electronics

MDPI

Article

Distributed and Lightweight Software Assurance in Cellular Broadcasting Handshake and Connection Establishment †

Sourav Purification [1,*], Jinoh Kim [2], Jonghyun Kim [3], Ikkyun Kim [3] and Sang-Yoon Chang [1,*]

[1] Department of Computer Science, University of Colorado Colorado Springs, Colorado Springs, CO 80918, USA
[2] Department of Computer Science and Information Systems, Texas A&M University–Commerce, Commerce, TX 75428, USA; jinoh.kim@tamuc.edu
[3] Electronics and Telecommunications Research Institute, Daejeon 34129, Republic of Korea; jhk@etri.re.kr (J.K.); ikkim21@etri.re.kr (I.K.)
* Correspondence: spurific@uccs.edu (S.P.); schang2@uccs.edu (S.-Y.C.)
† This paper is an extended version of our paper published in 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, USA, 24–27 July 2023; and Gamboni-Diehl, T.; Wuthier, S.; Kim, J.; Kim, J.; Chang, S.Y. Lightweight Code Assurance Proof for Wireless Software. In the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks, 2022, pp. 285–287, San Antonio, Texas, USA, 16–19 May 2022.

Abstract: With developments in OpenRAN and software-defined radio (SDR), the mobile networking implementations for radio and security control are becoming increasingly software-based. We design and build a lightweight and distributed software assurance scheme, which ensures that a wireless user holds the correct software (version/code) for their wireless networking implementations. Our scheme is distributed (to support the distributed and ad hoc networking that does not utilize the networking-backend infrastructure), lightweight (to support the resource-constrained device operations), modular (to support compatibility with the existing mobile networking protocols), and supports broadcasting (as mobile and wireless networking has broadcasting applications). Our scheme is distinct from the remote code attestation in trusted computing, which requires hardware-based security and real-time challenge-and-response communications with a centralized trusted server, thus making its deployment prohibitive in the distributed and broadcasting-based mobile networking environments. We design our scheme to be prover-specific and incorporate the Merkle tree for the verification efficiency to make it appropriate for a wireless-broadcasting medium with multiple receivers. In addition to the theoretical design and analysis, we implement our scheme to assure srsRAN (a popular open-source software for cellular technology, including 4G and 5G) and provide a concrete implementation and application instance to highlight our scheme's modularity, backward compatibility to the existing 4G/5G standardized protocol, and broadcasting support. Our scheme implementation incorporates delivering the proof in the srsRAN-implemented 4G/5G cellular handshake and connection establishment in radio resource control (RRC). We conduct experiments using SDR and various processors to demonstrate the lightweight design and its appropriateness for wireless networking applications. Our results show that the number of hash computations for the proof verification grows logarithmically with the number of software code files being assured and that the verification takes three orders of magnitude less time than the proof generation, while the proof generation overhead itself is negligible compared to the software update period.

Keywords: software-defined radio (SDR); software assurance; wireless networking; cellular networking; srsRAN; OpenRAN

check for updates

Citation: Purification, S.; Kim, J.; Kim, J.; Kim, I.; Chang, S.-Y. Distributed and Lightweight Software Assurance in Cellular Broadcasting Handshake and Connection Establishment. *Electronics* **2023**, *12*, 3782. https://doi.org/10.3390/electronics12183782

Academic Editors: Dejan Drajic, Zoran Cica and Philipp Svoboda

Received: 25 July 2023
Revised: 4 September 2023
Accepted: 4 September 2023
Published: 7 September 2023

Copyright: © 2023 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

Mobile networking control and function implementations are increasingly based on software. Such softwarization in control and function implementations include those for

security control, mobile applications, and software-defined radio (SDR). Networking softwarization facilitates/expedites advancements and innovations in wireless and mobile networking systems. The innovations and R&D in wireless systems enabled by networking softwarization include using radio-based or mobility-trajectory-based metrics for information/data processing and machine learning (e.g., [1–3]), adaptive and flexible channel spectrum access (e.g., [1,2,4–7]), intelligent and dynamic security control to protect the wireless channel (e.g., [4–6]), wireless sensor networking control (e.g., [8–10]), and magnetic- or induction-based communications control (e.g., [11,12]). The softwarization of radio frequency (RF) technology also facilitates open-source software development for wireless systems (e.g., srsRAN for 4G and 5G [13]) as well as transparency and inter-operability across wireless system vendors and implementations (e.g., OpenRAN [14,15]).

Due to the increased reliance on software implementations for mobile computing and networking, our goal is to achieve *software assurance* to ensure that a mobile user holds the correct software version/code for the networking implementations. Remote code attestation in trusted computing can provide such software assurance but is only available in centralized networking environments due to the technology's reliance on the trusted authority verifier. To enable software assurance in distributed networking applications forgoing centralized authority (c.f. internet-of-things or IoT), we design and build a software assurance scheme feasible and appropriate for distributed and lightweight networking. We distinguish our approach from remote code attestation by making it lightweight and distributed in hardware requirements (forgoing a hardware-based trusted computing/trusted execution environment), ecosystem (no need for trusted centralized server for verification), and in networking (no back-and-forth real-time networking). Our scheme also supports broadcasting communications by designing our proof to be prover-specific (but not verifier-specific) to enable one-to-many delivery. Because our scheme supports broadcasting applications with many receivers/verifiers, we prioritize the verification efficiency in our scheme design. However, our scheme for software assurance forgoing a centralized server and pre-established trust has a trade-off in the security property because it only assures that a wireless user/prover holds the software code files. In contrast, code *attestation* provides stronger integrity assurance, including the code execution. Section 2 further discusses remote code attestation and compares our scheme with that technology, and Section 7 discusses the application scope and relevance of our scheme.

We design our scheme to build a distributed and lightweight software assurance for mobile systems. Our scheme generates a user-specific proof by taking the software code files and the user ID as inputs. The user ID is distinct for each user; the user ID used in proof generation makes the proof unique to that user generating the proof. The proof-generating user shares their proof with the other mobile user who is interested in communicating. Our scheme prevents another user from generating the correct proof without having the correct software code files and from reusing other users' proofs. While building on the standard cryptographic primitives anchoring the modern-day digital security, we prioritize the distributed design (no need for centralized authority with pre-established keys) and the efficiency/lightweight operations (verification efficiency enabled by Merkle tree construction). Our scheme design thus enables the scheme in broader applications for mobile and wireless systems, including those for distributed ad hoc networking (without utilizing the networking-backend infrastructure), those based on resource-constrained devices (limited hardware and power capabilities), and those based on broadcasting communications (which wireless communications using the open air medium inherently supports).

In addition to the theoretical design and analysis (which is abstract and can be applied to different software and protocols), we implement our scheme on srsRAN (a popular open-source 4G/5G software for SDR [13]) to provide a concrete implementation instance of our scheme. We use srsRAN in two ways. First, we assure the srsRAN software; the srsRAN software becomes the input/object of our software assurance computing, and the proof generated and verified depends on the srsRAN software. Second, we build on the srsRAN software implementation to integrate the proof exchange and verification

in the cellular handshake and connection establishment in the radio resource control (RRC) in 4G/5G; the software assurance proof generation, however, computes on the software as downloaded from the srsRAN distributor rather than our updated software implementation to include the proof delivery in the RRC handshaking. We focus our implementation and experimentation on the scenario where the base station is the prover, and the proof is transmitted along with the base station's broadcasting message to initiate the RRC handshake with the user equipment. Such implementation highlights our scheme's broadcasting support and the compatibility with the existing 4G/5G standardized protocol. Furthermore, we describe and discuss some of the reliability-critical srsRAN software versions to highlight our scheme's application and utility.

The rest of the paper is organized as follows. We discuss related work in Section 2 and design our scheme, including the requirements and the prover and verification perspectives, in Section 3. Section 4 analyzes the computing overhead and how it scales with the number of code files for the assurance. We implement our scheme on srsRAN, open-source software for 4G/5G mobile networking implementations, in Section 5, while validating and analyzing the performance of our scheme in Section 6. In Section 7, we discuss our scheme's relevance and applications in wireless and mobile networking and highlight some of the srsRAN updates/versions we assure in our implementation in the previous section. We discuss our potential future research directions in Section 8, and Section 9 concludes our paper.

2. Related Work

We describe the related research in mobile networking to achieve software assurance/integrity. Section 2.1 describes the hash-based Merkle tree applications in other computing and networking contexts, as our work also utilizes Merkle tree as a part of our scheme but in mobile and cellular networking. Section 2.2 reviews the body of research in remote code attestation in trusted computing, which can provide software assurance (our goal) but relies on the trusted authority for verification and is thus prohibitive in decentralized and distributed environments.

2.1. Hash and Merkle Tree Application

Our scheme builds on the cryptographic hash function, which is often used for integrity protection of its arguments/inputs (the inputs can be networking payload or stored files or data). Among these hash applications are those using a Merkle tree structure for efficiency [16], similar to our approach. These applications using hash and Merkle tree include the block device integrity at the OS kernel level, e.g., dm-verity [17], file storage [18], the cryptocurrency software integrity [19], and the transaction integrity on cryptocurrency blockchains (Merkle tree is used for the data efficiency for ledger storage and networking) [20,21]. The Merkle tree and hash function also have applications in secure file transfer through the network [22], file storage in the cloud [23], and secure intelligent video surveillance [24]. Our work also uses the hash function and the Merkle tree structure as the underlying primitive/tool but with different inputs and for different applications of the software code assurance in mobile networking.

2.2. Remote Code Attestation

Related to our goal are the previous works in trusted computing that studied the remote code attestation using both the software and the hardware security (e.g., trusted platform module and trusted execution environment), real-time interactive protocols for providing fresh evidence (e.g., challenge and response), and a remote trusted server, e.g., [25–37]. Some of the aforementioned previous research specifically focused on the wireless and mobile networking context [34–37] and is thus especially related to our work. In [34], the authors proposed two attestation protocols for wireless sensor networks where the base station and the sensor node use node-specific identities during attestation and involve additional networking communications, including the challenge–response protocol (to

authenticate the attestation request). In [35], the nodes in the system rely on a trusted management center for registration, key distribution, and authentication for the attestation.

Remote code attestation has stronger security properties building on mutual authentication and implies assurance but requires greater resources in hardware [38], ecosystem, and networking. Its reliance on a trusted centralized server for attestation verification also challenges the deployment in the mobile applications in distributed or ad hoc environments. Its real-time interaction for fresh proofs between specific entities poses challenges to its modularity and deployment in broadcasting environments. For example, the remote code attestation literature requires the exact configuration and current/fresh states of the prover's computing and memory device during attestation [25–27,39,40]. In contrast, our work focuses on the source code files as downloaded and distributed by the software distributor (we do not use the compiled files for such a property). Furthermore, our work eliminates the real-time interaction between the entities for fresh proof and instead uses only local computing by the prover for the proof generation. Using the prover-specific ID for proof generation also makes our scheme feasible in broadcasting environments, as discussed in Section 1, in contrast to remote code attestation.

We do not intend to replace remote code attestation when remote code attestation can be afforded. However, our scheme can coexist with remote code attestation and enable greater assurance occurrences (cheaper and more lightweight in general). We design our software assurance scheme to broaden its applications and enable it in distributed, decentralized, and broadcasting environments. We discuss further our scheme's application scope and relevance in Section 7, including how it can coexist and be supplementary in centralized networking in Section 7.3.

3. Our Scheme: Design and Approaches

This section defines the prover and the verifier roles (these logical roles can be reversed between the wireless users), the proof generation process at the prover, the networking/communication exchange between the prover and the verifier, and the verification process at the verifier. Sections 3.2–3.5 describe the scheme and the protocol. Before describing our protocol scheme, Section 3.1 explains the requirements and how our design and approaches fulfill them; the first two requirements are for integrity, while the last two requirements are for correctness and efficiency by utilizing Merkle tree incorporation.

3.1. Approaches and Rationale

3.1.1. Requires the Software For Correctness and Integrity

We design our scheme so that only the users who have updated the software and have the corresponding files can generate and provide the assurance proof. We define the *attacker* as a user who attempts to generate or provide the proof without updating the software and having the software files. Such an attack violates the scheme's integrity because the attack, if successful, can assure and claim that it holds the software without actually holding it, defying our scheme's design goal. For security, to prevent an attacker without the files from generating the correct proof, we build on the well-established cryptographic primitives for our scheme, i.e., the one-way and collision-resistance properties of the cryptographic hash function. The one-way property, also known as the pre-image resistance, disallows the attacker from reversing the hash functions. Therefore, in our scheme, the attacker cannot derive the leaf nodes in the Merkle tree from the root proof. The weak collision resistance, also known as the secondary pre-image resistance, disallows the attacker from modifying the input of the hash function to find its corresponding hash output to match the expected one; in our scheme; the attacker cannot tamper with or control the file or the ID to derive the same root/proof of the Merkle tree as the expected proof. Furthermore, we take the following two approaches for security to disable an attacker from reusing the proof of other users (e.g., after eavesdropping). First, we design our scheme so that the proofs are unique to each user and depend on the user ID; we include the user ID (e.g., the user's IP address) as the input for generating the proof. Second, we only network the

proof output (the Merkle root) as opposed to the intermediate branch values, i.e., all the other intermediate values, except for the root proof, stay and are processed within the local host (similarly to a private key in public-key cryptography).

3.1.2. Synchronization

Because the proof generation and verification occur on separate users (called the prover and the verifiers), we require the proof to be consistent and the same across the users. We establish a rule for ordering the files for input into our scheme, which, in our implementation, is based on the file sizes. We also use the source code files, as opposed to the compiled files/executables or metadata, which can vary across the machine implementations and operating systems, for the inputs of our proof generation.

3.1.3. Broadcasting and Verification Prioritization

Our scheme is designed to support broadcasting with one transmitter and multiple receivers. Our scheme achieves broadcasting support by having the proof be prover-specific. The broadcast communications include the source user address (i.e., ID) but do not include a specific singular destination address. (In some broadcasting communications, such as Ethernet, a generic destination user address is used to address and transmit to multiple broadcasting receivers (i.e., broadcasting ID), but this is not specific to a single receiver/verifier.) In our scheme, the proof is prover-specific but not verifier-specific; the prover uses its source address as ID and source code files to generate the proof but does not use any information specific to the verifier. Different provers generate different proofs; however, given a prover and its proof, different verifiers use the same proof for verification. Hence, the proof does not include any information about a singular specific destination, enabling multiple receivers to verify the proof from multiple users. We use the Merkle tree to enable quicker verification of the prover-unique proof because only the ID changes across the different wireless users (while the rest of the code files remain the same). Using the binary Merkle tree, the verification complexity scales logarithmically with respect to the source code files, as opposed to the proof generation scaling linearly. We also prioritize the verification efficiency to better support the greater number of receivers/verifiers than the transmitter/prover in broadcasting and the greater verification occurrences than the generations in general, similar to the cryptographic ciphers prioritizing the efficiency of decryption/verification over encryption/signing. Our scheme's design to support broadcasting is a strength distinguishing our work from previous related work described in Section 2.2. We also choose a concrete implementation scenario, as a part of the 4G/5G radio control handshaking, to highlight these strengths in Section 5.

3.2. Introducing Prover and Verifier

In our scheme, the *prover* (providing the proof for its code assurance) requires generating the full tree to compute the root. The *verifier* is required to have the same code files (e.g., the same software version implemented) and have previously computed the prover operations. If a verifier does not have the same software and has not computed its own proof, then the verifier cannot assure the software.

3.3. Proof Generation at Prover

Our design uses the Merkle tree [16,41] with the code files and the user ID as inputs. We construct the Merkle tree as a binary hash tree where each parent node has two child nodes below itself. The parent node is the resultant hash value of the two child nodes. The top parent node is the Merkle root, hence the proof. The leaf nodes (i.e., the code files and user ID) are the initial child nodes of the Merkle tree in our scheme. We denote F as an ordered set of code files. To make it a balanced tree and have the number of leaves/files be a power of two, we introduce another set $\hat{\mathrm{F}}$ which is a cyclic extension of F with $2^{\lceil \log_2(|\mathrm{F}|) \rceil}$ number of elements (i.e., $|\hat{\mathrm{F}}| = 2^{\lceil \log_2(|\mathrm{F}|) \rceil}$). For example, in Figure 1, there are 10 code files, i.e., $\mathrm{F} = \{\mathrm{F}_1, \mathrm{F}_2, \ldots, \mathrm{F}_{10}\}$. We round this to the next power of two such that we construct a

new set $\hat{F}$ that has a size of $|\hat{F}| = 2^{\lceil \log_2(|10|) \rceil} = 2^4 = 16$. The additional files are generated by cycling through the previous files in F so that $\hat{F} = \{F_1, F_2, \ldots, F_{10}, F_1, F_2, \ldots, F_5\}$.

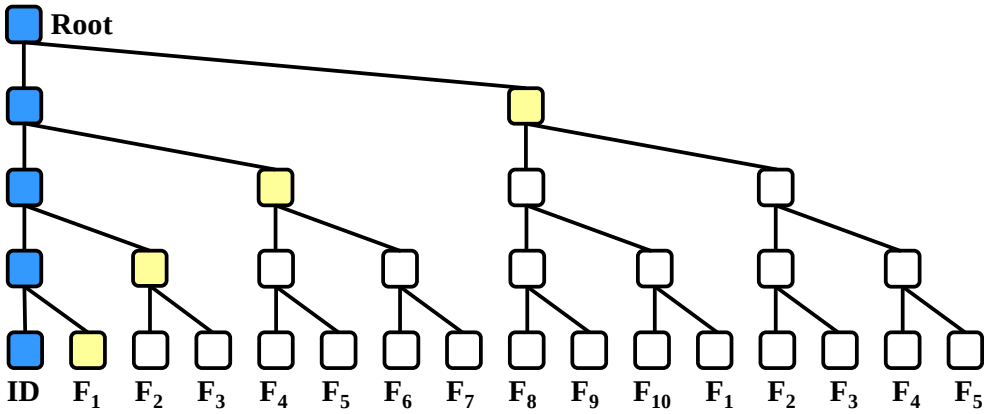


Figure 1. The binary Merkle tree structure of the proof (the "root") given the code files (F_1 to F_{10}).

3.4. Networking Proof from Prover to Verifier

Given $\hat{F}$, the prover generates the full tree, including the root which is the *proof* to be provided to the verifier. The other hash outputs/nodes within the Merkle tree stay within the prover's local machine and are not communicated/networked, i.e., the prover does not share the other hash outputs/nodes with anybody.

3.5. Verification at Verifier

To verify the prover's software, the verifier receives the proof from the prover and compares it (the received) with the one it computes/regenerates using the prover's ID (the computed/regenerated). The prover's ID is available at the verifier as a part of communication delivery. The locally stored nodes, marked in yellow in Figure 1, are known to those who hold the software code and have completed their own proof generation using their own ID. During verification, the locally stored nodes are used to regenerate the root/proof. For the regeneration, the verifier, thus, only needs to update the hash nodes which are affected by the ID (to be more precise, the prover's ID which is different from the verifier's ID). Hence, during proof verification, the verifier uses the prover's ID to regenerate the root/proof. In Figure 1, the nodes marked in blue and located on the far left correspond to the set of hashes that are unique to the ID and need to be regenerated for verification.

4. Computing Scalability Analysis

Our scheme can be applied to a part of the code requiring stronger assurance or the entire code base and its computational complexity depends on the number of code files being tested. Table 1 summarizes our computational scalability analysis. Section 6 measures how the overhead scales in physical time, including measuring the time overhead of a hash function computation, our scheme, and across software version updates varying the number of files.

Table 1. The computing overhead with respect to the number of files, $|F|$.

	Proof Generation	**Proof Verification**
Actual Overhead	$2^{\lceil \log_2(\|F\|) \rceil} - 1$	$\lceil \log_2(\|F\|) \rceil$
Bound Approximation	$2\|F\| - 1$	$\log_2(\|F\|) + 1$
Complexity	$O(\|F\|)$	$O(\log_2(\|F\|))$

4.1. Proof Generation at Prover

The prover generates the full Merkle tree as the proof (the root of the tree) for the proof generation at the prover that requires the entire tree construction. The number of hash computations required for the entire tree construction is $|\hat{\mathrm{F}}| - 1 = 2^{\lceil \log_2(|\mathrm{F}|) \rceil} - 1$. Because $2^{\lceil \log_2(|\mathrm{F}|) \rceil} \leq |\mathrm{F}|$, this is upper-bounded by $2|\mathrm{F}| - 1 = O(|\mathrm{F}|)$ and grows linearly with respect to the number of files, $|\mathrm{F}|$.

4.2. Proof Verification

The verification at the verifier is much more efficient than the proof generation by the prover because of the use of the Merkle tree. The verifier only needs to recompute the Merkle tree nodes which are updated due to the change in ID; it will regenerate part of the tree whose inputs are affected by the new prover ID received (the blue nodes in the example in Figure 1). The verification, therefore, only requires the number of hash updates of $\log_2(|\hat{\mathrm{F}}|) = \log_2(2^{\lceil \log_2(|\mathrm{F}|) \rceil}) = \lceil \log_2(|\mathrm{F}|) \rceil$. Given the number of files $|\mathrm{F}|$, the number of hash computations needed for verification is upper-bounded by $\log_2(2|\mathrm{F}|) = \log_2(|\mathrm{F}|) + 1 = O(\log_2(|\mathrm{F}|))$. The hash computation scales logarithmically with the number of files.

5. Implementation of Software Assurance on Cellular SDR Communication Software

While we describe and analyze our scheme in the abstract, using variables and theoretical analysis in Sections 3 and 4, we implement and incorporate our scheme in a cellular networking system to provide a concrete implementation instance in this section. We implement, validate, and test our scheme on srsRAN. srsRAN is an open-source software-defined radio (SDR) suite for researchers and developers in mobile networking, 4G LTE, and 5G NR [13]. We separate computing (intra-node operation) and cellular handshake incorporation (inter-node operation) for our presentation. We mostly focus on the latest srsRAN version 23.04 but test it with other software versions as well, for example, Section 6.3 tests and evaluates our scheme on all srsRAN versions. Our implementation is based on Java 17 and the C++ programming language. It uses the IP address for the ID in the computing and base station cell ID in the cellular handshake, while we use SHA-256 for the hash function for both of the implementations. The experiments are conducted over a range of samples ($10^3 \sim 10^5$), and we present the averages and the 95% confidence intervals to show the statistical significance of our experimental results. We make our source code repository public [42,43] for the community to reproduce the results.

5.1. Computing Software Assurance of srsRAN

We implement the proof generation and the proof verification as described in Section 3. The proof generation on the prover involves the construction of the Merkle tree (the Merkle root is the proof) where the source code files of srsRAN are input to the leaf nodes. The source code files are arranged in ascending order based on the file sizes and read into an *ArrayList* representing F, from which $\hat{\mathrm{F}}$ is derived. Each node in the Merkle tree includes the following data fields: the path to the file, the corresponding hash of the file, and the index mapping from files to the ordered set of leaf nodes $\hat{\mathrm{F}}$. We implement a binary Merkle tree, and the construction details of the binary Merkle tree are described in Section 3.3. On the other hand, the proof verification involves a partial reconstruction of the Merkle tree (updating the nodes affected by ID) and comparing it with the proof value transmitted by the prover. Initially, the verifier generates its proof using its ID and stores the nodes locally, as described in Section 3.5. The verification is successful if the reconstructed Merkle root is the same as the proof transmitted by the prover.

5.2. Cellular Handshake Incorporation in RRC

We incorporate our software assurance in the cellular handshake implemented in srsRAN. We build on the offline and intra-node operations of proof generation and verification computing (the srsRAN software is the subject and the input), and then use the

resulting proof and incorporate it in the real-time handshaking networking implemented by srsRAN.

While any entity can serve as the prover and the verifier, we focus on the scenario where the base station is the prover and the user equipment is the verifier to highlight the broadcast support strength of our scheme. More specifically, we focus on radio resource control (RRC) communications, where the base station broadcasts its RRC messages to the nearby user equipment. In our scheme implementation, the base station as the prover appends the proof along with the SIB1 message, and the user equipment, as the verifier, receives the proof and verifies the base station software.

We build on the standardized RRC protocol and incorporate our scheme into it so that it is backward compatible with the standardized protocol. Our implementation supports the modularity of our software assurance design and the backward compatibility by not introducing additional networking transmissions on the RRC handshake; it adds more bytes within the allowable range by the current 4G/5G standardization. Figure 2 illustrates the standardized RRC communication (black color) along with our scheme's incorporation (blue color). RRC, as the name indicates, focuses on pairing and radio resource establishment for the communication link between the user to the base station, including the medium access control (MAC) channel selection and control. RRC communication begins with the base station that broadcasts system information messages, namely, master information block (MIB), system information block type 1 (SIB1), and other system information blocks (SIBs) to the user equipment. Among these system information messages, the SIB1 message contains cell-specific important system parameters (such as network identifiers including cell ID) that user equipment use to initiate the connection with the base station. The user equipment establishes an RRC connection setup with the base station by a three-way handshake after receiving the broadcast messages.

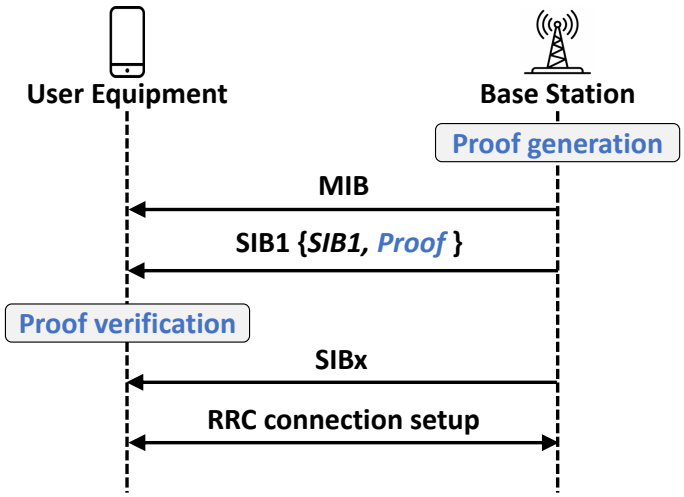


Figure 2. Our scheme's incorporation into 4G/5G cellular networking. Our scheme generates the proof and the proof is delivered as a part of the 4G/5G RRC handshake and connection establishment. The existing 4G/5G RRC handshake according to the 3GPP standardization is depicted in black, while our software assurance scheme incorporation is in blue.

In our implementation design (refer to blue color in Figure 2), the base station generates the *proof* of its software version using cell ID before initiating the RRC communication and appends it in an SIB1 message while broadcasting. After receiving the SIB1 message, the user equipment updates its Merkle root/proof using the cell ID (received in SIB1) and verifies the proof by matching it with the base station's *proof*. Upon successful proof verification, the user equipment initiates the RRC handshake. In the implementation, only 32 bytes of data are added into the standard SIB1 message (52 bytes), and only the proof verification overhead is contributed to the overall RRC handshake time.

5.3. User Equipment and Base Station Simulations and Hardware

We simulate a cellular user equipment and a base station for our cellular communication implementation and experimentation. We use a Mini PC to simulate the *user equipment* (the client of the cellular service provision) and a computer to simulate the *base station* (the immediate node communicating with the user equipment as part of the cellular service provider), the specifications of which are in Table 2. While we use additional hardware platforms to implement our software assurance computing (analyses of which are in Section 6.2), we use the Mini PC for the user equipment simulation because of its compatibility with srsRAN to enable the radio resource control (RRC) implementations. For example, srsRAN does not support the RRC implementations on the phone. We mainly focus our results and analyses on the user equipment (based on Mini PC) and the base station (based on the computer), except for in Section 6.2. We show the experimental setup for incorporating our software assurance in cellular RRC handshake using the base station and the user equipment in Figure 3. We use Ettus USRP B210 software-defined radios for wireless connectivity between the user equipment and the base station while maintaining a 5 meter line of sight distance. We modify the source code of the RRC protocol stack in srsRAN version 23.04 to include the *proof* inside the *SIB1* message, as described in Section 5.2.

Table 2. Hardware specifications for user and base station.

Simulating	Device Type	Processor	Memory
User Equipment	Mini PC	Intel 12th N95, 3.4 GHz	8 GB
Base Station	Computer	AMD Ryzen 7 5700U, 4.3 GHz	16 GB

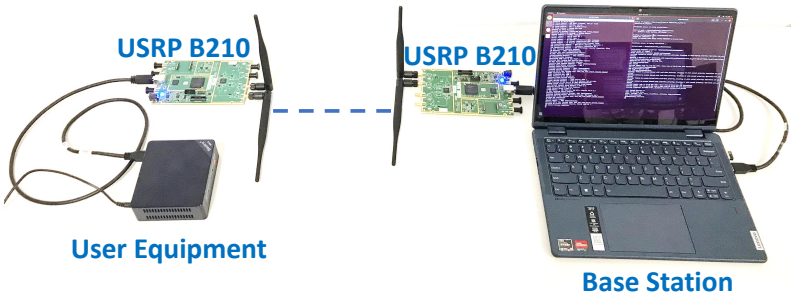


Figure 3. Experimental setup for cellular RRC handshake using base station (laptop) and user equipment (Mini PC) with USRP B210 SDRs while maintaining 5 m distance between them.

6. Experimental Results and Analyses

6.1. Computing Performances on User Equipment and Base Station

This section provides an analysis of the computational performance of our scheme on cellular network nodes, i.e., the user equipment and the base station. We perform this experiment to reflect how these network nodes (user equipment and base station) will perform when incorporated into a cellular networking environment with respect to computation. We analyze the performance based on the average value taken over 26,000 samples and the 95% confidence intervals to show the statistical significance. Figure 4 illustrates the computational performances (latency) of the user equipment and the base station for proof generation (*Proof Gen.*), proof verification (*Proof Ver.*) and underlying hash function computation only (*Hash Only*). The *Hash Only* focuses on the function itself without the peripheral overheads of reading and writing on the Merkle tree data.

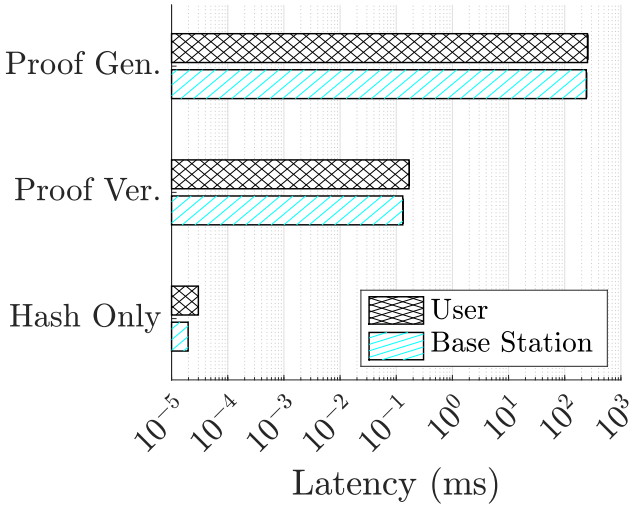


Figure 4. Time latency cost measurements for proof generation (*Proof Gen.*), proof verification (*Proof Ver.*), and hash function computation (*Hash Only*) for user equipment and base station. The 95% confidence intervals are included for *Proof Gen.* and *Proof Ver.*, although barely visible.

We simulate the user equipment and the base station such that the base station has higher computing power than the user equipment. This choice implies that the computational latency is lower for the base station than for the user equipment, which is illustrated in Figure 4. The user equipment requires $\frac{254.27}{242.12} = 1.05$ times greater time for proof generation and $\frac{0.169}{0.131} = 1.29$ times greater time for proof verification than the base station.

Our scheme is based on the Merkle tree structure to prioritize the verification due to its higher frequency in use with mobile/wireless networking (e.g., the user equipment verifies the base station proof every time it attempts RRC connection setup, as described in Section 5.2). Therefore, in our software assurance, proof verification is significantly more efficient than proof generation. The proof verification latency overheads compared to that of the proof generation are $\frac{0.169}{254.27} = 0.07\%$ and $\frac{0.131}{242.12} = 0.06\%$ for the user equipment and the base station, respectively. We investigate the impact of the computational performance of the user equipment and the base station while incorporating our scheme using the open-source cellular networking suite srsRAN in Section 6.4.

We measure the *Hash Only* computations that take generating hash functions only without other overheads (e.g., read/write file contents for constructing the Merkle tree). We take the average value only over the whole sample space (10^5 samples) and do not include the confidence interval. Furthermore, while the proof verification involves 11 hash computations (and the generation involves 2047 computations, including the inner-node computations within the tree), the read/write of the data from/to the Merkle tree data structure causes a sizable overhead in the proof verification. We estimate the hash-computation overheads of these operations from the *Hash Only* overhead measurements. The *Hash Only* computation contributes $\frac{(0.000030*2048)}{254.27} = 0.024\%$ and $\frac{(0.000018*2048)}{242.12} = 0.017\%$ overhead to the proof generation at the user equipment and the base station, respectively. For proof verification, the *Hash Only* computation contributes $\frac{(0.000030*11)}{0.169} = 0.195\%$ and $\frac{(0.000018*11)}{0.131} = 0.0168\%$ overhead at the user equipment and the base station, respectively.

6.2. Computing Performances on Other Platforms (Summary of Our Previous Work)

We implement our software assurance on more hardware platforms (beyond those used for the user equipment and base station, described in Section 5) to show the general applicability of our scheme and to simulate different hardware/computing capabilities. This section overlaps with our previous conference publication [44], and we, therefore, provide a summary to highlight the general applicability; our previous conference publication

presents more analyses and results on the phone and Raspberry Pi (refer to Figure 5) based on the average values of 10^5 samples. We use a phone (Samsung Exynos 9820 at 2.84 GHz clock frequency), Raspberry Pi (Broadcom BCM2711 Cortex-A72 1.8 GHz clock frequency), and a computer (AMD Ryzen Threadripper 3960X at 4.5 GHz clock frequency) to vary the hardware platforms and assure srsRAN version 22.04. Because srsRAN is not compatible with a phone or a Raspberry Pi for the RRC implementations (described in Section 5.2), we focus on presenting the results and analyses on the user equipment and the base station entities beyond this section.

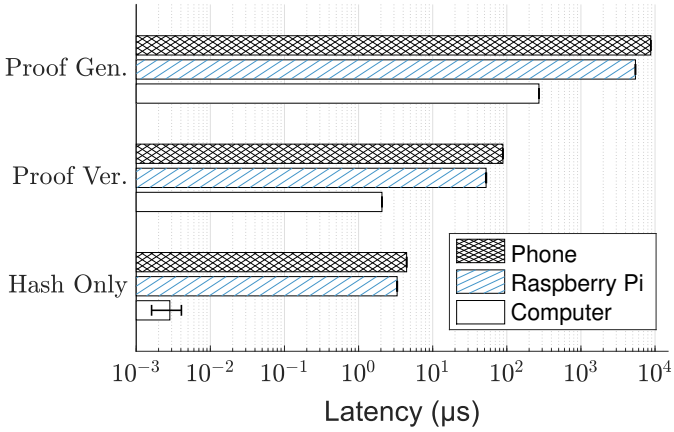


Figure 5. Time latency cost measurements for proof generation (*Proof Gen.*), proof verification (*Proof Ver.*), and hash function computation (*Hash Only)* across different computing platforms using srsRAN version 22.04. The 95% confidence intervals are included, although barely visible.

The computational overheads decrease as the computing capabilities increase from phones to Raspberry Pi to computers. For proof verification, the phone takes $\frac{88.59}{52.53} = 1.69$ times longer than the Raspberry Pi and $\frac{88.59}{2.066} = 42.9$ times longer than the computer. For the proof generation, the phone takes $\frac{8757}{5439} = 1.61$ times longer than the Raspberry Pi and $\frac{8757}{271.4} = 32.3$ times longer than the computer.

For the phone, the hash computations take $11 \cdot 4.49$ µs = 48.939 µs, which is $\frac{48.939}{88.59} = 55.24\%$ of the proof-verification overhead; for the Raspberry Pi, the hash computation takes $11 \cdot 3.312$ µs = 36.432 µs, which is $\frac{36.432}{52.53} = 69.35\%$ of the proof-verification overhead; for the computer, the hash computation takes $11 \cdot 0.00284$ µs = 0.03124 µs, which is $\frac{0.03124}{2.066} = 1.51\%$ of the proof-verification overhead.

6.3. Computing and Analyzing Different srsRAN Versions

While Section 6.1 and Section 6.2 present computational analyses with one srsRAN version (srsRAN v23.04 and srsRAN v22.04, respectively), we extend the analyses to other srsRAN versions in this section. We show the general applicability of our work, which is important as we envision a device using mobile networking to communicate with other devices which have different networking-implementation versions. This section focuses on the computational performances of the prover (base station) and the verifier (user equipment) with different versions of srsRAN. We take the average value over 1000 samples and include the 95% confidence intervals for the random experiments whose values are not fixed.

Since its initial release in June 2017 [45], the software has seen changes mainly in functionality and performance, with the number of code files tripling in size since its initial release (from 482 to 1582 for v2.0 and v23.04, respectively). Consequently, the time overhead to generate proofs and verify them has also increased. We compare the number of code files changed with the number of bytes for each code file changed throughout the software's lifespan in Figure 6.

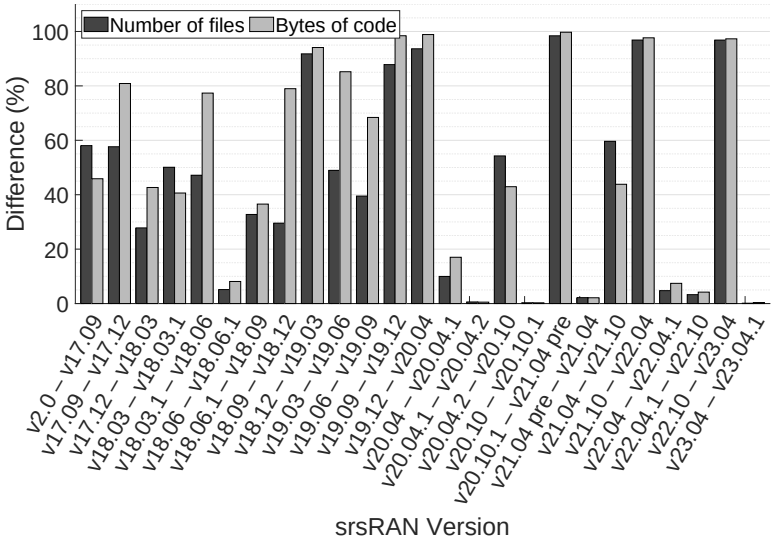
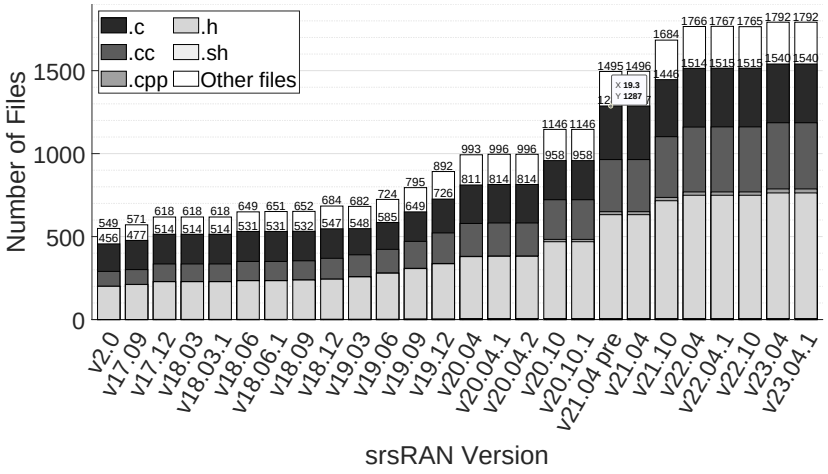


Figure 6. The changes in files and codes from the previous version.

Our scheme only assures the code files (consistent across the users) and excludes the "other files" (which can vary depending on the OS/system)) in Figure 7a. For example, $|\mathrm{F}| = 456$ for srsRAN v2.0 on the far left.

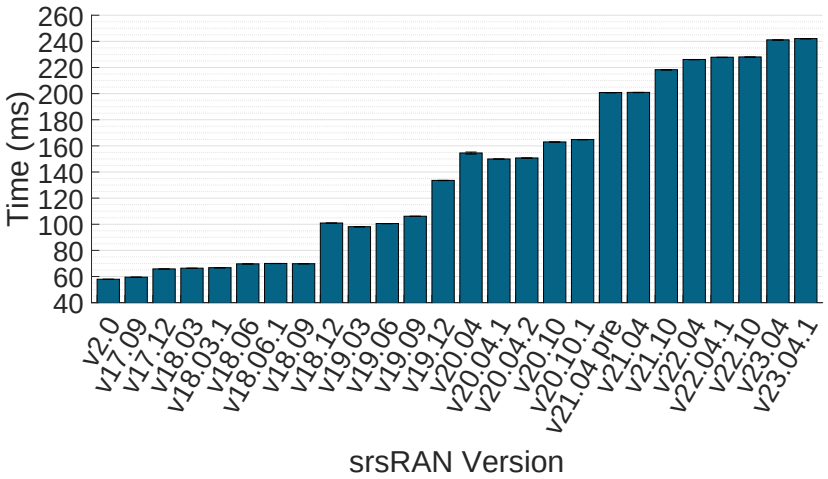
We compare the time overhead for the proof generation of the base station (including hashing the code files and constructing the Merkle tree) in Figure 7c across the software versions. The computational overhead for the proof generation significantly increases from 57.917 ms for the initial version to 242.118 ms for the latest version. Therefore, the proof generation cost is proportional to the number of software code files we approximate in Table 1. Most of the proof-generation time overhead is for reading the contents of the file and hashing them (97.26% on average across the srsRAN versions), while the remainder, for the Merkle tree construction and generating the final root/proof, is small (2.74%). Furthermore, the proof generation overhead is significantly smaller than the version update period, as the srsRAN versions were updated 26 times from 8 June 2017 to 15 July 2023. The srsRAN version is used for $\frac{2228}{26} = 85.7$ days or $\frac{2228 \cdot 24 \cdot 60 \cdot 60}{26} = 7.40 \cdot 10^6$ s on average (averaged across the version updates to date), which is significantly larger (seven orders of magnitude larger) than the proof generation costs (e.g., the maximum proof generation cost is 254.383 ms for v23.04). The proof generation is a single-time operation after any software version update, and the computational overhead (time, resource usage) depends on the number of source code files. In the real-world scenario, the frequency of software updates is much less than a single proof generation time. Therefore, our scheme is scalable in real-world software version update scenarios.

From the verifier's (user equipment) perspective, Figure 7b compares the time cost to verify the proof across the different srsRAN versions. Because the verifier has already constructed the Merkle tree of the same srsRAN version, it only needs to update the ID-dependent nodes in the tree, making the overhead logarithmic compared to the proof generation. As the srsRAN version is updated from v17.9 to v17.12 and from 20.10.1 to 21.04pre, the number of leaf nodes for the tree construction ($|\hat{\mathrm{F}}|$) changes from 512 to 1024, and from 1024 to 2048, respectively, causing a jump in computational time (by 0.039 ms and 0.077 ms, respectively). The verification time step size is almost two times ($\frac{0.077}{0.039} = 1.98$) greater than previously as the order of the leaf nodes increases. Because the number of files is large (ranging from 456 to 1582, as seen in Figure 7a), the proof verification is significantly more efficient than the generation, i.e., the proof verification is three orders of magnitude smaller than the proof generation for all of the srsRAN versions. To be more precise, the proof verification costs relative to the proof generation (i.e., the verification cost divided by the generation cost) for each srsRAN version range from 0.000501 (v23.04) to 0.000978 (v17.12).



(**a**)

(**b**)



(**c**)

Figure 7. Computational analyses of different srsRAN software versions. The 95% confidence intervals are included, although barely visible. (**a**) The distribution of files with each software version. Our scheme only assures the code files, which are consistent across the users. (**b**) Proof verification cost at the user equipment. (**c**) Proof generation cost at base station.

6.4. Cellular Handshake with Software Assurance

In this section, we analyze our software assurance while incorporating it into cellular RRC handshaking. For the analysis, we calculate the average value over 1000 samples and include the 95% confidence intervals to illustrate the value range. Our experimental results are presented in Figures 8 and 9. In Figure 8, we show the proof verification time (*Proof Ver.*) at the user equipment to verify the proof is $\frac{0.137}{234.616} = 0.058\%$ of the RRC handshake time. Hence, the proof verification computational cost at the user equipment has a significantly low impact on RRC connection completion time. The proof generation time (*Proof Gen.*) at the base station is 623.616 ms, which is independent of the RRC handshaking time. This is because the base station generates the proof after it receives the version update, which is prior to any RRC communication initiation (refer to Figure 2). However, proof generation is dependent on the software version update rate. We illustrate the srsRAN software version update rate in the last six years in Figure 9. The average number of software version updates per year is 4.33, which implies the software version is updated every 85.7 days. The base station needs to generate the proof of the software 4.33 times in a year (with a period of 85.7 days), and it takes only 623.616 ms each time, which is a significantly lower overhead with respect to the software update period. Furthermore, we see the *Proof Gen.* time while incorporated in the RRC handshake is $\frac{623.616}{242.12} = 2.6$ times higher than that without incorporation (refer to Figure 4). This happens because our software assurance scheme shares computing resources with the srsRAN application.

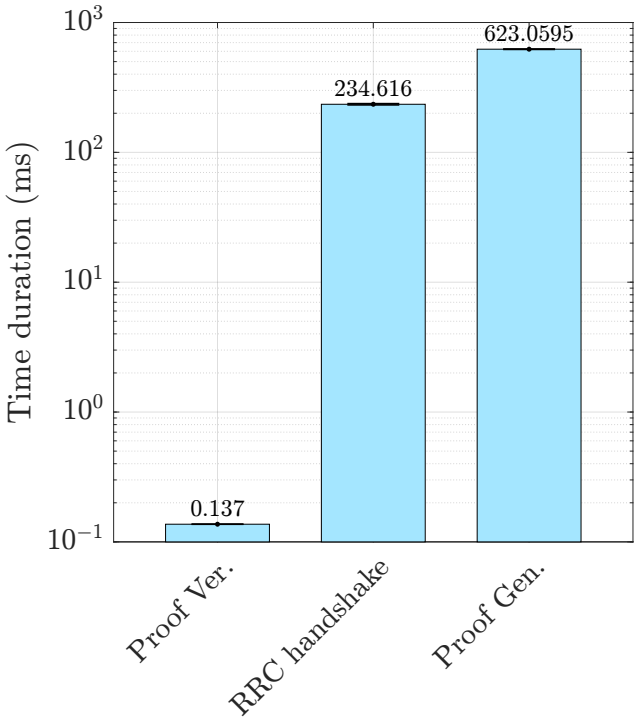


Figure 8. Proof verification, proof generation, and RRC handshake time. The 95% confidence intervals are included, although barely visible.

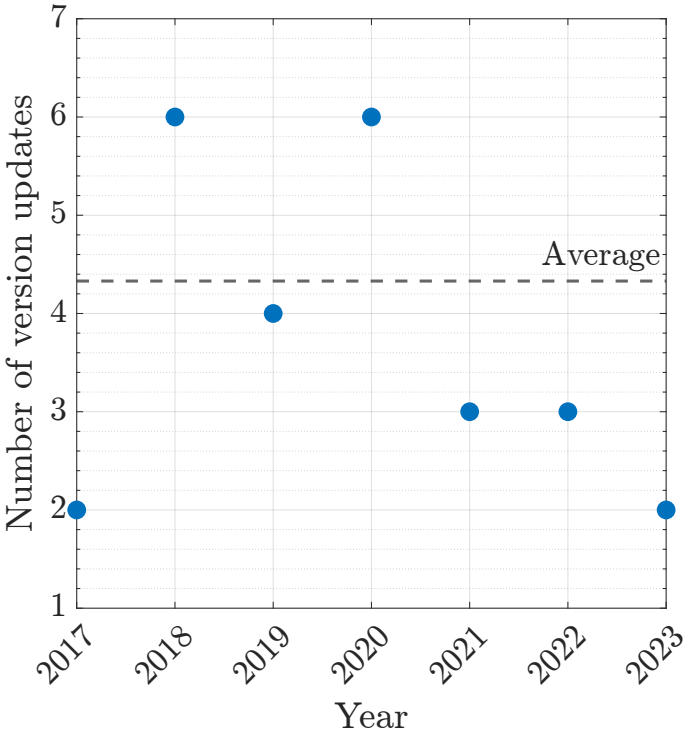


Figure 9. Number of srsRAN software version updates in last six years. The number of average software updates is 4.33 per year, as shown in the horizontal line.

7. Discussion About Relevance and Applications

7.1. Synergy with OpenRAN and Open-Source Networking Software Developments

Our work has high synergy with OpenRAN to improve transparency and open-source developments. srsRAN, the target software used in our implementation, provides the 4G/5G mobile protocol software implementations in open-source and in open-code to enable OpenRAN. We implement our scheme on all the srsRAN versions in Section 6.3. However, to provide more concrete utility and impacts of our scheme, we highlight some of the srsRAN version updates focusing on the reliability, stability, and security updates in Table 3. Applying our scheme to these software versions can provide greater assurances that the prover has the corresponding reliability, e.g., fixed bugs/issues/errors.

Table 3. Notable software srsRAN updates related to reliability, stability, and security.

Version(s)	Updates
v18.06.1	Fixed eNB instability and fixed eNB instability
v18.12	Add encryption support and refactor core network (EPC)
v19.03	Virtual radio (ZMQ)-based fake RF driver implementation and user-plane encryption for srsENB
v19.12	Add packet data convergence protocol (PDCP) discard
v20.04.1	Fix for UE MIMO segfault issue and fix GPS tracking synchronization
v20.04.2	Fix attach issue for some newer phones
v20.10.1	Fix bug in srsENB that effectively disabled uplink retransmission and error correction (HARQ)
v21.04pre	Improved error handling (S1AP) and enhanced event reporting
v21.04	Fixed crash when uplink-information transfer is received for invalid ID
v22.04.1	Fix crash when the user attempts to re-establish in standalone mode (SA)
v22.10	Fix DL NAS integrity checks in srsUE
v23.04	Updated 4G RRC ASN.1 to Rel 17
v23.04.1	Hotfix applying of dedicated PUSCH and PDSCH DMRS configuration

7.2. Lightweight, Distributed, Broadcasting-Friendly, and Expanding the Feasibility to More Applications

Our work emphasizes the lightweight and distributed design and implementation and, thus, provides greater deployment feasibility in enabling software assurance to broader applications than the remote code attestation in the following ways. First, the lightweight

design will enable functionality in resource-constrained devices with limited hardware and power/networking capabilities. Second, the lack of real-time and user-specific networking makes our scheme broadcasting-friendly. Third, the verifier can be any machine/user which has the software code it wants to verify so that our scheme can be applied to distributed, decentralized, and ad hoc environments. Therefore, in distributed networking forgoing a centralized server for sensor, device-to-device, or ad hoc networking, our scheme can achieve software assurance and verify the software codes/artifacts. In contrast, remote code attestation relies on centralized server operations and is, thus, prohibited in such networking environments. Examples of such distributed-networking applications include vehicular, drone, peer-to-peer (P2P), and military/defense field communications.

7.3. Limitations and Supplementary in Centralized Networking Environments

In other applications that can afford a centralized infrastructure, our scheme can supplement remote code attestation to enable verification/assurance with greater frequency and greater deployment places in the networking topology. For example, the base stations and the user-closer routers on the cellular service provider's edge can deploy our scheme for additional layers of verification and assurance. In contrast, there is no such verification/assurance on those edge network devices but only on the core network servers (hops/routers away from the user) in the current standardization and deployment of 4G and 5G [46]. We envision our software assurance scheme to be complementary to the remote code attestation in such centralized applications. For example, our software assurance can be used when the centralized verifier does not have stable connectivity (frequently unreliable) or in resource-constrained networking where additional real-time networking interactions become cost-dominant. However, our scheme does not replace remote code attestation in centralized networking applications, because remote code attestation can provide additional integrity protection beyond our scheme, such as real-time execution as discussed in Section 2.2.

8. Future Research Directions

8.1. Systems Study to Apply Our Scheme to Other Distributed Computing Applications

This paper takes a systems approach to incorporate our software assurance in cellular broadcast handshaking. Our scheme can be effective in checking whether a node holds the right software in other distributed networks, including internet of things (IoT), sensor networks, smart grid, vehicular networking, and distributed cryptocurrency [19], among others. We consider our scheme to be especially useful when distributed networking devices implement their functionalities based on open-source software. In these distributed networking contexts, the machines (the distinct users in ad hoc networking or the end devices and the edge devices in cloud networking) can utilize the software assurance to verify/assure the software version they are using. Such implementation omits the reliability of centralized servers to verify the software codes of the nodes which saves communication and computational overhead.

8.2. Coexistence Study with Code Attestation in Centralized Computing

Some mission-critical applications (including those in IoT, sensor networks, and smart grids) widely use remote code attestation, as discussed in Section 2.2, when centralized server and real-time commit-and-reveal can be afforded. Our lightweight scheme has higher feasibility as well as lower deployment requirements than the remote code attestation, enabling the scheme's implementation on greater nodes. For example, edge networking devices can provide software assurance of end devices using our scheme, and the real-time interaction between the end device and the centralized trusted server can occur for further code attestation later on.

8.3. Assuring Beyond the Software Code

We use source code files to generate proof of our software assurance scheme, i.e., we assure that the prover holds the software code. Incorporating a reproducible build of the software can improve our scheme and, more specifically, its security benefits. A build or compiler's outputs are *reproducible* "if given the same source code, build environment, build instructions, any party can recreate bit-by-bit identical copies of all specified artifacts" [47]. The reproducible build assures the consistency and integrity of the built software, making it more difficult for attackers to alter the code during its compiling/construction. Fourné et al., in [48], discussed the importance of the reproducible build for software security and provided recommendations on integrating reproducible builds in open-source software. It can be used in our software assurance scheme to generate the proof using the executable files because, in a reproducible build, the executable files are the same regardless of the environment in which it is built. Reproducible builds can aid in software security because of their defense against the compiler-level compromise and, therefore, can especially be useful in mission-critical systems. For example, related to our current paper (we incorporate software assurance on 4G/5G networking), mission-critical vehicular networking in cellular vehicle-to-anything (C-V2X) communications builds on the cellular networks of 4G or 5G and can incorporate reproducible build for greater software integrity than just using our scheme for assuring software code only.

9. Conclusions

We design and build a lightweight and distributed scheme forgoing hardware-based and centralized trusted computing for software assurance in mobile and resource-constrained device networking. Our assurance proof construction and verification build on the robust cryptographic primitives of the hash function for the integrity of our scheme so that only users who have the software can generate and verify the proof. Our proof also depends on the prover ID, and the prover and the verifier communicate only the proof (the minimal but sufficient information needed for the assurance). Our scheme utilizes the Merkle tree for proof verification efficiency. We analyze the scheme's scalability with the number of target software code files being assurance and show that the number of hash computations for proof verification grows logarithmically, in contrast to the proof generation growing linearly. In addition to theoretically describing and analyzing our scheme, we implement and incorporate our scheme into a cellular networking system using the popular open-source cellular implementation in srsRAN. Our implementation involves various computing processors and software-defined radios (SDRs) to simulate the user equipment and base station in 4G/5G networking. We integrate our scheme to assure the wireless software in the cellular networking handshaking in RRC to highlight the broadcasting support and compatibility to the 3GPP-standardized networking protocol. Our experiments show that the computational overhead in the time duration has a proof verification overhead three orders of magnitude smaller than the generation, and that the proof generation itself is seven orders of magnitude smaller than the average software version period (the average period between the software updates). Because our scheme is lightweight (in hardware requirement, ecosystem/server-reliance requirement, and networking), it can enable software assurance in broader wireless applications, including those requiring distributed, ad hoc, resource-constrained, or broadcasting-based operations. We anticipate our scheme will increase in its importance and use as more of these wireless applications emerge, e.g., internet of things (IoT).

Author Contributions: Conceptualization, J.K. (Jinoh Kim), J.K. (Jonghyun Kim) and S.-Y.C.; Methodology, S.P. and S.-Y.C.; Software, S.P.; Validation, S.P. and S.-Y.C.; Formal analysis, S.-Y.C.; Investigation, S.P., I.K. and S.-Y.C.; Resources, I.K.; Data curation, S.P.; Writing—original draft, S.P., J.K. (Jinoh Kim), J.K. (Jonghyun Kim) and S.-Y.C.; Writing—review & editing, S.P., J.K. (Jinoh Kim), J.K. (Jonghyun Kim), I.K. and S.-Y.C.; Visualization, J.K. (Jinoh Kim), J.K. (Jonghyun Kim), I.K. and S.-Y.C.; Supervision, J.K. (Jinoh Kim), J.K. (Jonghyun Kim), I.K. 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 the Institute of Information and Communications Technology Planning and Evaluation (IITP) grant funded by the Korean government (MSIT) (No. 2021-0-00796, Research on Foundational Technologies for 6G Autonomous Security-by-Design to Guarantee Constant Quality of Security).

Data Availability Statement: The data presented in this study are openly available in the GitHub public repositories.

Acknowledgments: This work extends our previous conference paper [44] and a poster paper [49]. In addition to overall presentation improvements, we take more of a systems approach for this journal manuscript than the conference paper. More specifically, while our conference version focuses only on the computing aspects of our software assurance scheme, assuming the proof delivery from the prover and the verifier, this journal paper incorporates our scheme in the RRC handshaking of 4G/5G cellular networking. Our scheme implementation, assuring the base station's software and delivering the proof along with the base station's broadcasting RRC message, not only shows that our scheme is feasible as a part of the cellular handshake but also highlights and demonstrates the broadcasting, modularity/compatibility, and efficiency strengths of our scheme. Our results and analyses, thus, were updated throughout, and we present them in this journal paper.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Clancy, C.; Hecker, J.; Stuntebeck, E.; O'Shea, T. Applications of machine learning to cognitive radio networks. *IEEE Wirel. Commun.* **2007**, *14*, 47–52. [CrossRef]
2. Thilina, K.M.; Choi, K.W.; Saquib, N.; Hossain, E. Machine learning techniques for cooperative spectrum sensing in cognitive radio networks. *IEEE J. Sel. Areas Commun.* **2013**, *31*, 2209–2221. [CrossRef]
3. Jiang, H.; Wang, M.; Zhao, P.; Xiao, Z.; Dustdar, S. A utility-aware general framework with quantifiable privacy preservation for destination prediction in LBSs. *IEEE/ACM Trans. Netw.* **2021**, *29*, 2228–2241. [CrossRef]
4. Chang, S.Y.; Hu, Y.C.; Laurenti, N. SimpleMAC: A Jamming-Resilient MAC-Layer Protocol for Wireless Channel Coordination. In Proceedings of the 18th Annual International Conference on Mobile Computing and Networking, Mobicom '12, Istanbul, Turkey, 22–26 August 2012; pp. 77–88. [CrossRef]
5. Chang, S.Y.; Hu, Y.C. SecureMAC: Securing Wireless Medium Access Control Against Insider Denial-of-Service Attacks. *IEEE Trans. Mob. Comput.* **2017**, *16*, 3527–3540. [CrossRef]
6. Vo-Huu, T.D.; Vo-Huu, T.D.; Noubir, G. Spectrum-Flexible Secure Broadcast Ranging. In Proceedings of the Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec '21, Abu Dhabi, United Arab Emirates, 28 June–2 July 2021; pp. 300–310. [CrossRef]
7. Liu, D.; Cao, Z.; Jiang, H.; Zhou, S.; Xiao, Z.; Zeng, F. Concurrent Low-Power Listening: A New Design Paradigm for Duty-Cycling Communication. *ACM Trans. Sens. Netw.* **2022**, *19*, 1–24. [CrossRef]
8. Luo, T.; Tan, H.P.; Quek, T.Q.S. Sensor OpenFlow: Enabling Software-Defined Wireless Sensor Networks. *IEEE Commun. Lett.* **2012**, *16*, 1896–1899. [CrossRef]
9. Lv, Z.; Kumar, N. Software defined solutions for sensors in 6G/IoE. *Comput. Commun.* **2020**, *153*, 42–47. [CrossRef]
10. Sriramulu, R.K.; Park, Y.; Chang, S.Y.; Liu, K. Dynamic Cost-Effective Emergency Network Provision. In Proceedings of the Proceedings of the First CoNEXT Workshop on ICT Tools for Emergency Networks and DisastEr Relief, I-TENDER '17, Incheon, Republic of Korea, 11–12 December 2017; pp. 37–43. [CrossRef]
11. Liu, G. A Q-Learning-based distributed routing protocol for frequency-switchable magnetic induction-based wireless underground sensor networks. *Future Gener. Comput. Syst.* **2023**, *139*, 253–266. [CrossRef]
12. Chang, S.Y.; Kumar, S.L.S.; Hu, Y.C. Cognitive wireless charger: Sensing-based real-time frequency control for near-field wireless charging. In Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), Atlanta, GA, USA, 5–8 June 2017; pp. 2302–2307.
13. Gomez-Miguelez, I.; Garcia-Saavedra, A.; Sutton, P.D.; Serrano, P.; Cano, C.; Leith, D.J. srsLTE: An open-source platform for LTE evolution and experimentation. In Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization, New York, NY, USA, 3–7 October 2016; pp. 25–32.
14. Yang, M.; Li, Y.; Jin, D.; Su, L.; Ma, S.; Zeng, L. OpenRAN: A Software-Defined Ran Architecture via Virtualization. In Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, SIGCOMM '13, Hong Kong, China, 12–16 August 2013; pp. 549–550. [CrossRef]

15. Yang, M.; Li, Y.; Jin, D.; Su, L.; Ma, S.; Zeng, L. OpenRAN: A Software-Defined Ran Architecture via Virtualization. *SIGCOMM Comput. Commun. Rev.* **2013**, *43*, 549–550. [CrossRef]
16. Merkle, R.C. A Digital Signature Based on a Conventional Encryption Function. In Proceedings of the A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology, CRYPTO '87, Santa Barbara, CA, USA, 16–20 August 1987; pp. 369–378.
17. 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.
18. Kan, J.; Kim, K.S. MTFS: Merkle-tree-based file system. In Proceedings of the 2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), Seoul, Republic of Korea, 14–17 May 2019; pp. 43–47.
19. Sarker, A.; Wuthier, S.; Kim, J.; Kim, J.; Chang, S.Y. Version++: Cryptocurrency Blockchain Handshaking With Software Assurance. In Proceedings of the 2023 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, USA, 8–11 January 2023.
20. Nakamoto, S.; Bitcoin, A. A peer-to-peer electronic cash system. *Bitcoin* **2008**, *4*, 2.
21. Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Proj. Yellow Pap.* **2014**, *151*, 1–32.
22. He, D.; Chan, S.; Tang, S.; Guizani, M. Secure data discovery and dissemination based on hash tree for wireless sensor networks. *IEEE Trans. Wirel. Commun.* **2013**, *12*, 4638–4646. [CrossRef]
23. Mao, J.; Zhang, Y.; Li, P.; Li, T.; Wu, Q.; Liu, J. A position-aware Merkle tree for dynamic cloud data integrity verification. *Soft Comput.* **2017**, *21*, 2151–2164. [CrossRef]
24. Lee, D.; Park, N. Blockchain based privacy preserving multimedia intelligent video surveillance using secure Merkle tree. *Multimed. Tools Appl.* **2021**, *80*, 34517–34534. [CrossRef]
25. 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.
26. 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.
27. 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.
28. 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.
29. 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. [CrossRef]
30. Coker, G.; Guttman, J.; Loscocco, P.; Herzog, A.; Millen, J.; O'Hanlon, B.; Ramsdell, J.; Segall, A.; Sheehy, J.; Sniffen, B. Principles of remote attestation. *Int. J. Inf. Sec.* **2011**, *10*, 63–81. [CrossRef]
31. Eldefrawy, K.; Rattanavipanon, 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.
32. Nunes, I.D.O.; Eldefrawy, K.; Rattanavipanon, 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.
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, WiSec '21; Abu Dhabi, United Arab Emirates, 28 June–2 July 2021; pp. 37–47. [CrossRef]
34. AbuHmed, T.; Nyamaa, N.; Nyang, D. Software-Based Remote Code Attestation in Wireless Sensor Network. In Proceedings of the GLOBECOM 2009—2009 IEEE Global Telecommunications Conference, Honolulu, HI, USA, 30 November–4 December 2009; pp. 1–8. [CrossRef]
35. Cao, J.; Zhu, T.; Ma, R.; Guo, Z.; Zhang, Y.; Li, H. A Software-Based Remote Attestation Scheme for Internet of Things Devices. *IEEE Trans. Dependable Secur. Comput.* **2023**, *20*, 1422–1434. [CrossRef]
36. Sisinni, S.; Margaria, D.; Pedone, I.; Lioy, A.; Vesco, A. Integrity Verification of Distributed Nodes in Critical Infrastructures. *Sensors* **2022**, *22*, 6950. [CrossRef] [PubMed]
37. 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.
38. Francillon, A.; Nguyen, Q.; Rasmussen, K.B.; Tsudik, G. A minimalist approach to remote attestation. In Proceedings of the 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 24–28 March 2014; pp. 1–6.
39. Yang, X.; He, X.; Yu, W.; Lin, J.; Li, R.; Yang, Q.; Song, H. Towards a low-cost remote memory attestation for the smart grid. *Sensors* **2015**, *15*, 20799–20824. [CrossRef]
40. Kiyomoto, S.; Miyake, Y. Lightweight attestation scheme for wireless sensor network. *Int. J. Secur. Its Appl.* **2014**, *8*, 25–40. [CrossRef]
41. Merkle, R.C. Protocols for public key cryptosystems. In *Secure Communications and Asymmetric Cryptosystems*; Routledge: New York, NY, USA, 2019; pp. 73–104.

42. Simeon Wuthier. srsRAN Version Comparison in Software Assurance. 2023. Available online: https://github.com/simewu/srsRAN-version-compare (accessed on 18 July 2023).
43. Simeon Wuthier. Software Assurance in srsRAN. 2023. Available online: https://github.com/BS-Authentication-NSSL/merkle_authentication (accessed on 18 July 2023).
44. 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'23), Las Vegas, NV, USA, 24–27 July 2023.
45. Srsran. srsRAN Releases. 2023. Available online: https://github.com/srsran/srsRAN_4G/releases (accessed on 18 July 2023).
46. 3GPP. TS 33.501. 5G; Security Architecture and Procedures for 5G System. 2022. Available online: https://www.etsi.org/deliver/etsi_ts/133500_133599/133501/16.03.00_60/ts_133501v160300p.pdf (accessed on 3 September 2023).
47. ReproducibleBuildsproject. Available online: https://reproducible-builds.org/docs/definition (accessed on 18 July 2023).
48. Fourné, M.; Wermke, D.; Enck, W.; Fahl, S.; Acar, Y. It's like flossing your teeth: On the Importance and Challenges of Reproducible Builds for Software Supply Chain Security. In Proceedings of the 44th IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 21–25 May 2023.
49. Gamboni-Diehl, T.; Wuthier, S.; Kim, J.; Kim, J.; Chang, S.Y. Lightweight Code Assurance Proof for Wireless Software. In Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks, San Antonio, TX, USA, 16–19 May 2022; pp. 285–287.

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.