applied sciences

MDPI

Article

Distributed Software Build Assurance for Software Supply Chain Integrity

Ken Lew [1,*], Arijet Sarker [2], Simeon Wuthier [1], Jinoh Kim [3], Jonghyun Kim [4] and Sang-Yoon Chang [1,*]

[1] Department of Computer Science, University of Colorado Colorado Springs, Colorado Springs, CO 80918, USA; swuthier@uccs.edu
[2] Department of Computer Science, Florida Polytechnic University, Lakeland, FL 33805, USA; asarker@floridapoly.edu
[3] Department of Computer Science and Information Systems, Texas A&M University-Commerce, Commerce, TX 75428, USA; jinoh.kim@tamuc.edu
[4] Electronics and Telecommunications Research Institute, Daejeon 34129, Republic of Korea; ikkim21@etri.re.kr
* Correspondence: klew2@uccs.edu (K.L.); schang2@uccs.edu (S.-Y.C.)

Abstract: Computing and networking are increasingly implemented in software. We design and build a software build assurance scheme detecting if there have been injections or modifications in the various steps in the software supply chain, including the source code, compiling, and distribution. Building on the reproducible build and software bill of materials (SBOM), our work is distinguished from previous research in assuring multiple software artifacts across the software supply chain. Reproducible build, in particular, enables our scheme, as our scheme requires the software materials/artifacts to be consistent across machines with the same operating system/specifications. Furthermore, we use blockchain to deliver the proof reference, which enables our scheme to be distributed so that the assurance beneficiary and verifier are the same, i.e., the node downloading the software verifies its own materials, artifacts, and outputs. Blockchain also significantly improves the assurance efficiency. We first describe and explain our scheme using abstraction and then implement our scheme to assure Ethereum as the target software to provide concrete proof-of-concept implementation, validation, and experimental analyses. Our scheme enables more significant performance gains than relying on a centralized server thanks to the use of blockchain (e.g., two to three orders of magnitude quicker in verification) and adds small overheads (e.g., generating and verifying proof have an overhead of approximately one second, which is two orders of magnitude smaller than the software download or build processes).

Keywords: software supply chain; assurance; software integrity; applied cryptography; blockchain

Citation: Lew, K.; Sarker, A.; Wuthier, S.; Kim, J.; Kim, J.; Chang, S.-Y. Distributed Software Build Assurance for Software Supply Chain Integrity. *Appl. Sci.* **2024**, *14*, 9262. https://doi.org/10.3390/app14209262

Academic Editors: Alexander Barkalov, Larysa Titarenko and Kazimierz Krzywicki

Received: 12 August 2024
Revised: 5 October 2024
Accepted: 6 October 2024
Published: 11 October 2024

1. Introduction

We use computer software products in our everyday lives from digital information transfer to sensor implementations to machine control. Due to our heavy reliance on software products, securing software development and its supply chain has become of paramount importance. Malicious modifications or injections, such as malware and backdoors, can compromise software, yielding vulnerabilities for applications and operations building on the software. Software development trends, such as open-source software projects, an increasing number of software repositories, and collective software developments, have resulted in even greater security risks, as they contribute to having more diverse, dynamic, and complex software supply chain involving many actors and contributors to software development.

Previous and ongoing research and development contribute toward securing software development and its supply chain. Such research and development includes the Software Bill of Materials (SBOM) and reproducible build. SBOM generates a record of software supply chain components to enable transparency, accountability, and tracking of the software development process. SBOM has been mandated for software products in commerce in the US by an executive order [1]. Reproducible build, on the other hand, is also known

as deterministic verifiable build because, given the source code and the build condition, it generates an identical build. Reproducible build can detect a compromise (malicious change, such as a backdoor or malware insertion at the compiler) in the build condition. Instead of advancing SBOM and reproducible build technologies themselves, we build on previous research and development for these technologies to secure their outputs.

We secure the software supply chain against malicious compromise or injections in various steps in the supply chain. More specifically, we build a software build assurance scheme to detect the following compromises [2,3]: there has been a compromise and a change/insertion in the build or distribution process, the source code has been modified, or the download repository provides a different/modified code. The user downloading the software can detect such changes and verify the software integrity confirming that the software has not been modified from that by the legitimate source. To achieve such goals, the user generates a proof based on the software artifacts (code, compiler outputs, SBOM outputs) and compares it with a reference proof. We rely on the security-robust cryptographic hash function to generate the proof so that a malicious adversary cannot manipulate the proof or avoid detection, e.g., the adversary's injection or change in the software, which can occur in different parts of the software download and build processes, changes the pseudo-random proof.

We use blockchain to serve as the trusted authority to provide the reference proof for verification on the user machine. Blockchain is appropriate for such purpose because it has high security in payload integrity (cannot change or modify the payload content on the blockchain ledger), resistance to a single point of failure (as long as the majority of the distributed participants follow the protocol), and high visibility and accountability (thanks to the distributed storage and easy read-access). We also utilize blockchain's strength in automatic broadcasting and distributed storage to distribute the proof offline, i.e., the reference proof can be shared any time before the software build assurance proof verification.

Our work applies generally to all software projects in principle. We thus describe our scheme in the abstract using variables in Section 4 to highlight the general applicability. For synergy and to provide a concrete implementation instance of our scheme for validations and system-based analyses, we implement our scheme in Section 5 targeting the Ethereum cryptocurrency software (developed by a collective group of developers, while implementing the operations to process transactions of approximately tens of billions of dollars market capitalization magnitude). We implement the blockchain, serving as a decentralized trusted authority for our scheme using Ethereum Smart Contract, which can be used to build various digital decentralized applications.

In this paper, we build on reproducible build and blockchain and apply them to achieve the following research contributions beyond the state of the art.

- We assure the software artifacts, including but beyond the source code.
- We make the software assurance distributed so that the assurance beneficiary and verifier align, i.e., the node downloading and using the software verifies its materials and artifacts.
- We achieve significant performance gains compared to the approach based on a centralized server.

2. Software Supply Chain Background

Generic Software Supply Chain Paths: The software supply chain starts with the developer who stores and shares the source code (s) in a repository. The source code, along with the dependencies, will then be used in the build process involving compiling the source code into machine-readable code. The build/compilation process outputs the software build (b) and the documentation (d).

There are two popular paths to download and build software. First, the user can download the source code and the dependencies from the source code repository and the dependencies repositories, respectively, and then build/compile the software themself. Second, a distribution platform can facilitate the download by providing the pre-build packages to the user. The user then compiles and builds the software from those packages.

Our work focuses on these two paths as they are the most typical paths for software download/build.

SBOM and Reproducible Build: More recent software download and build processes involve reproducible build for b and the software bill of materials (SBOM) for d. SBOM is mandated for the software for commerce and generates a record and documentation of the software supply chain components to enable transparency, accountability, and tracking of the software development process. Reproducible build, also known as deterministic verifiable build, provides an identical build output given the source code and the build condition to detect a compromise in the compiler or in the build condition. Major open-source software projects, such as Debian for Linux distribution, Tor for anonymous routing, and cryptocurrency blockchain (including Bitcoin Core and Ethereum), support reproducible build; we discuss the relevant recent research in Section 7.2. We also describe the concrete implementation of SBOM using Syft and reproducible build for Ethereum Archanes in greater detail in Section 5.

3. Problem Statement and Our Contribution

3.1. Problem Statement: Threats against Software Supply Chain

In this section, we describe the threats against the software supply chain that motivate our work. More specifically, we describe the threats residing within the main components and entities, described in Section 2 and illustrated in Figure 1.

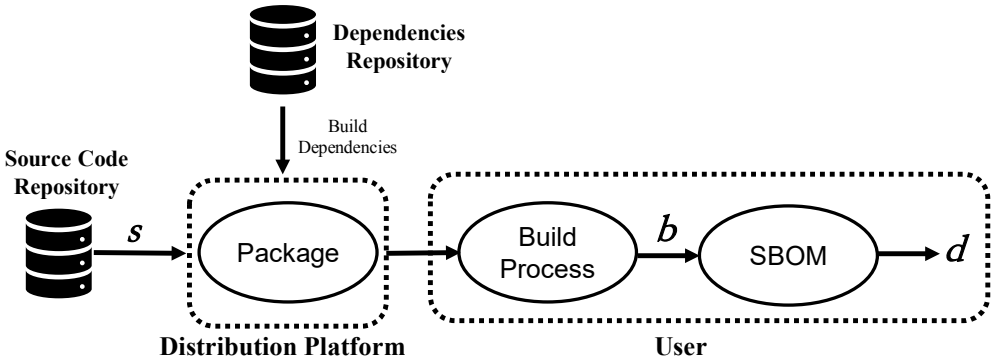

Figure 1. Software supply chain involving SBOM and reproducible build. This diagram illustrates the distribution-platform path, while the other user-direct path without the distribution is not illustrated due to space constraint.

Source Code Repository: Malicious actors can create an entirely new repository that is malicious from the creation. By doing so, victims will be downloading from a malicious repository. If it is an existing repository, malicious actors can also submit malicious merge requests that will turn into a backdoor or be exploitable by hiding the malicious intent via control characters, intentional concealing, and masking code differences [4]. This would effectively conceal malicious intent within the source code in the repository and hence compromise it. Another threat could also be to utilize configuration errors like leaked API keys in repositories and social engineering techniques that can empower a malicious actor to gain control of the source code or repository [5]. Once gaining control, malicious actors can modify the repository to carry out malicious intent.

Dependencies Repository and Build Process: Similar to source code repositories, the build process involves retrieving dependencies from designated repositories. This is facilitated by the utilization of package managers like npm and pip. This, however, could also lead to vulnerabilities like injections of malicious code from repositories into the dependency tree [6] that would affect the build process. Another type of threat that is not related to compromise of the dependencies repository but is related to the build process would be typosquatting or package confusion [7]. This technique is where malicious actors rely on and exploit common typing errors of users based on misspelling, character substitution, and pluralization of package names to carry out the attack. As examples, 'ethereum-go' vs. 'etheruem-go', 'backdoor-pkg' vs. 'backdoor-pkgs', and many more.

Distribution Platform: While malicious actors can attempt to inject malicious code into the source code repository, which would compromise the software supply chain, the distribution platform may also often rely on third-party dependencies and if these dependencies have vulnerabilities, it is another attack vector that malicious actors may choose to exploit.

SBOM: CISA suggests various types of SBOM, but in our case, we are focusing on the build type [8]. This means the SBOM creates a list of all the components needed for our project from the build process (b). If both the dependencies repository and the build process are compromised, it puts the accuracy of the SBOM at risk.

3.2. Our Contributions: Assurance Targets Across Software Supply Chain and Blockchain for Distributed Authority

We build a software build assurance scheme to check and assure the integrity of the various steps in the supply chain to defend against the active modification and malicious threats described in Section 3.1, More specifically, we use the software artifacts generated from the supply chain, namely, the source code (s), the build output including compilation (b), and the SBOM documentation (d), as the inputs and targets of our software assurance. We describe and explain these software artifacts as well as the software build assurance scheme in greater detail in Section 4.

In addition to assuring the software in multiple stages in the supply chain, our work advances beyond the approaches requiring a centralized authority and real-time protocol interactions with the centralized server, c.f., remote code attestation. Our scheme does not involve such centralized authority and instead uses blockchain to replace the centralized server authority. We also build on recent software development technologies of reproducible build and SBOM to enable our scheme; the reproducible build, in particular, enables our use of blockchain because it generates a deterministic software artifact in b to enable multiple recipients.

Our software build assurance is agnostic to the software applications, and our work can apply to any software in principle. However, our work requires reproducible build so that the software artifacts/compiler outputs are consistent across different machines If the software artifacts (the inputs of our assurance scheme) are not consistent and vary across machines, the verification does not work due to our scheme's use of the hash function (which produces a different output if using different inputs). While we provide a concrete validation using one software application for the Ethereum blockchain, we leave further validation and demonstration of other software applications to show the general applicability for future work. Previous research, however, has applied similar techniques as ours on other software applications, e.g., Bitcoin [9] and in cellular networking [10], although these assure only the software codes (which tends to be more consistent, e.g., a legitimate software source provides the same set of source code files).

4. Our Scheme

We build on SBOM and reproducible build, described in Section 2. We provide *assurance* of the software artifacts so that any changes and modifications of the software artifacts are detected. We thus take the outputs and the artifacts of the software supply chain, including those from SBOM d and reproducible build b, and input them into our scheme. The user of the software development, and the entity downloading the software conduct the software build assurance.

We present our scheme using abstraction and variables in this section while providing a more concrete implementation instance in Section 5.

4.1. Our Protocol Overview

The software supply chain makes use of the source code s and builds/compiles the software to generate b and the documentation d, as described in Section 2. Our scheme uses the SBOM to generate d and reproducible build to generate b so that b, given the user's compiler and the build conditions, provides the same b. Thanks to the use of reproducible build, other entities, given the same build condition and environment, also produce the same b. Our scheme, described in Figures 2 & 3 executed by the user, takes these software

artifacts, i.e., s, b, d, to construct a proof p. Table 1 lists these variables used in our paper, and the rest of the section describes our scheme in greater detail.

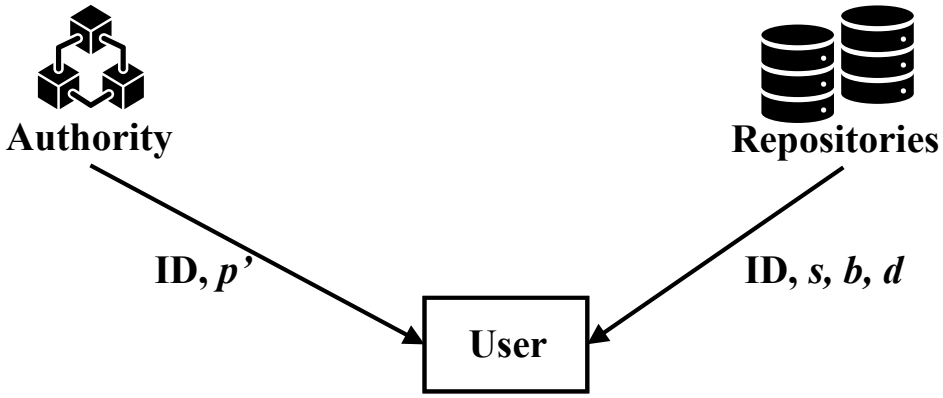

Figure 2. Our scheme overview, including information sources and inputs from the authority (pro-
rmation
elies on

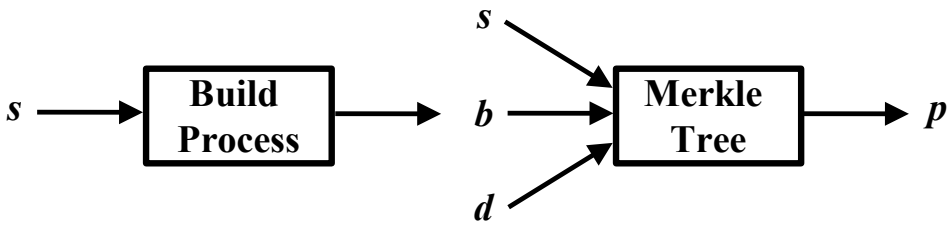

Figure 3. Online process from source code s to build/compilation b to the generation of proof p.

Table 1. Variable descriptions (Section 4) and assignments for implementation (Section 5).

Variable	Description	Our Implementation
s	Source code files of the specific version of the target software downloaded by user	Source code files of the latest Ethereum software, Archanes (v1.13.4)
b	Reproducible build compiler output from user	Geth of Archanes (v1.13.4) stored in the Ethereum repository [11]
d	SBOM documentation of the specific version of the target software generated by user	SBOM documentation of the Ethereum software, Archanes (v1.13.4)
p	Proof for the target software generated by the user	Proof for Archanes (v1.13.4) generated by the user
s'	Source code files of the specific version of the target software from authority	Source code files of the latest Ethereum software, Archanes (v1.13.4)
b'	Reproducible build compiler output from infrastructure from authority	Geth of Archanes (v1.13.4) stored in the Ethereum repository [11]
d'	SBOM documentation of the specific version of the target software from authority	SBOM documentation of the Ethereum software, Archanes (v1.13.4)
p'	Proof reference from blockchain	Proof stored in the Ethereum blockchain and centralized server
ID	Index of Proof	Index of Proof, p' in the Ethereum blockchain
H	Hash function	SHA-256

The authority is a logical entity that provides the information to the user about what the software artifacts and the proof should be, which we call the references and use the prime variables in Table 1. The authority can be implemented in many ways as long as it is an entity that has ownership over the source code in terms of storing or distributing it to users, for example, the software vendor who developed it. The authority provides the reference build (b'), the reference documentation (d'), the software ID and the reference proof (p') to the user. p' is generated in the same way as p but only from the authority. p' is computed by the authority and is a deterministic function of s', b', d', similarly to how p is computed by the authority and is a deterministic function of s, b, d. Thus, if $s = s'$, $b = b'$, and $d = d'$ and there has been no unauthorized modifications and changes in s', b', d', then $p = p'$. The user assures that the software artifacts have been unmodified and the software supply chain integrity by comparing p and p'; if $p \neq p'$, then the breach is detected.

4.2. Authority Implementation and Offline vs. Online

For the authority implementation, we compare our scheme against the centralized server approach. In the centralized server approach, the authority providing the reference inputs for software build assurance is a centralized remote server and requires real-time networking between the server and the user, c.f., remote code attestation discussed in Section 7.

Our distributed scheme, in contrast, implements the authority using blockchain. We use blockchain to replace the centralized server and to provide the proof reference p'. The use of blockchain enables the user to forgo networking beyond its own node for software assurance verification. We describe the blockchain design and application for our assurance scheme to implement the distributed authority in greater detail in Section 4.5.

We define online vs. offline to better explain the overhead differences. The online operations occur at the time of the software download, while the offline operations can occur before the software download and build. The offline overheads and costs are lower than the online overheads because the offline activity can occur any time before the time of the software download and build. Our work specifically enables the delivery of the authority-reference inputs offline by using blockchain. Blockchain automatically synchronizes and delivers the reference information offline at the time when the software updates are available in the repositories, which is in advance of the online software update to the user.

4.3. Building Blocks: Hash Function, Merkle Tree, and Blockchain

The cryptographic hash function is a mathematical algorithm that turns input into a fixed-size string of characters, known as a hash value. It ensures data integrity by generating unique hash values for different inputs and exhibiting properties like collision and pre-image resistance, making it challenging for adversaries to tamper with or to reverse-engineer the data.

Merkle trees, building on the cryptographic hash functions themselves, utilize these properties to create a hierarchical structure. Building the tree starts with a set of leaf node values $x_1, x_2, \ldots, x_n$, where each node x_i would contain hash value $H(x_i)$. In the next level of the tree moving up would be the parent node that would contain the hash value of the concatenation of two of the children nodes, for example, $H_{12} = H(H(x_1)||H(x_2))$. The process will continue to build on the next level until a root hash is formed. This root hash represents the entire dataset concisely. A Merkle tree [12] based on a hash function is known to be efficient for verification and is popularly used in many computing applications, as described in Section 7.3.

Blockchain provides a cryptographic system with robust security properties using the likes of the hash function, the Merkle tree, and the public-key digital signature. It provides a digital ledger with high integrity where one can securely track who wrote the transaction on the ledger. Specifically, blockchain utilizes the public-key digital signature to provide authenticity, non-repudiation, and data integrity [13]. Such robust security properties enable the blockchain to be applied to high-risk security applications. One notable example is the cryptocurrency blockchains processing digital financial transactions. However, while we use similar cryptographic construction for the ledger security as the

cryptocurrency, our blockchain application environment having the software authorities (e.g., known software sources which are more reputable than the others) enables the use of a permissioned consensus protocol (e.g., as opposed to a permissionless protocol, such as that based on proof of work), providing significantly more efficient processing.

4.4. Proof Generations for p and p′ Using Merkle Tree for Setting Up Assurance

In Figure 4, we use ID, s, b, and d as the leaf nodes (which are the nodes on the bottom of the tree) highlighted in gold color. As the source file inputs (s) contain multiple files, we denote each individual file as F_i, where i indicates the source code file index. With the use of a one-way hash function, the Merkle tree computation propagates from bottom to top. The final Root computation yields the Root, which serves as the proof p. The same computation occurs from the authority side providing the reference but using ID, s', b', and d' to generate p', as described in Section 4.5. The Merkle tree is used in both offline blockchain and online user-side computing, which are described in greater detail in the following sections.

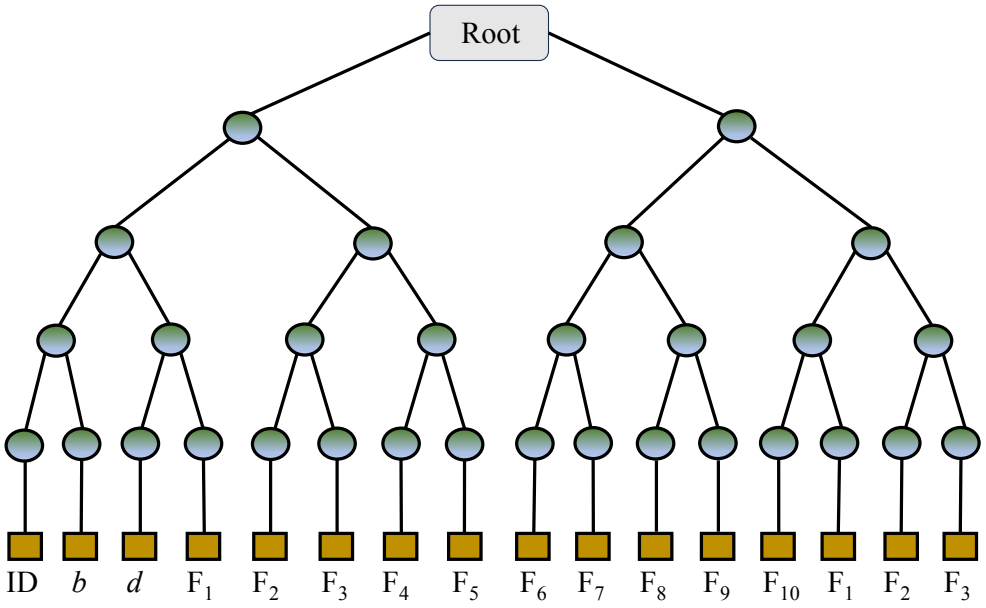

Figure 4. Merkle tree application for our scheme, where F_i are the source code s files and the Root is the proof p.

4.5. Blockchain for Authority: Reference Distribution in Offline and Access in Online

We use blockchain to serve as the authority to provide the reference information to enable the online user-side verification, in contrast to the centralized server approach implementing the logical authority on a physical remote server machine, as described in Section 4.2. In addition to having high payload integrity (cannot change or modify the payload content on the blockchain ledger) and high visibility and accountability (thanks to the distributed storage and easy read-access), we apply blockchain to implement the authority because blockchain supports automatic broadcasting and distributed storage to distribute the proof offline.

In our scheme, blockchain enables the reference proof to be distributed to the prospective downloading users offline (i.e., any time before the software download, build, and the build assurance verification). The networking and sharing of the proof reference information of p' (s', b', d' are used to compute p' but do not get networked) are offline so that they are stored within the user machine before the software download and build. In online, i.e., when downloading and building the software, the user simply accesses the blockchain database stored within its machine (computing only). In contrast, the centralized server approach involves the p' delivery and networking online (networking and computing). Because the networking and distribution of the reference information of p' are offloaded to before the software download/build, our scheme is significantly more efficient than the

centralized server approach. without the explicit networking with another node, as this information is already stored in the local database provided by the blockchain.

We build the authority using a permissioned blockchain, which controls the entities who can write on the blockchain and require registration. These entities generate the references s', b', d', p' and distribute them to write on the blockchain database, the database being locally stored and distributed to many other nodes. The entities can vary, including the software developers who write/commit/upload on the repositories or a third-party entity dedicated for software build assurance, in our scheme implementation. blockchain monitors and tracks the s', b', d', p' for accountability, so such entity control can be dynamic, i.e., an entity can be revoked. In our work, we focus on the scheme framework itself as opposed to making strong recommendations about the ecosystem when implementing our scheme framework, including who can generate the blockchain payloads and who has the write-access on the blockchain and the trade-off analyses (e.g., third-party entity outside of the developers themselves or the source-code repository can provide the separation, which can be useful in some software applications). While our scheme supports different ecosystem models and can support various entities who can contribute to writing/uploading s', b', d', p', in our concrete implementation instance in Section 5, we introduce a third-party entity to write and upload the s', b', d', p' on the blockchain data. Because of the permissioned blockchain, we use proof-of-authority for our consensus protocol, which is significantly more efficient and transaction-scalable (generating more transactions) than the permissionless blockchains, such as those for cryptocurrency (e.g., Bitcoin and Ethereum).

4.6. User-Side Verification (Online)

In online, i.e., during the software development, including the download of s and the build of b and d, the user computes and generates the proof p described in Section 4.4. In user-side computing, a user first downloads the ID and s of the target software from the repository and compiles s in their machine to obtain b and d. Then, it is involved in the proof generation process by constructing the Merkle tree using s, b, and d to calculate its proof, p, for the target software version. The proof verification process involves retrieving the respective ID and p' from the blockchain and comparing p with p'. If both p and p' match then it assures the user about the s, b, and d, i.e., all the software artifacts correspond to the authority-references and $s = s'$, $b = b'$, and $d = d'$.

5. Our Scheme Implementation to Assure the Ethereum Software Chain

While Section 4 describes our scheme using abstraction and the variables which are generally applicable across software projects and cryptographic function choices, we provide an implementation instance of our scheme in this section to provide a proof-of-concept and provide concrete analyses. Our implementation consists of the generation of proof on both the blockchain side and the user side. In this section, we focus on the user-side, computing p using s, b, d, but the same applies to the authority (blockchain in our scheme), with p' depending on s', b', d' and thus omits the prime variables on the authority.

5.1. Authority Implementation For Our Scheme (Blockchain) vs. Centralized Server Approach (Remote Server)

Our scheme uses blockchain to distribute the authority-reference information offline, and the user can use this reference information to check the software artifacts of s, b, d online at the time of software download and build, as described in Section 4. The use of blockchain enables the user to access and retrieve the relevant reference information within its device, as blockchain provides distributed storage on the user. In contrast, the centralized server approach requires the user to communicate with the remote server authority to retrieve the reference information.

To demonstrate the efficiency of our scheme's proof verification enabled by the blockchain use, we implement the centralized server approach described in Section 4.2 and implement a remote server as the centralized authority to provide the references for software build assurance. We implement the remote servers in various locations and, more specifically, in four distinct locations in the world (two in the United States and two overseas) through the Google Cloud Platform. More specifically, the remote servers are located

in "Iowa", "California", "Singapore", and "London". We use these geographical locations to label and identify the distinct remote authority servers for the centralized authority.

5.2. Target Software for *s*

Our target software is the Ethereum cryptocurrency software. More specifically, the source code *s* is the latest Ethereum software version, Archanes (v1.13.4). Our work is generally applicable to any software project in principle. However, we chose the Ethereum cryptocurrency software because it is highly relevant to our work. Ethereum cryptocurrency is highly relevant because of its high-security risk. Ethereum cryptocurrency is developed by a collective group of developers, making it highly vulnerable to software compromise. Cryptocurrency is also very popular with its operations to process transactions (the market capitalization is in tens of billions of dollars), and thus compromising the integrity of the software has a high breach impact.

5.3. Building Blocks for Our Scheme for *b* and *d*

5.3.1. *b* Using Reproducible Build

The current Ethereum builds and supports reproducible build in popular OSs, such as Linux, Windows, and MacOS [11]. In our implementation, we focus on the reproducible build for the Linux 64-bit processor. The reproducible build output *b* (which becomes one of the inputs to our proof generation in our scheme) is the compilation output, which is the geth (Archanes (v1.13.4)) running the Ethereum node and wallet. This *b* is stored in the official Ethereum repository.

5.3.2. *d* Using SBOM

We use an open-source SBOM generator tool called Syft [14] for our implementation. Using Syft, the SBOM includes information on geth for Archanes (v1.13.4), 91 packages as dependencies, and its relationships (total 181) between packages. The generated SBOM will then be used as *d* in our implementation. We chose Syft as it can convert between SBOM formats easily and works well with another tool called 'Grype', which is a vulnerability scanner that can scan for any common vulnerability and exposure (CVE). This would be particularly helpful to incorporate in a future proof of concept.

5.4. Proof Generation Using Merkle Tree and Hash

Based on the overview of the Merkle tree in Section 4.3 and Figure 4, we build Proof generation with the Merkle tree construction using the open-source implementation of Merkle tree [15] and Proof verification functionalities using C++, OpenSSL SHA-256 hash implementation and recursive directory iterator, and a regular expression to retrieve all relevant files of the Ethereum latest version, Archanes (v1.13.4) (excluding dynamic user-specific and configuration files) associated with C, C++, and shell. The ID, *s*, *b*, and *d* of Archanes (v1.13.4) are the inputs of the Merkle tree for proof generation. We have a total of 1350 inputs of the Merkle tree with *s* containing 1347 inputs and 1 input each from ID, *b*, and *d*. However, we use cyclic extension of these files till 2048 inputs as discussed in [9] to make it a balanced Merkle tree with the number of leaf nodes equal to the power of two.

5.5. Blockchain for Reference Proof Delivery

We build a smart contract using Solidity v0.8.21 to upload the ID and its respective *p* (as a form of key-value pair) on the local private Ethereum blockchain. As a proof-of-concept (PoC) implementation, we do not upload ID and *p* in the Ethereum Mainnet to avoid transaction fees. Our smart contract design enables the maintainers of the smart contract to add new nodes with write permission. We use two mappings in our smart contract: authorization mapping (maps the address to a Boolean value specifying the authorization of the address to modify the ledger) and key-value pair mapping (maps the Proof with the respective ID). It can also verify whether a node has the write privilege using the authorization map before granting permission to upload an ID with the *p* on the blockchain. A user can retrieve *p* for the respective ID from the blockchain for proof verification.

5.6. Hardware and Experimental Setup

Our experiment consisted of two main pieces of hardware that we used for user and remote servers. The remote servers are only used in the centralized server approach. For user hardware, we use Intel Xeon CPU E5-1650 v2 with 32GB RAM. We use the Google Cloud Platform (GCP) for our remote server that has 2 vCPU with 8GB RAM. While the power of a processor (CPU) is important, the memory specification on the random access memory (RAM) provides information on the capability of a computer to access and process data. We also implemented an in-memory data store using the Redis open-source database [16] for higher efficiency in data storage access.

6. Experimental Analyses

We implement our scheme and empirically verify the correctness of our scheme in implementation, i.e., correctly detect the modifications of the software artifact inputs to our scheme in Section 6.1. After verifying that our scheme correctly detects the changes and modifications of the software artifacts, we analyze the performances. Our performance measurements and analyses include the online user-side computing for proof generation and verification (compared with the software download/build latencies) in Section 6.2. For our performance analyses, we measure latency, storage/memory, and processing. Furthermore, to highlight the efficiency of the authority delivering the references of s', b', d', p', we implement and compare the performances of our blockchain-based scheme vs. the approach based on a centralized server.

6.1. Verification Correctness

We observe that a change in s, b or d changes the proof p so that it becomes $p \neq p'$ due to the pseudo-random hash functions used in our implementation. More specifically, in our implementation, omission of a file, addition of a new file, and even a single bit flip in s, b or d changes p so that $p \neq p'$, failing the integrity check and verification, and detecting the changes in the software supply chain in s, b or d.

6.2. Computing Performances

Computational Latency: A user needs to perform leaf node construction (hashing all the files of the Ethereum software, compiler output, and SBOM document) and then finally generate the non-leaf nodes up to the Merkle root to generate a proof using the Merkle tree in Figure 4. The leaf-node construction corresponds to the bottom lines for the hash computations of the leaf nodes (ID, b, d, and the source files F) in Figure 4, while the non-leaf node construction takes the preceding hash outputs as the inputs to progress upward. The leaf-node construction is the dominant factor of the computing overhead because of the longer sizes and the processing for ID, b, d, and the source files F. In our measurements, the leaf nodes construction and the non-leaf nodes construction take 1082 ms or 1.082 s and 14.48 ms, respectively, resulting in a proof generation latency of 1096 ms or 1.096 s. On the other hand, the proof verification takes only 0.4254 ms. Therefore, the computing overhead including the proof generation and verification is 1097 ms or 1.097 s for the software build assurance of the Ethereum software artifacts. To compare, the Ethereum software download takes 157 s and 371 s for compilation, which are two orders of magnitude greater than our online scheme overhead of 1.097 s. Both the software download/build and the user-side proof generation and verification occur online.

Storage, Memory, and Processing: Though each Ethereum software version can have different b values (thus different p) based on the operating systems and processors, a user needs to store the Merkle tree of only the Ethereum software version it is using. For the latest Ethereum software version (Archanes (v1.13.4)), the Merkle tree size is 197.42 kilobytes with a total of 1972 files, including the SBOM document and compiler output. Moreover, in our experiment, it takes only 3–5% RAM and 1–2% CPU usage for Proof generation and verification.

6.3. Authority Delivering References: Our Scheme vs. Centralized Server Approach

As described in Section 4.5, our scheme utilizing blockchain enables the networking and distribution of the reference information of p' offline, in contrast to the centralized server approach which relies on a server and the online networking and sharing of p'. The proof verification process online (more specifically, after the software download/build and the proof generation for p) operates very differently between our scheme vs. the centralized server approach. In this section, we focus on the online software build assurance verification process. We also analyze how the efficiency of the verification affects the bandwidth of the assurance operations (the number/rate of assurances supported by our scheme vs. the centralized server approach).

Online Proof Verification For Our Scheme vs. Centralized Server Approach: Because our scheme only involves computing while the centralized server approach involves both networking and computing (computing and verification based on the communication receiving results of p'), our scheme is significantly more efficient in the online proof verification operation. Figure 5 shows the empirical measurement distribution in CDF and identifies the average values in dotted vertical lines. The proof verification takes 0.4254 ms, while the centralized server approach takes at least 91.42 times more, depending on the locations. Among the distinct remote servers serving as the authority to our scheme, the Iowa server is the quickest at 38.89 ms (91.42 times slower than our use of blockchain for the authority) and the London server is 370.9 (871.9 times slower). In general, the remote servers in the US nation (the user is also in the US) perform much better than the ones outside, because they go through a smaller number of distinct internet service providers (ISPs). In our experiments, our scheme based on the blockchain-authority performs two to three orders of magnitude quicker than having a centralized, remote authority.

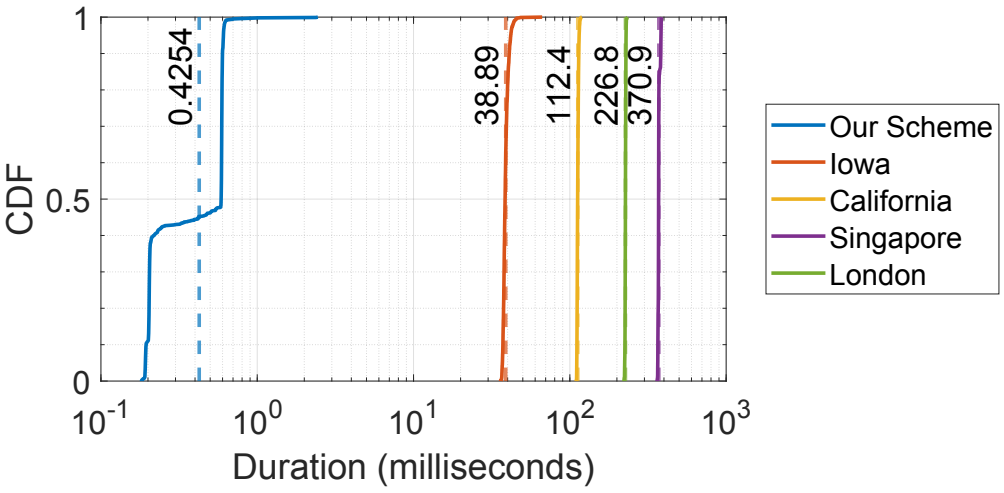

Figure 5. The proof verification time between our scheme using distributed blockchain vs. centralized remote servers.

Assurance Bandwidth For Our Scheme vs. Centralized Server Approach: Using blockchain as the authority for our scheme as opposed to the centralized server approach also provides significant improvement in the rate support. We define the bandwidth here to be the number of assurance resolutions that can be supported using our scheme. We experimented using 1000 requests per second and observed how many of those requests were resolved, including the final verification. As shown in Figure 6, if using the blockchain as the authority, our scheme can support 905.9 assurance resolutions per second. If using a remote centralized server as the authority for our scheme, the bandwidth is limited to between 17.90 (London with the worst performance) and 63.78 (Iowa with the best performance). Comparing between the distinct remote-server simulations, similarly to the above latency measurements, the remote servers in the US perform much better than the authority servers outside.

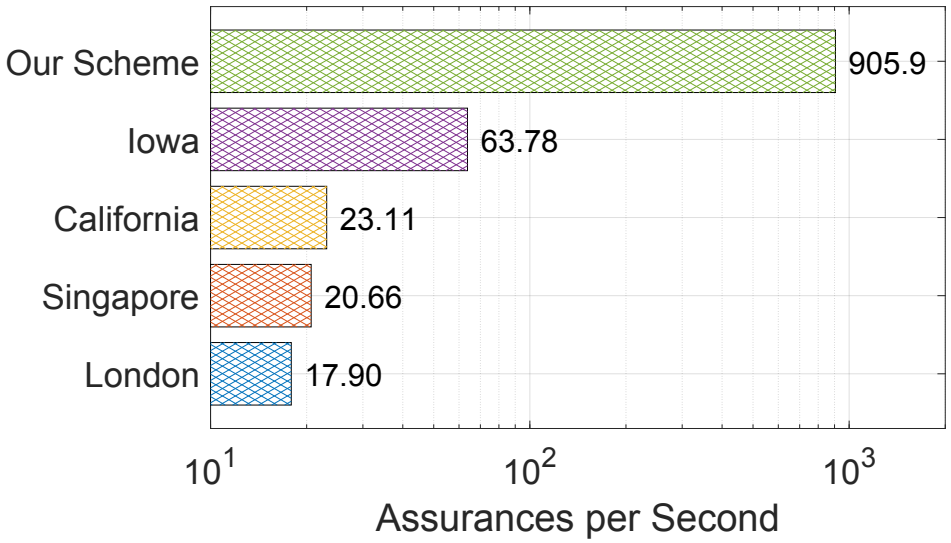

Figure 6. Bandwidth performance in assurance resolutions per second in comparison between our scheme and remote server locations.

6.4. Supports Software Assurance Requirements

Our scheme is applied for the software build assurance. New software versions, which are the new inputs of our software assurances, only occur when the software is updated. For Ethereum cryptocurrency, which is our target software for assurance, there have been 22 version updates since 2014 [17], so 2.2 version updates per year. The current Ethereum stable build [11] supports three different operating systems of Linux, Mac OS, and Windows, so 6.6 updates per year. Our scheme bandwidth performance of 905.9 assurance resolutions per second in Section 6.3 is enough to support our software build assurance application for Ethereum, as a 6.6 assurance resolutions *per year* requirement is significantly smaller than the 905.9 assurance resolutions per second our scheme supports.

The blockchain ledger also grows according to the software build assurance requirement to support new software version updates. The 6.6 version updates per year requirement results in the blockchain ledger growing 1.3030 kB = 6.6 $\times$ 197.42 kB per year if storing the entire Merkle tree nodes, where 197.42 kB is for each software build version as empirically analyzed in Section 6.2. This blockchain storage requirement is significantly less than most applications, such as the global financial transaction processing in cryptocurrency, whose ledger sizes are typically of the order of tens of GB, and, for more popular cryptocurrencies, such as Bitcoin and Ethereum, of the order of hundreds of GB. As the ledger's storage requirements remain minimal, even with multiple updates, the overall storage overhead is negligible, given the relatively small size of each update.

7. Related Work

7.1. Remote Code Attestation

Our work is related to remote code attestation that verifies the integrity of the running code on the users' system, adopting several hardware-based, software-based, and hybrid techniques. Hardware-based attestation techniques [18–20] involve hardware components, e.g., trusted platform module (TPM), software guard extensions (SGX), and software-based attestation techniques [21–24], that involve a challenge and response protocol in real-time, time synchronization, empty memory space filling, etc. An advancement of these techniques, hybrid attestation techniques [25–27], integrates both hardware-based and software-based techniques, minimizing the hardware cost with embedded processing components, e.g., field-programmable gate array (FPGA), memory protection unit (MPU) and micro-controller unit (MCU), etc. Our scheme goes beyond the research in remote code attestation in two ways: first, our scheme verifies the integrity of a software in different stages of the supply chain; second, our scheme does not rely on the centralized authority by using a distributed blockchain.

7.2. Reproducible Build

Reproducible build yields a deterministic build which achieves reproducibility when, with identical source code, build environment, and build instructions, any entity can recreate precise, bit-for-bit copies of all specified artifacts. [28] Projects and development efforts, such as Debian packages [29], Gitian [30], and other Linux distribution [31,32], focus on the reproducibility and verifiable build where there will be involvement using a virtual environment that would fix non-deterministic variables. Some other methods also include removing embedded timestamps from binary and making changes to the CFLAGS variable to produce a deterministic output [33]. It is also worth mentioning that [34] suggested that having a deterministic build for close-source software proves to be difficult due to lack of documentation and many other compilation processes that are susceptible to non-deterministic output [35]. Our work assumes, builds on, and uses the reproducible build technology.

7.3. Hash and Merkle Tree Applications for Securing Integrity

Our work builds on the Merkle tree and hash function to construct our scheme for software integrity. Previous research has utilized the Merkle tree and hash function for integrity in other applications and purposes than ours, including in over-the-air vehicular software updates [36,37], vehicular firmware updates [38,39], transactions, blocks, and the consensus mechanism in blockchain [40,41], cryptocurrency and cellular networking software assurance [9,10]. Version control systems (VCSs) used for project collaborations include Git [42], Subversion [43], and Mercurial [44]. Our work uses the Merkle tree to achieve different goals than these previous works, i.e., we use it to assure software integrity.

7.4. Blockchain for Security

Blockchain provides security for many types of systems, e.g., identity management [45], electronic voting [46], decentralized cryptocurrencies [47], supply chain management [48], healthcare [49], public key infrastructure (PKI) [50,51], cellular networking [52], the Internet of Things (IoT) [53], and vehicular networking [54], etc. The use of blockchain in these systems mitigates several types of security risks and attacks, e.g., single point of failure, data tampering, unauthorized access control, users' identity and data breach, distributed denial-of-service (DDoS) attacks, Sybil attacks, replay attacks, injection attacks, and man-in-the-middle (MITM) attacks, etc. with its properties such as decentralization, anonymity, immutability, transparency, and automatic synchronization, etc. In our scheme, we use these properties of blockchain to preserve the software supply chain integrity, which has a different goal from that of the previous research.

8. Conclusions

In this paper, we design and build a software build assurance scheme assuming reproducible build (for deterministic build output) and SBOM (for documentation, recording, and tracking of the software development). Our software build assurance scheme checks the integrity against (unauthorized or unintentional) changes and modifications in multiple software artifacts generated in the software supply chain. More specifically, it checks the integrity of the software artifacts of the source code (s or F, the latter of which corresponds to the source code in multiple files), build or compilation output (b), and the documentation or record (d). Computing on these inputs and using cryptographic hash and Merkle tree yields the proof p in our scheme. We further use blockchain to serve as the authority providing the reference information for our scheme; this information provides the comparison reference and corresponds to what the software artifacts should be. In addition to the blockchain providing high integrity and control on the write-access, the distributed storage and automatic broadcasting strengths of the blockchain provide significant performance advancements in the online software build assurance verification processes by offloading the reference delivery to offline before the time of the software download/build.

In addition to describing our scheme using abstraction and variables, we empirically implement our scheme targeting the Ethereum software as the object of assurance to show the correctness of our scheme (i.e., changes are detected). Our empirical performance

analyses based on our implementation also show that our scheme is appropriate and provides small and manageable costs in the software download/build process, e.g., our scheme's online overhead is two orders of magnitude smaller than both the download and build/compilation duration. Furthermore, the blockchain-based authority is also two to three orders of magnitude quicker than the authority based on a centralized remote server.

Author Contributions: Conceptualization, K.L. and S.-Y.C.; Methodology, K.L. and S.-Y.C.; Software, K.L.; Validation, K.L. and S.-Y.C.; Formal analysis, K.L, A.S. (Arijet Sarker), S.W. (Simeon Wuthier); Investigation, K.L. and S.-Y.C.; Resources, J.K. (Jinoh Kim), J.K. (Jonghyun Kim) and S.-Y.C.; Data curation, S.W.; Writing—original draft, K.L. and S.-Y.C.; Writing—review & editing, K.L., A.S. (Arijet Sarker), S.W. (Simeon Wuthier), J.K. (Jinoh Kim), J.K. (Jonghyun Kim) and S.-Y.C.; Visualization, K.L. and S.W. (Simeon Wuthier); Supervision, J.K. (Jinoh 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 in part by the National Science Foundation under Grant No. 1922410 (50%) and by an Institute of Information and Communications Technology Planning and Evaluation (IITP) grant funded by the Korean government (MSIT) (No. 2021-0-02107, with collaborative research on element Technologies for 6G Security-by-Design and standardization-based International cooperation, 50%).

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

Conflicts of Interest: The authors declare no conflicts 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. The United States Government. The White House. 2023. Available online: https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/ (accessed on 29 October 2023).
2. SolarWinds Supply Chain Attack | Fortinet. Available online: https://www.fortinet.com/resources/cyberglossary/solarwinds-cyber-attack (accessed on 29 September 2024).
3. Advanced Persistent Threat Compromise of Government Agencies, Critical Infrastructure, and Private Sector Organizations | CISA. Available online: https://www.cisa.gov/news-events/cybersecurity-advisories/aa20-352a. (accessed on 29 September 2024).
4. Wu, Q.; Lu, K. On the Feasibility of Stealthily Introducing Vulnerabilities in Open-Source Software via Hypocrite Commits 2021 Available online: https://api.semanticscholar.org/CorpusID:233479632 (accessed on 11 August 2024).
5. Meli, M.; McNiece, M.R.; Reaves, B. How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories. In Proceedings of the Network and Distributed System Security Symposium 2019, San Diego, CA, USA, 24–27 February 2019. [CrossRef]
6. Liu, C.; Chen, S.; Fan, L.; Chen, B.; Liu, Y.; Peng, X. Demystifying the vulnerability propagation and its evolution via dependency trees in the NPM ecosystem. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022. [CrossRef]
7. Neupane, S.; Holmes, G.; Wyss, E.; Davidson, D.; Carli, L.D. Beyond Typosquatting: An In-depth Look at Package Confusion. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA, 9–11 August 2023; pp. 3439–3456.
8. Cybersecurity and Infrastructure Security Agency (CISA). Types of Software Bill of Material (SBOM) Documents. Available online: https://www.cisa.gov/sites/default/files/2023-04/sbom-types-document-508c.pdf (accessed on 11 August 2024).
9. 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), IEEE, Las Vegas, NV, USA, 8–11 January 2023; pp. 804–809.
10. 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. [CrossRef]
11. The Go-Ethereum Authors. Stable Releases. 2023. Available online: https://geth.ethereum.org/downloads (accessed on 29 October 2023).
12. Merkle, R.C. A digital signature based on a conventional encryption function. In *Advances in Cryptology—CRYPTO '87*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1988; pp. 369–378. [CrossRef]
13. Bitcoin.org. Available online: https://bitcoin.org/bitcoin.pdf (accessed on 1 October 2024).
14. GitHub—Anchore/Syft: CLI Tool and Library for Generating a Software Bill of Materials from Container Images and Filesystems. Available online: https://github.com/anchore/syft. (accessed on 10 November 2023).
15. Microsoft. Microsoft/Merklecpp: A C++ Library for Creation and Manipulation of Merkle Trees. 2023. Available online: https://github.com/microsoft/merklecpp (accessed on 20 October 2023).

16. Redis. Available online: https://redis.io/ (accessed on 29 January 2024).
17. Ethereum.org. The History of Ethereum. 2024. Available online: https://ethereum.org/en/history/ (accessed on 6 August 2024).
18. Trusted Computing Group. Trusted Platform Module (TPM). 2023. Available online: https://trustedcomputinggroup.org/work-groups/trusted-platform-module/ (accessed on 26 May 2023).
19. 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).
20. ARM. Layered Security for Your Next SoC. 2023. Available online: https://www.arm.com/products/silicon-ip-security (accessed on 26 May 2023).
21. 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.
22. 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, IEEE, San Francisco, CA USA, 20–23 May 2012; pp. 239–253.
23. 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]
24. 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.
25. Nunes, I.D.O.; Eldefrawy, K.; Rattanavipanon, 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.
26. 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.
27. 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.
28. Reproducible Builds. A Set of Software Development Practices That Create an Independently-Verifiable Path from Source to Binary Code. Available online: https://reproducible-builds.org/ (accessed on 10 November 2023).
29. ReproducibleBuilds—Debian Wiki. Available online: https://wiki.debian.org/ReproducibleBuilds (accessed on 10 November 2023).
30. GitHub—Devrandom/Gitian-Builder: Build Packages in a Secure Deterministic Fashion Inside a VM. Available online: https://github.com/devrandom/gitian-builder (accessed on 10 November 2023).
31. Build Result Compare Script. Available online: https://build.opensuse.org/package/show/openSUSE:Factory/build-compare (accessed on 10 November 2023).
32. GitHub—Kholia/ReproducibleBuilds: Reproducible Builds in Fedora ("Remock"). Updated for Fedora 23, and Rawhide. Available online: https://github.com/kholia/ReproducibleBuilds (accessed on 10 November 2023).
33. FOSDEM 2014—Reproducible Builds for Debian. Available online: https://archive.fosdem.org/2014/schedule/event/reproducibledebian/ (accessed on 10 November 2023).
34. de Carné de Carnavalet, X.; Mannan, M. Challenges and implications of verifiable builds for security-critical open-source software. In Proceedings of the 30th Annual Computer Security Applications Conference, ACM, New Orleans, LA, USA, 8–12 December 2014. [CrossRef]
35. 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 2023 IEEE Symposium on Security and Privacy (SP), IEEE, San Francisco, CA, USA, 22–25 May 2023. [CrossRef]
36. Bazzi, A.; Shaout, A.; Ma, D. MT-SOTA: A Merkle-Tree-Based Approach for Secure Software Updates over the Air in Automotive Systems. *Appl. Sci.* **2023**, *13*, 9397. [CrossRef]
37. Ghosal, A.; Halder, S.; Conti, M. STRIDE: Scalable and secure over-the-air software update scheme for autonomous vehicles. In Proceedings of the ICC 2020-2020 IEEE International Conference on Communications (ICC), IEEE, Dublin, Ireland, 7–11 June 2020; pp. 1–6.
38. Nilsson, D.K.; Sun, L.; Nakajima, T. A framework for self-verification of firmware updates over the air in vehicle ECUs. In Proceedings of the 2008 IEEE Globecom Workshops, IEEE, New Orleans, LA, USA, 30 November–4 December 2008; pp. 1–5.
39. Ghosal, A.; Halder, S.; Conti, M. Secure over-the-air software update for connected vehicles. *Comput. Netw.* **2022**, *218*, 109394. [CrossRef]
40. Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Bus. Rev.* **2008**. [CrossRef]
41. Buterin, V. A next-generation smart contract and decentralized application platform. *White Paper* **2014**, *3*, 2-1.
42. Git. Git—Distributed-Even-If-Your-Workflow-Isnt. 2023. Available online: https://git-scm.com/ (accessed on 10 January 2023).
43. Apache Subversion. Apache Subversion. 2023. Available online: https://subversion.apache.org/ (accessed on 10 January 2023).
44. Mercurial. Work Easier Work Faster. 2023. Available online: https://www.mercurial-scm.org/ (accessed on 10 January 2023).
45. Liu, Y.; He, D.; Obaidat, M.S.; Kumar, N.; Khan, M.K.; Choo, K.K.R. Blockchain-based identity management systems: A review. *J. Netw. Comput. Appl.* **2020**, *166*, 102731. [CrossRef]

46. Benabdallah, A.; Audras, A.; Coudert, L.; El Madhoun, N.; Badra, M. Analysis of blockchain solutions for E-voting: A systematic literature review. *IEEE Access* **2022**, *10*, 70746–70759. [CrossRef]
47. Ghosh, A.; Gupta, S.; Dua, A.; Kumar, N. Security of Cryptocurrencies in blockchain technology: State-of-art, challenges and future prospects. *J. Netw. Comput. Appl.* **2020**, *163*, 102635. [CrossRef]
48. Lim, M.K.; Li, Y.; Wang, C.; Tseng, M.L. A literature review of blockchain technology applications in supply chains: A comprehensive analysis of themes, methodologies and industries. *Comput. Ind. Eng.* **2021**, *154*, 107133. [CrossRef]
49. Villarreal, E.R.D.; García-Alonso, J.; Moguel, E.; Alegría, J.A.H. Blockchain for healthcare management systems: A survey on interoperability and security. *IEEE Access* **2023**, *11*, 5629–5652. [CrossRef]
50. Matsumoto, S.; Reischuk, R.M. IKP: Turning a PKI around with decentralized automated incentives. In Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP), IEEE, San Jose, CA, USA, 22–26 May 2017; pp. 410–426.
51. Sarker, A.; Byun, S.; Fan, W.; Chang, S.Y. Blockchain-based root of trust management in security credential management system for vehicular communications. In Proceedings of the 36th Annual ACM Symposium on Applied Computing, Virtual, 22–26 March 2021; pp. 223–231.
52. Nguyen, D.C.; Pathirana, P.N.; Ding, M.; Seneviratne, A. Blockchain for 5G and beyond networks: A state of the art survey. *J. Netw. Comput. Appl.* **2020**, *166*, 102693. [CrossRef]
53. Huo, R.; Zeng, S.; Wang, Z.; Shang, J.; Chen, W.; Huang, T.; Wang, S.; Yu, F.R.; Liu, Y. A comprehensive survey on blockchain in industrial internet of things: Motivations, research progresses, and future challenges. *IEEE Commun. Surv. Tutor.* **2022**, *24*, 88–122. [CrossRef]
54. Alladi, T.; Chamola, V.; Sahu, N.; Venkatesh, V.; Goyal, A.; Guizani, M. A comprehensive survey on the applications of blockchain for securing vehicular networks. *IEEE Commun. Surv. Tutor.* **2022**, *24*, 1212–1239. [CrossRef]