadri, A.; Luk, M.; Perrig, A. SAKE: Software attestation for key establishment in sensor networks. In Proceedings of the Distributed Computing in Sensor Systems: 4th IEEE International Conference, DCOSS 2008, Santorini Island, Greece, 11–14 June 2008; Proceedings 4; Springer: Berlin/Heidelberg, Germany, 2008; pp. 372–385.
13. Choi, Y.G.; Kang, J.; Nyang, D. Proactive code verification protocol in wireless sensor network. In Proceedings of the Computational Science and Its Applications—ICCSA 2007: International Conference, Kuala Lumpur, Malaysia, 26–29 August 2007; Proceedings, Part II 7; Springer: Berlin/Heidelberg, Germany, 2007; pp. 1085–1096.
14. Gardner, R.W.; Garera, S.; Rubin, A.D. Detecting code alteration by creating a temporary memory bottleneck. *IEEE Trans. Inf. Forensics Secur.* **2009**, *4*, 638–650. [\[CrossRef\]](#)
15. Trusted Computing Group. Trusted Platform Module (TPM). 2023. Available online: <https://trustedcomputinggroup.org/work-groups/trusted-platform-module/> (accessed on 26 May 2023).
16. Intel. Intel Software Guard Extensions. 2023. Available online: <https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html> (accessed on 26 May 2023).
17. ARM. Layered Security for Your Next SoC. 2023. Available online: <https://www.arm.com/products/silicon-ip-security> (accessed on 26 May 2023).
18. Schellekens, D.; Wyseur, B.; Preneel, B. Remote attestation on legacy operating systems with trusted platform modules. *Sci. Comput. Program.* **2008**, *74*, 13–22. [\[CrossRef\]](#)
19. Sailer, R.; Zhang, X.; Jaeger, T.; Van Doorn, L. Design and implementation of a TCG-based integrity measurement architecture. In Proceedings of the USENIX Security Symposium, San Diego, CA, USA, 9–13 August 2004; Volume 13, pp. 223–238.
20. McCune, J.M.; Parno, B.J.; Perrig, A.; Reiter, M.K.; Isozaki, H. Flicker: An execution infrastructure for TCB minimization. In Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008, Glasgow, UK, 1–4 April 2008; pp. 315–328.
21. Noorman, J.; Agten, P.; Daniels, W.; Strackx, R.; Van Herreweghe, A.; Huygens, C.; Preneel, B.; Verbauwhede, I.; Piessens, F. Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. In Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13), Washington, DC, USA, 14–16 August 2013; pp. 479–498.
22. Wang, J.; Hong, Z.; Zhang, Y.; Jin, Y. Enabling security-enhanced attestation with Intel SGX for remote terminal and IoT. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2017**, *37*, 88–96. [\[CrossRef\]](#)
23. Kucab, M.; Boryło, P.; Cholda, P. Remote attestation and integrity measurements with Intel SGX for virtual machines. *Comput. Secur.* **2021**, *106*, 102300. [\[CrossRef\]](#)
24. Petroni, N.L., Jr.; Fraser, T.; Molina, J.; Arbaugh, W.A. Copilot-a coprocessor-based kernel runtime integrity monitor. In Proceedings of the USENIX Security Symposium, San Diego, CA, USA, 9–13 August 2004; pp. 179–194.
25. Xu, W.; Zhang, X.; Hu, H.; Ahn, G.J.; Seifert, J.P. Remote attestation with domain-based integrity model and policy analysis. *IEEE Trans. Dependable Secur. Comput.* **2011**, *9*, 429–442. [\[CrossRef\]](#)
26. Eldefrawy, K.; Rattanaivanon, N.; Tsudik, G. HYDRA: Hybrid design for remote attestation (using a formally verified microkernel). In Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks, Boston, MA, USA, 18–20 July 2017; pp. 99–110.
27. Eldefrawy, K.; Tsudik, G.; Francillon, A.; Perito, D. Smart: Secure and minimal architecture for (establishing dynamic) root of trust. In Proceedings of the NDSS Symposium 2012, San Diego, CA, USA, 5–8 February 2012; Volume 12, pp. 1–15.
28. Nunes, I.D.O.; Eldefrawy, K.; Rattanaivanon, N.; Steiner, M.; Tsudik, G. VRASED: A Verified Hardware/Software Co-Design for Remote Attestation. In Proceedings of the USENIX Security Symposium, Santa Clara, CA, USA, 14–16 August 2019; pp. 1429–1446.
29. Nunes, I.D.O.; Eldefrawy, K.; Rattanaivanon, N.; Tsudik, G. APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise. In Proceedings of the USENIX Security Symposium, Boston, MA, USA, 12–14 August 2020; pp. 771–788.
30. Koeberl, P.; Schulz, S.; Sadeghi, A.R.; Varadharajan, V. TrustLite: A security architecture for tiny embedded devices. In Proceedings of the Ninth European Conference on Computer Systems, Amsterdam, The Netherlands, 14–16 April 2014; pp. 1–14.
31. Brasser, F.; El Mahjoub, B.; Sadeghi, A.R.; Wachsmann, C.; Koeberl, P. TyTAN: Tiny trust anchor for tiny devices. In Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, 7–11 June 2015; pp. 1–6.
32. Kuang, B.; Fu, A.; Susilo, W.; Yu, S.; Gao, Y. A survey of remote attestation in Internet of Things: Attacks, countermeasures, and prospects. *Comput. Secur.* **2022**, *112*, 102498. [\[CrossRef\]](#)
33. Ammar, M.; Crispo, B.; De Oliveira Nunes, I.; Tsudik, G. 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, Abu Dhabi, United Arab Emirates, 28 June–2 July 2021; pp. 37–47.
34. Shi, E.; Perrig, A.; Van Doorn, L. Bind: A fine-grained attestation service for secure distributed systems. In Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05), Oakland, CA, USA, 8–11 May 2005; pp. 154–168.
35. Seshadri, A.; Perrig, A.; Van Doorn, L.; Khosla, P. SWATT: Software-based attestation for embedded devices. In Proceedings of the IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 9–12 May 2004; pp. 272–282.
36. Kennell, R.; Jamieson, L.H. Establishing the Genuinity of Remote Computer Systems. In Proceedings of the USENIX Security Symposium, Washington, DC, USA, 4–8 August 2003; pp. 295–308.

37. Seshadri, A.; Luk, M.; Shi, E.; Perrig, A.; Van Doorn, L.; Khosla, P. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, Brighton, UK, 23–26 October 2005; pp. 1–16.
38. Baines, M.; Drewry, W. Integrity-checked block devices with device mapper. In Proceedings of the Linux Security Symposium, Santa Rosa, CA, USA, 8 September 2011.
39. Chang, S.Y.; Wuthier, S.; Kim, J.; Kim, J. Lightweight Software Assurance for Distributed Mobile Networking. In Proceedings of the International Conference on Security and Management (SAM 2023), Las Vegas, NV, USA, 24–27 July 2023.
40. AlSobeh, A.M.; Magableh, A.A. BlockASP: A Framework for AOP-Based Model Checking Blockchain System. *IEEE Access* **2023**, *11*, 115062–115075. [\[CrossRef\]](#)
41. Magableh, A.A.; Al Sobeh, A.M. Securing software development stages using aspect-orientation concepts. *Int. J. Softw. Eng. Appl.* **2018**, *9*, 57–71. [\[CrossRef\]](#)
42. Alsobeh, A.M.; Magableh, A.A.A.R.; AlSukhni, E.M. Runtime reusable weaving model for cloud services using aspect-oriented programming: The security-related aspect. *Int. J. Web Serv. Res.* **2018**, *15*, 71–88. [\[CrossRef\]](#)
43. Zhang, Y.; Tan, R.; Kong, X.; Tan, Q.; Liu, X. Bitcoin Node Discovery: Large-Scale Empirical Evaluation of Network Churn. In Proceedings of the International Conference on Artificial Intelligence and Security, New York, NY, USA, 26–28 July 2019; Springer: Berlin/Heidelberg, Germany, 2019; pp. 385–395.
44. Tapsell, J.; Akram, R.N.; Markantonakis, K. An evaluation of the security of the bitcoin peer-to-peer network. In Proceedings of the 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), Halifax, NS, USA, 30 July–3 August 2018; pp. 1057–1062.
45. Bitcoindeveloper. P2P Network. 2022. Available online: https://developer.bitcoin.org/reference/p2p_networking.html#version (accessed on 20 August 2022).
46. Suhas Daftuar. WTXID-Based Transaction Relay. 2020. Available online: <https://github.com/bitcoin/bips/blob/master/bip-0339.mediawiki> (accessed on 17 April 2023).
47. van der Laan, W.J. addrv2 Message. 2019. Available online: <https://github.com/bitcoin/bips/blob/master/bip-0155.mediawiki> (accessed on 17 April 2023).
48. Daftuar, S. Sendheaders Message. 2015. Available online: <https://github.com/bitcoin/bips/blob/master/bip-0130.mediawiki> (accessed on 17 April 2023).
49. Corallo, M. Compact Block Relay. 2016. Available online: <https://github.com/bitcoin/bips/blob/master/bip-0152.mediawiki> (accessed on 17 April 2023).
50. Microsoft. Microsoft/merklecpp: A C++ Library for Creation and Manipulation of Merkle Trees. 2021. Available online: <https://github.com/microsoft/merklecpp> (accessed on 20 July 2022).
51. Bitnodes. Global Bitcoin Nodes Distribution. 2022. Available online: <https://bitnodes.io/dashboard/> (accessed on 1 August 2022).
52. Li, W.; Zhang, D.; Yang, J.; Xie, G. On evaluating the differences of TCP and ICMP in network measurement. *Comput. Commun.* **2007**, *30*, 428–439.
53. Sarker, A.; Wuthier, S.; Kim, J.; Kim, J.; Chang, S.Y. Version++: Cryptocurrency Blockchain Handshaking with Software Assurance. In Proceedings of the 2023 IEEE 20th Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, USA, 8–11 January 2023; pp. 804–809. [\[CrossRef\]](#)
54. Sarker, A.; Wuthier, S.; Kim, J.; Kim, J.; Chang, S.Y. Version++ Protocol Demonstration for Cryptocurrency Blockchain Handshaking with Software Assurance. In Proceedings of the 2023 IEEE 20th Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, USA, 8–11 January 2023; pp. 915–916.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.