Received 29 April 2022, accepted 13 June 2022. Date of publication 00 xxxx 0000, date of current version 00 xxxx 0000.

Digital Object Identifier 10.1109/ACCESS.2022.3185046

Decentralized Review and Attestation of Software Attribute Claims

OLIVER STENGELE, CHRISTINA WESTERMEYER, AND HANNES HARTENSTEIN, (Member, IEEE)
Institute of Information Security and Dependability (KASTEL), Karlsruhe Institute of Technology (KIT), 76131 Karlsruhe, Germany

Corresponding author: Oliver Stengele (oliver.stengele@kit.edu)

The work of Oliver Stengele was supported in part by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF), and in part by the Competence Center for Applied Security Technology (KASTEL) Security Research Labs. The work of Christina Westermeyer was supported by the German Federal Ministry for Economic Affairs and Climate Action within the Project Software-Defined Car (SofDCar) according to a Decision of the German Federal Parliament under grant 19S21002K. This work was supported by the KIT Publication Fund of the Karlsruhe Institute of Technology.

ABSTRACT Software can be described, like human users and other objects, through attributes. For this work, we define software attributes as humanly verifiable, falsifiable, or judgeable statements regarding characteristics of said software. Much like attributes in general, software attributes require robust identities for their source but also for their target, meaning a software in general or a binary in particular. As software can be of critical importance, performing an independent review of attribute claims appears beneficial. We posit that decentralized platforms that were developed and refined over the past decade can bridge the gap between existing tools and methods for software review and their open, transparent, and accountable use for the benefit of users. In this work, we explore the feasibility and implications of decentralizing an independent review of software attribute claims. We envision the decentralization of a review process from initialization and execution to the persistent recording of results. We sketch the available design space by decomposing the overall process into a modular design and describe how each component covers overarching objectives. To illustrate practical implications and tradeoffs, we present ETHDPR, a proof of concept implementation based on Ethereum and IPFS. Through a quantitative and qualitative evaluation, we show that a decentralized software review is practically feasible. We illustrate the flexibility of the proposed approach using a toy example of a software component in automotive systems. Lastly, we provide a discussion on fundamental limits and open issues of facilitating independent reviews via technological means.

INDEX TERMS Decentralized systems, software attributes, software certification, software identity management, software review, software transparency.

I. INTRODUCTION

Software holds a position of ever increasing importance in modern society, both for individuals as well as governments and businesses. However, the rigor with which software is created, managed, and evaluated prior to its wide-spread use still leaves room for improvement, as frequent incidents[1] show. Kwan *et al.* [1] recently explored how transparency in the development process of software can help to foster trust among users and other relying parties with the long-term goal of "Socially Responsible Software". In a similar way, works by Denney and Fischer [2] and Heck *et al.* [3] argue for the usefulness and importance of software audits and subsequent certification. Indeed, in some open-source software projects like Apache, structured review processes among developers are employed to great effect [4].

We posit the existence of a gap between software review, attestation, or certification concepts like the ones cited above and their wide-spread practical adoption that public decentralized platforms could help to overcome, particularly for open-source software projects. Notice that the example of Apache mentioned above deliberately conducts communication via mailing lists,[2] i.e. one of the oldest decentralized communication infrastructures of the Internet. Decentralized platforms are built and maintained through a communal contribution of resources and secured through mutual

The associate editor coordinating the review of this manuscript and approving it for publication was Thanh Ngoc Dinh.

[1]Most recently, the Log4Shell vulnerability: https://nvd.nist.gov/vuln/detail/CVE-2021-44228

[2]https://www.apache.org/foundation/how-it-works.html#communication

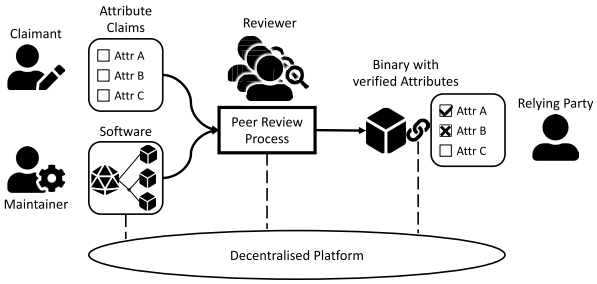

FIGURE 1. Software attributes are statements about various characteristics of a specific software that are objectively verifiable/falsifiable through methods and tools or require a subjective assessment by a human. Our overall goal is to initiate, conduct, and document the results of a review process for such attributes via decentralized platforms, such that relying parties can obtain both software binaries and attestation of verified attributes.

vigilance. Consequently, they can serve as a neutral but reliable medium for communication and long-term archive independent of any single developer community. Through their high availability and tamper-resistance, decentralized platforms have the potential to foster an easily accessible ecosystem of transparent and accountable exchange.

In this paper, we envision the decentralization of a general software review process for attribute claims from initialization to execution and the persistent recording of results, as depicted in Fig. 1. *Attribute claims* are human-readable statements regarding the characteristics of a software that are either objectively verifiable or falsifiable, or require a subjective assessment. Consequently, a *software attribute* is a statement by a reviewer whether or not, or to which extent, an attribute claim about a particular software holds as the result of their individual review. Note that we use the term *review* to mean both the general process of examining software for the purpose of judging an attribute claim as well as the result of said process, which can consist of multiple attributes. Establishing such software attributes properly, therefore, requires robust identities not only for the reviewers but for the attributed software as well. Lastly, in order to facilitate an independent generation of attributes, i.e. that no attributing party is privy to the assessment of other parties prior to committing to their own assessment, a coordinated disclosure of these attributes is necessary as well. We note here that such independent work can only be encouraged and enabled through technical means but not strictly enforced, a point we elaborate in Section VII.

We describe the four roles depicted in Fig. 1 as follows: A *maintainer* manages a particular software, including its identity representation, by developing new releases and attaching or detaching them to said software identity as needed. *Claimants* posit specific attribute claims of a particular software release to initiate and direct an independent review with subsequent attributions. *Reviewers* participate in such a review process and attest or refute posited claims of a particular software release as part of their independently generated attributions. *Relying parties* of a software under review are interested in the results of a review process, for example, to make an informed decision about using or building upon the attributed software. Relying parties could be end users who want to have independent information on the software they are using, customers who want to ensure the deployed software fulfills specific requirements or other interested parties that may be affected by the software or are representatives of those parties. Lastly, we need an entity that governs the execution of a review process, more specifically the aforementioned coordinated disclosure of attributes for the purpose of independence. Note that none of the above roles are mutually exclusive. We term the union of all listed roles as stakeholders of a software under review.

In this paper, we build upon insights from our previous works [5]–[7] to give a comprehensive overview from problem statement over a conceptual design to a concrete implementation and to provide a concise basis for future work. More precisely, our contributions are as follows:

- Conceptual design of a decentralized software review process to provide an overview and a modular decomposition in order to illustrate the design space.
- ETHDPR, a modular, extensible framework enabling a decentralized public review of software attribute claims via Ethereum, a public blockchain.
- Quantitative evaluation of our implementation of ETHDPR's on- and off-chain operations; and a qualitative evaluation of our implementation's capabilities and limitations.
- Discussion on fundamental limits, practical implications, and open challenges of a decentralized software review process.

The remainder of this paper is structured as follows. We begin by discussing related work in Section II. In Section III, we introduce and examine the general problem of reviewing software attribute claims and posit objectives that a practical implementation should aim to achieve. With Section IV, we give a conceptual design by decomposing the general problem into distinct but interconnected modules and by describing how each of them can address the desired objectives from above. To illustrate the practical implications of decentralizing a software review, we present ETHDPR, a modular and extensible framework based on Ethereum smart contracts in Section V. We then provide a quantitative and qualitative evaluation of our implementation of ETHDPR in Section VI to highlight the tradeoffs involved with the decentralization of such a review process. In Section VII, we offer a discussion of points both related to the general problem at large as well as our proof of concept implementation, and we also provide directions for future work. With Section VIII, we conclude the paper.

II. RELATED WORK

Software review as a process for quality control is well documented in the literature [4], [8]. A similar but distinct practice is the certification of software or its components [2], [3], [9] for the purpose of quality attestation, which is closer to the problem statement of this work. While these works

deal with *what* a software review entails, the work presented here examines *how* and to which extent such tasks can be accomplished on decentralized platforms.

Instead of relying on third parties to examine a software regarding its characteristics, proof-carrying code [10], [11] tasks creators of software with attaching safety proofs to binaries which can be evaluated on end-user systems before execution. We view the concept presented in this work as complimentary to proof-carrying code in the following way: Our approach allows more general software characteristics to be attested but in exchange, the certainty and veracity of each individual attestation can be lower. While proof-carrying code is intended to be highly automated, we deliberately allow the inclusion of human decisions as part of the review process.

Crowdsourced software testing [12] transfers testing activities to the crowd, e.g. to access a diverse pool of testers and test environments to evaluate more software variants. Pastore *et al.* [13] present a crowdsourcing approach to automate the generation of test oracles. Yan *et al.* [14] propose a platform for crowdsourced testing of web and mobile applications. Decentralized testing approaches [15], [16] are also discussed in the related research field of collaborative software testing. Architectures for decentralized, blockchain-based malware detection are proposed by Homayoun *et al.* [17] and Hu *et al.* [18]. Chen *et al.* [19] discuss a decentralized software auditing approach similar to ours but restricted to data integrity auditing. To our knowledge, previous work therefore either concentrates on distributing test activities or on decentralizing specific tasks, whereas our approach aims at decentralizing arbitrary, independent software examinations.

Within the context of distributed ledgers, data oracles [20], [21], some of which also rely on crowdsourcing, are an area of active research. A crucial difference between the problem explored in this paper and blockchain oracles is who the supplied data is used by. Generally, blockchain oracles are currently built to supply data for smart contracts to drive logic dependent on real-world data whereas the software attestations solicited in this work are primarily meant to be used by human actors to inform their usage decisions regarding a certain software. Still, there may be valuable solutions in works on oracles that could be transferred to the problem space of software attestation.

Two prominent works in the field of decentralized systems to realize or support software distribution processes are Chainiac by Nikitin *et al.* [22] and SmartWitness by Guarnizo *et al.* [23]. While Chainiac is built from the ground up as a permissioned, decentralized system, SmartWitness makes use of the Ethereum blockchain and smart contract functionality similar to our approach. SmartWitness is particularly noteworthy in relation to our work as it introduced the ability for accredited security providers to attach a ''security score'' to software releases, albeit without any mechanism to ensure independent assessments by multiple such providers. More recently, Ince *et al.* [24] presented a blockchain-based package management system, which may also be extended with independent review functionality as we describe it in this work.

III. PROBLEM STATEMENT

We begin by defining the objectives and non-objectives of the problem at hand.

A. OBJECTIVES

To explicitly describe the problem of conducting an independent software review process, we divide it into smaller objectives. First, we identify several objectives regarding the overall review process:

- **O1** Reviews should be created independently.
- **O2** Review process should be censorship resilient.
- **O3** Review process should be transparent and enable traceability of artifacts.

Objective **O1** describes a fundamental requirement to enable an independent review process. The submitted reviews themselves as results of the review process should be produced independently.

Another key requirement to enable an independent process is censorship resilience in **O2**. We can think of different forms of censorship that may be enforced at various stages in the process. Censorship could target reviewers to exclude unwanted reviews, e.g. by selectively announcing the review process or suppressing review submissions. In addition, censorship could also focus on the disclosure of review results by completely or selectively dismissing the publication of reviews to all parties or targeted relying parties. In summary, any form of censorship would distort review results for some or all relying parties and therefore compromise an independent review process.

Lastly, an independent review process is enabled by **O3**. Note that the term ''transparency'' has diametrically opposed meanings in the field of computer science, either referring to the unhindered observability or complete obscurity of a process. By transparency, we refer to the former meaning and describe the ability of all stakeholders of a software to observe crucial operations of its review, such as its initiation or the submission of review results. Traceability, meanwhile, describes the ability of stakeholders to understand the provenance and authorship as well as the relation between artifacts within a given review instance. In this way, transparency of the overall process is a necessary prerequisite for traceability of corresponding artifacts. Thus, **O3** enables relying parties to interpret and assess review results. This objective is also critical to verify the overall process, i.e. **O1** and **O2**. Consequently, transparency of the overall process and traceability of corresponding artifacts is critical for all stakeholders to assure themselves of a fair review process.

Each review process references different types of artifacts, i.e. a software binary under review, attribute claims, and submitted review results. To enable the process-related objectives (**O1**-**O3**) we frame the following objective for such review artifacts:

O4 Review artifacts, including results, should be identifiable, persistently available, and traceable to the review instance and the software release under review.

Artifacts should be clearly identifiable to ensure that all reviewers review the very same software binary with respect to consistent definitions of attribute claims. In addition, relying parties must be able to uniquely identify all review submissions of a review instance and of a specific software release. This ability prevents disputes on whether a specific review was submitted within the considered review instance as well as disputes on the context of reviews. Traceability of reviews is also of particular importance to judge the independence of obtained review results in hindsight, as any given software may undergo a review process more than once. Results of later reviews may then be dependent on previous reviews, but they should still be independent of each other within their own round. Complementary, the persistent retrievability and traceability of all review artifacts ensures that the review process is traceable for all stakeholders in the long-term. Thus, it enables a relying party to collect review results pertaining to a given release at any time.

Finally, we specify the scope of facilitated reviews:

O5 Possible attribute claims and review consolidation methods should be use case agnostic.

Objective **O5** describes that the overall process should not be tailored to certain kinds of attribute claims to allow diverse investigations of a given software. It should also be flexible with respect to the consolidation of reviews. Therefore, relying parties are enabled to use results in a way that meets their specific information needs.

B. EXEMPLARY REVIEW SCENARIO

We illustrate a software review process in the context of automotive software update management systems. This domain faces several challenges: updating software in a mixed-critical system that can harm road users; complex pre-conditions for software changes as a consequence of many product variants; and updating a system-of-systems that runs software from a variety of software producers. To enable decisions on a particular software update, we aim to describe it with attributes that verifiably result from an independent and transparent review process.

To present a more concrete example, we introduce a toy open-source software component under review. Note that this paper discusses claims of open-source software for illustration purposes. The overall process, as we describe it later, does not require open-source software as review objects and is expected to be equally suited for closed-source software, e.g. by specifying claims that can be evaluated using black-box testing techniques. The source code of our example is listed in the appendix in Listing 1. The component is located on the interface of a Battery Management System (BMS) in an Electric Vehicle (EV). It is responsible for reducing the precision of the externally shared State of Charge (SoC). The EV's SoC may be of interest for various processes outside the EV, e.g. to display the current SoC on the owner's smartphone or to plan charging stops in the back end of a third party navigation system. Sharing battery data of an EV entails the risk of inadvertently disclosing sensitive information on the driver's behavior [25] and may allow profiling [26]. This risk is comparable to privacy problems enabled by the Battery Status API in HTML5, reported by Olejnik *et al.* in 2015 [27]. Device fingerprinting was made possible by allowing accessed websites to query the host's high precision SoC [27], [28]. In response, the World Wide Web Consortium now warns of exposing high precision battery status information [29]. Therefore, the correct implementation of reducing the precision of shared SoC data may be of interest for end users.

If a user wants to ensure the compliance of the SoC precision reduction functionality in their vehicle with respect to its specification, they have to trust the responsible Original Equipment Manufacturer (OEM). This reliance can be reduced using an independent review process of the OEM's software attribute claims. By having a group of independent reviewers examine attribute claims, the user no longer has to trust a central authority. Independent reviews may be especially relevant for software components that are of interest to users but are not critical enough to require a rigorous third party assessment before its release according to type approval regulations.

We motivate the objectives set in the previous section with this example. Vehicle users want to decide on a software update without blindly trusting the corresponding OEM. Hence, users expect reviews to be conducted by different, organizationally independent parties, as specified by **O1**. Independent reviews assure users that the OEM's claims are reviewed critically. Users only benefit from these independent reviews if they get access to all reviews submitted during the review period. Censorship resilience of **O2** is therefore important to ensure that users are not misled by a selective publication of review results. In addition, users also benefit from a transparent review process as specified by **O3**. Transparency ensures that users are able to retrace how the review was conducted. As a result, users can assess the quality of the review process as well as the results' applicability to their information needs. Other parties also benefit from transparency. For example, it can protect the OEM from unfounded criticism.

To enable a transparent and retraceable process, users must be able to identify and trace all referenced artifacts at any point after the reviews completion (see **O4**). This objective therefore enables car users to access relevant review results when required. They can also assure themselves that the updated software in their vehicle is indeed the same component that was reviewed. Additionally, users can retrace the exact claims and additional information that were made available to reviewers.

Lastly, **O5** targets the flexibility of attribute claims and consolidation methods. Regarding the toy software component of our use case, the following exemplary claims illustrate the range of conceivable attribute claims:

C1 Reproducible build
C2 Reproducible formal verification of precision-loss functionality
C3 Valid formal specification of precision-loss functionality
C4 No unspecified behavior detected

A reproducible build [30] in **C1** describes the attribute that the build process from source files to the reviewed software binary is independently reproducible with regard to specified dependencies and settings. The resulting attribute therefore attests the absence of compromises in the build process of the binary and enables the transfer of claims related to source code (**C2** - **C4**) to the reviewed binary. Reproducible builds also enable the vehicle user to make sure that the software binary in their vehicle corresponds to the exact source code presented to reviewers.

For claims **C2** and **C3** we assume that the software maintainer used formal verification to prove the functional correctness of the software under review, i.e. the precision-loss of the resulting SoC is implemented as specified. Claims **C2** and **C3** aim to independently verify and validate the OEM's formal verification. **C2** is similar to **C1** since the reviewers are asked to reproduce the specified formal verification process in their own environment and check if the verification passes successfully. If independent reviewers can reproduce the OEM's functional verification, the vehicle owner can be assured that the OEM actually executed the verification and reported its result honestly.

C3 refers to the subjective validation of the given formal functional specification with respect to the source code and its specified precision-loss functionality. Reviewers are asked to check if the formal specification correctly matches the intended functionality. If the reviewers attest **C3**, the vehicle user can be assured that the OEM provided a sound specification. In summary, the resulting attributes from **C2** and **C3** attest that the intended functionality of the software in question was correctly specified and successfully verified.

C4 describes an even more subjective attribute. By attesting that no unspecified behavior was detected, the resulting attribute states that no unwanted or potentially malicious functionality was found in an independent review according to the subjective assessment of each individual reviewer.

In consequence, if all four claims are attested in a review process, the user may be assured that the software component in their vehicle fulfills its specification and does not introduce unexpected behavior.

Since **O5** also targets the flexibility of review consolidation methods, the user receives direct access to each review result and can interpret and consolidate them according to their needs. For example, a vehicle user might benefit from a more aggregated result consolidation, e.g. the percentage of agreeing reviewers, whereas more proficient parties may require detailed results.

C. NON-OBJECTIVES

The set of objectives considered in this paper is not exhaustive to specify an independent software review. In particular, we exclude the following challenges from the scope of this paper.

We assume the voluntary participation of a group of independent reviewers. We omit the specification of an incentive mechanism to motivate reviewers, since this is a separate field of research. Haddi and Benchaiba [31] survey different incentive mechanisms and categorize them into economy- and reciprocity-based approaches, which includes reputation systems. In addition, we exclude mechanisms to mitigate collusion between reviewers, which poses a threat to independent reviews. Collusion detection and resistance also constitute a different research area [32], [33].

Complementary to the execution of the general review process, particular tools and methods for software examination, while vitally important to our concept, are not the focus of this work. Our goal is to enable those with expertise in the matter to conduct decentralized reviews of software with as broad of a set of tools and methods as possible.

IV. CONCEPTUAL DESIGN

In this section, we present a modular system design for a decentralized software review, depicted in Fig. 2, and describe how each module supports the objectives described in Section III.

A. DECENTRALIZED PLATFORM

As the fundamental layer of our system design, decentralized platforms like distributed ledger technologies (DLTs), particularly blockchains, fit well with the objectives of censorship-resilience (**O2**) as well as transparency and persistence of results (**O4**). Without single points of trust or failure, these systems generally make it difficult to maliciously prevent any party from participating in or observing procedures executed on top of them. In a similar way, they also ensure the integrity and continued availability of past records as part of their core functionality.

B. SOFTWARE IDENTITY MANAGEMENT

As the primary subject of our endeavor, software in general and associated binaries in particular need to be identifiable and referenceable. A software identity management system enables developers and maintainers of software to establish and manage an identity representation of their software and associated binaries. One important aspect of such a system is to ensure an integrity-protecting binding between the identity representation of a software and an associated binary release. Such bindings are of particular importance for our purposes because they prevent mismatches between the inputs and results of a review process and form a necessary prerequisite for objectives involving links to the software under review (**O4**).

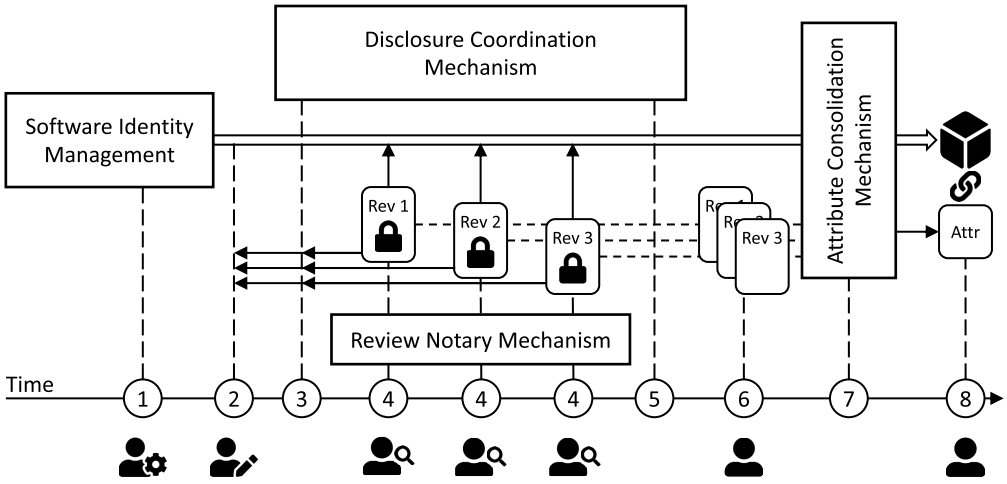

FIGURE 2. A module interaction diagram based on the modules described in Section IV. (1) Maintainer establishes software identity. (2) Claimant initiates review by publishing attribute claims to be verified. (3) Setup for disclosure coordination mechanism completed, starting the review period. (4) Reviewers submit reviews through notary mechanism to log them within the review period. (5) Disclosure coordination mechanism ends the review period, making contents of all appropriately prepared reviews public. (6) Relying parties can obtain review results and perform their own aggregation. (7) Review results are collectively consolidated into concise attributes and attached to software identity. (8) Relying parties can obtain consolidated attributes along software binary.

C. DISCLOSURE COORDINATION MECHANISM

While the transparency of decentralized platforms and the immutability of their stored records are beneficial to **O3**, these properties also pose a challenge to the independence of reviews. We define the module of a disclosure coordination mechanism to overcome this challenge and facilitate the logging of independently generated reviews (**O1**). More specifically, such a mechanism employs a cryptographic scheme to decouple the publication, timestamping, and immutable logging of reviews from the disclosure of their contents. In conjunction with the decentralized platform, this disclosure coordination mechanism also serves to document the independent creation of reviews for posterity. In order to not introduce a single point of trust, a suitable disclosure coordination mechanism needs to be decentralized as well.

D. REVIEW NOTARY MECHANISM

As mentioned in the previous module, the timestamping and logging of reviews is an integral part of an independent (**O1**) review process that is encapsulated in the review notary mechanism. Furthermore, this module serves to establish verifiable and undeniable links between each reviewer, their reviews, the respective software releases, and the employed disclosure coordination mechanisms (**O3**, **O4**). Fortunately, logging such interlinked data is precisely what distributed ledgers were designed for.

E. ATTRIBUTE CONSOLIDATION MECHANISM

The results of individual reviewers can already be considered attributes. However, depending on use-case and context, we envision two general approaches for post-processing and consolidating these attributes after the disclosure coordination mechanism ends a given review period:

1) INDIVIDUAL CONSOLIDATION

Relying parties can obtain and examine individually generated reviews for a software release and consolidate them either manually or algorithmically according to their own standards or requirements. With this approach, there would not be one singular module for attribute consolidation. Similarly, no further records need to be logged publicly, since this is a subjective and local procedure.

2) COLLECTIVE CONSOLIDATION

Contrary to the subjective and local consolidation, there could also be the need for a more public consolidation. In this case, conflicts between reviews need to be resolved and the final result, namely a set of concise attributes, is cryptographically attached to the software or binary as defined by the software identity management module. For the sake of transparency, traceability, and accountability, decisions made during this consolidation should be logged and persisted on the decentralized platform. Similar to the disclosure coordination mechanism, a practical implementation of this module should not introduce or rely on a single point of trust, as that would invalidate the goal of a fully decentralized software review process.

V. IMPLEMENTATION

In this section, we present a proof of concept implementation for a decentralized public review of software attribute claims via Ethereum to illustrate practical implications and tradeoffs. We construct this implementation by integrating

previous works [5]–[7] together with some additions into a coherent system based on the modular design presented in Section IV.

A. COMPONENTS

We first cover individual components of our implementation and describe how they fit into the modular system design of Section IV. For the sake of clarity, we focus the description of each component on the functionality and interface most relevant to their role within the overarching system, an overview of which is depicted in Fig. 3.

1) DECENTRALIZED PLATFORM

We opted to use Ethereum as the first part of a decentralized base layer for our implementation. Due to its popularity and market capitalization, Ethereum provides a realistic benchmark for practical feasibility and costs. As a public permissionless replicated state machine, Ethereum not only covers the basic functionality of a distributed ledger when it comes to recording and making available data, but it also provides an execution environment for on-chain programs called *smart contracts*, which are an essential building block for our implementation. A subtle but rather important feature of Ethereum smart contracts are their globally unique addresses. In the proof of concept implementation, we employ contract addresses to reference contracts as objects as well as to further reference data that defines its state.

Ethereum also provides a structured logging and monitoring functionality for applications that rely on smart contracts in the form of *events*. An event is a contract-defined data structure consisting of a number and type of parameters, up to three of which can be indexed. Ethereum clients can efficiently monitor new blocks for certain events or search prior blocks for events on behalf of users or applications to identify contracts or transactions of relevance. Indexed event parameters allow more fine-grained monitoring and searching queries. Unindexed event parameters are stored as data within transaction receipts, a data structure contained in every block of the Ethereum blockchain. While such event data is inaccessible to smart contracts, this mechanism constitutes a cheap and efficient way of logging and distributing data. In our case, we use it for references to resources stored off-chain.

Rather than attempting to record artifacts only on the Ethereum blockchain, which would result in either severe file-size limitations or excessive costs, we employ the InterPlanetary File System (IPFS) [34] as the second part of our decentralized base layer and only record references on-chain. These on-chain references are so called content identifiers (CIDs) which take the form of specially formatted strings of variable but small lengths. CIDs employ cryptographic hash functions to form integrity-protecting references to files based on their contents. For our purposes, it is sufficient to understand that, given a CID, the corresponding review artifact can be retrieved and its integrity verified by anyone, so long as someone hosts this file via IPFS. Those obtaining such a file can host it themselves, thereby increasing its availability.

It is important to note that our implementation relies on the standard functionality of Ethereum and IPFS client applications. Our implementation did not necessitate any changes to these clients and we therefore do not evaluate their performance in Section VI.

2) SOFTWARE IDENTITY MANAGEMENT

Before explaining this component, it is helpful to first recall and expand the brief introduction of software identities from Subsection IV-B. We describe a software identity as a uniquely referenceable and managed collection of binaries which are usable expression of said software. It is vitally important that only the maintainer of a software is able to make changes to this collection by adding new binaries or removing those no longer fit for use. Similarly, the link between a software identity and each of its binaries must ensure the integrity of the latter. Ultimately, the goal of software identity management is to enable users of a software to ensure that the binary they obtained belongs to the software they expect, that the respective maintainer has not revoked it, and that it was not modified in any way. Consequently, the data representing a software identity must be both highly available and protected against unauthorized changes.

We previously devised a software identity management system based on Ethereum smart contracts called Palinodia [5], [6]. A software maintainer establishes a Palinodia instance consisting of multiple smart contracts to represent a software identity as described above. Most important to this work are Binary Hash Storage (BHS) contracts. Their main purpose is to function as a key-value registry for cryptographic hashes of binaries to attach them in an integrity-protecting way to their software identity. To add a new binary to a BHS contract, a maintainer chooses a new *HashID*, includes it together with the globally unique address of said BHS contract as metadata in the binary, and submits both HashID and a cryptographic hash over binary and metadata to the BHS contract through a `publishHash` transaction. During this transaction, the programming of the BHS contract ensures that the transaction was sent by an authorized maintainer and that the submitted HashID is free. The BHS contract then persists the submitted hash as part of its state with the HashID as its key. Consequently, a BHS contract address together with a HashID uniquely identifies a binary and allows anyone to check its integrity and revocation status.

For the purpose of monitoring important changes to software identities, Palinodia makes ample use of Ethereum events. Most crucial for the purpose of this work are events emitted during the publication of new binaries to a BHS contract as part of the `publishHash` transaction, which we extend to serve as the initiation of a review process for the newly released binary.

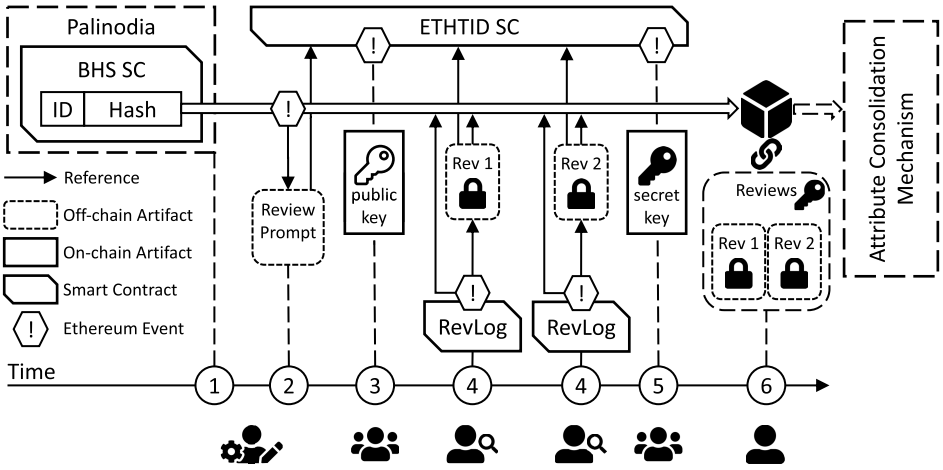

FIGURE 3. Overview of actors, smart contracts, events, and artifacts congruent to the system design depicted in Fig. 2. (1) Maintainer establishes software identity via Palinodia. (2) Maintainer publishes a new binary release to Palinodia referencing a review prompt (on IPFS) containing attribute claims to be verified and a reference to an ETHTID instance for scheduling. (3) Council of ETHTID instance releases a public key for encryption, thus starting the review period. (4) Reviewers perform their examinations and encrypt their results using the ETHTID public key before making them available via IPFS. Each reviewer announces their review through their own Review Log (RevLog) smart contract to timestamp them within the review period and link them to both the software under review and ETHTID instance for coordinated disclosure. (5) Council of ETHTID instance releases secret key for decryption, thus ending the review period. (6) Relying parties can obtain and decrypt review results attached to a software binary.

3) DISCLOSURE COORDINATION MECHANISM

As a distributed ledger, Ethereum is designed to persistently log and quickly propagate information. This functionality presents a challenge for an independent review process as each review should be recorded without knowledge of any other reviews for a given software release but all reviews should become available after a well-defined review period ends. To overcome this challenge, we employ ETHTID [7] as the disclosure coordination mechanism of our implementation.

An ETHTID instance consists of a single Ethereum smart contract that orchestrates a council of independent secret keepers for a threshold-based secret sharing. The basic functionality of ETHTID is to provide a decentralized ''time capsule'' functionality as a service by providing the public and secret keys of an ElGamal [35] key pair at different points in time: The public key is generated by the respective council during a setup procedure whereas the secret key exists in a shared state among said council until its scheduled recovery and publication. The release of both public and secret key is accomplished through transactions, `submitPubKey` and `submitSecKey` respectively, sent by any council member at the appropriate time. Similar to the publication of a new binary hash in Palinodia, both of these key release transactions emit an Ethereum event.

By referencing an ETHTID smart contract and using its public key, arbitrary information can be be added to a time capsule, i.e. prepared for a coordinated disclosure. With the publication of the secret key, the corresponding time capsule is opened and all properly prepared information it contained becomes public. Note that one ETHTID time capsule can be referenced and used by multiple applications so long as the specified schedule matches their requirements.

4) REVIEW NOTARY MECHANISM

In addition to the components based on our previous works described above, we also need to introduce Review Log smart contracts as a new component to realize the review notary mechanism described in Subsection IV-D. Fortunately, the majority of the design objectives for this module are inherently covered by the core functionality of Ethereum. The primary utility of each Review Log contract is to facilitate an access-controlled transaction named `publishReview` for emitting events to time stamp the release of a review and verifiably link its IPFS CID to both its author's Ethereum address as well as the software under review via a BHS contract address and HashID.

For the sake of simplicity, we opted for each reviewer deploying and using their own Review Log contract, the functionality of which could in practice also be included in another smart contract rather than being separate. Another option would be to deploy only one autonomous Review Log contract for everyone, in which case the ''review published''-event would have to be extended to include each review's author.

B. ETHDPR

In this section, we present ETHDPR, our proof of concept implementation of a decentralized public software review on Ethereum. As a process-oriented, modular framework, we describe the interactions and artifacts in the order they are relevant to the initiation, execution, and conclusion of

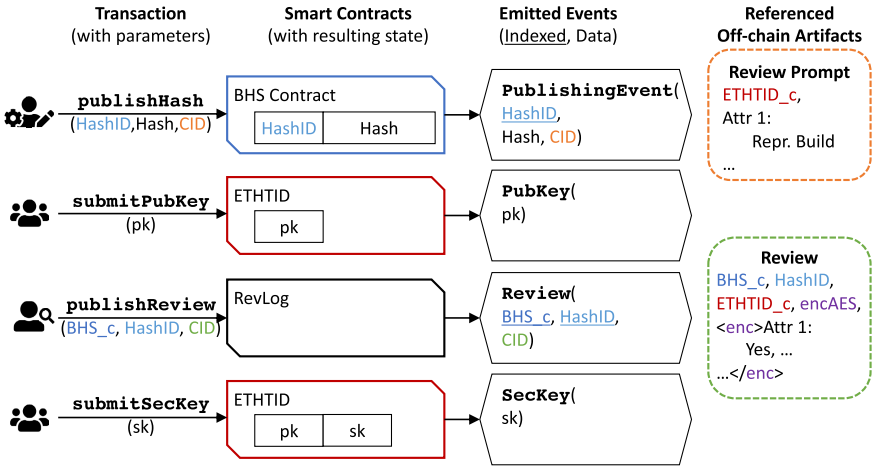

FIGURE 4. Relations between transactions, smart contracts with their resulting states, emitted events, and referenced off-chain artifacts. Colors denote references, either as contract addresses (ETHTID_c, BHS_c), strings (HashID), or IPFS CIDs. Underlined event attributes are indexed for the purpose of monitoring and searching. On-chain artifacts are stored as part of the state of their respective smart contracts and depicted as enclosed boxes.

a software review as depicted in Fig. 3. An overview of all transactions, resulting smart contract states, emitted events, and artifacts is given in Fig. 4.

First, a software maintainer creates a Palinodia instance to establish an identity for their software in order to attach a new binary release for the purpose of a review. It is important to note that we opted to unify the roles of software maintainer and claimant as described in Section I for the sake of simplicity, a point we elaborate in Section VII.

Next, a software maintainer creates and references an ETHTID instance to enforce the schedule of a software review, particularly the coordinated disclosure, via its "time capsule" functionality. Note that an already existing ETHTID instance with a suitable schedule could also be used. The way we integrate ETHTID into ETHDPR is through an Elliptic Curve Integrated Encryption Scheme (ECIES) [37]. Basically, each reviewer derives a unique symmetric AES key from an ETHTID public key and attaches it to their review such that it can only be recovered with the corresponding ETHTID secret key, as depicted in Fig. 5.

A software maintainer initiates an ETHDPR execution by preparing a JSON-formatted *review prompt* consisting of a reference to an ETHTID instance for scheduling, a description of claimed software characteristics to be verified, and all other resources necessary for the review process such as source files and metadata. Within such a review prompt, each software attribute to be verified is assigned an identifier that reviewers later use to structure their results. As part of the `publishHash` transaction to add the hash of a new binary to their BHS contract, the software maintainer also includes the CID of their review prompt, which they make available via IPFS. To save on costs, the CID is only emitted as event data and not stored as part of the BHS contract state.

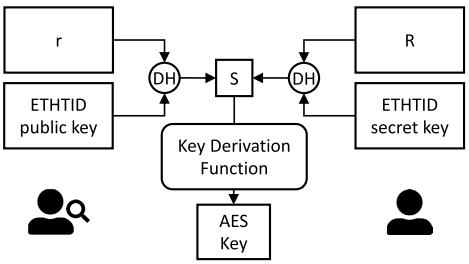

FIGURE 5. In order to prepare their review results for a coordinated disclosure, each reviewer derives a unique AES key by first generating a random, ephemeral ElGamal key pair (*r*, *R*) and then performing a Diffie-Hellman key exchange [36] with *r* and an ETHTID public key to generate a shared secret *S*. Through a key derivation function (SHA-3 in our case), an AES key is derived from *S* which each reviewer uses to encrypt their results. Each reviewer attaches *R* to their review, thus enabling any relying party to recover *S* and the AES key once the corresponding ETHTID secret key is available, thereby completing a coordinated disclosure of review results.

Reviewers with an interest in reviewing new releases of a particular software can instruct their Ethereum clients to monitor new blocks for events of corresponding BHS contracts. By parsing such events, reviewers find CIDs of the maintainer's review prompts and are able to obtain copies via IPFS. They can then also retrieve the public key of the referenced ETHTID instance via their Ethereum client, or instruct their client to monitor the ETHTID instance for a future release of said key via the `PubKey` event. Reviewers can now perform the requested review tasks to the best of their abilities and with their preferred or available tools and prepare their JSON-formatted report as follows.

To facilitate the coordinated disclosure of results, reports are split into a header and an encrypted payload. Using the identifiers for software properties from the review prompt, each reviewer records the result of their examination alongside additional information such as tools or methods

used, additional observations, or counterexamples. Then, using ECIES with the ETHTID public key as depicted in Fig. 5, each reviewer generates a random AES key to encrypt their results into the payload of their report while adding the necessary information to recover said AES key with the ETHTID secret key to the header of their report. The header of each report also contains the addresses of both the ETHTID and BHS contracts as well as the HashID of the binary release, thereby linking the software under review and enabling anyone obtaining the review from eventually obtaining its contents.

Once the report is complete, each reviewer makes it available via IPFS and broadcasts the CID via a `publishReview` transaction to their own Review Log smart contract which emits a `Review` event. Similar to the `PublishingEvent` emitted during the `publishHash` transaction, both the BHS contract address and HashID are included as indexed event parameters to facilitate monitoring for and collecting of reviews related to a particular release.

Once the ETHTID council releases their secret key via the `submitSecKey` transaction, the review period ends and the encrypted payloads of all correctly prepared reports become decryptable. Since events signifying the release of binaries, attached reports, and ETHTID keys (see Fig. 4) are timestamped and immutable proofs of existence for their respective artifacts, they serve as a time base both during and after an ETHDPR execution. More specifically, reports can be generated, encrypted, and logged on-chain after an ETHTID secret key is published, but the order of the corresponding `Review` events relative to the `SecKey` event clearly distinguishes such reports as potentially dependent on previously logged reports.

C. EXEMPLARY REVIEW EXECUTION

To illustrate ETHDPR, we demonstrate an exemplary decentralized review regarding a software update of the toy software component introduced in Subsection III-B. Exemplary review artifacts are listed in the appendix (Listings 1 to 7).

In this example, an OEM takes on the maintainer role. In addition, this OEM acts as the claimant in order to increase vehicle users' trust in the software under review. Since ETHDPR is open for any interested reviewer, we assume the participation of a variety of reviewers such as industry experts, researchers, or vehicle users with knowledge in software testing. The parties relying on the review results consist of vehicle users and the OEM itself. As part of this review, the OEM claims **C1**-**C4** as introduced in Subsection III-B.

To enable the review process, the OEM has already established a uniquely referenceable software identity of the software under review on Ethereum using Palinodia (**(1)** in Fig. 3). The specific software release under review represents the binary of an updated version of this software. This new binary release is added to the established software identity. In our proof of concept implementation, the review prompt (**(2)** in Fig. 3) is published together with the new software release under review by the maintainer.

The review prompt is structured as follows and drafted in Listing 2. First, the prompt contains references to the unique context of the specific review process. To find the corresponding disclosure mechanism, the chosen ETHTID contract address is referenced. To identify the software binary under review, the software identity is referenced by its BHS contract address and HashID. The remaining review prompt defines the different attribute claims reviewers are asked to examine in this review instance. We note that attribute claims such as **C1**-**C4** are themselves based on, potentially unverified, attributes of the software release, e.g. the software's source code or specification. To ensure consistency among claims within a review, the artifacts specifying these attributes are defined in a nested JSON object and referenced in the corresponding claims. The content of an artifact may either be directly specified in its JSON value or pointed to by an integrity-protecting reference, e.g. an IPFS CID. Please recall that a relying party can easily verify that the source code regarded in reproducible builds of **C1** is the same as in the remaining source code related claims **C2**-**C4**. Therefore, an attestation of **C1** allows the relying party to apply the results of **C2**-**C4** to the software binary under review. The claims are listed as key-value pairs to allow unique references in reviews to the examined claim by its corresponding key.

Let us explain claim **C2** in more detail. As stated in the review prompt in Listing 2, **C2** references the C source code of the software release under review as well as a formal specification in the ANSI/ISO C Specification Language (ACSL) [38] and metadata to reproduce the deductive verification of the software's precision-loss functionality using the Weakest Precondition (WP) plug-in in Frama-C [39]. Frama-C is a tool suite to analyze source code in C.[3] Note that Frama-C and its WP plug-in merely serve as an example in this paper, other tools could be used as well. In total, **C2** claims that the referenced source code can be formally verified using the referenced formal specification if the verification process is conducted as described in the referenced additional information, i.e. using the same Frama-C command. Listing 1 shows C source code of the software under review for our toy example. The State of Charge (SoC) representation is based on the published encoding of SoC data in controller area network (CAN) bus messages [40], [41]. In particular, we assume an integer representation of the SoC which is computed by multiplying the decimal SoC with an integer factor. For example, the car manufacturers Peugeot and Renault reportedly use the factor 100 [41]. If the decimal SoC is 12,34% and the factor is 100, the encoded input SoC of the software's function `encoded_floor` is 1234. To reduce the SoC's precision, `encoded_floor` should return the encoded equivalent of the floor function, in this case 1200 which represents

[3] https://frama-c.com/

12.00%. If the function's input is invalid, i.e. the values are out of range, `encoded_floor` should return the error code −1. To verify the functionality of `encoded_floor`, the OEM formally specifies the expected functionality in ACSL, as depicted in Listing 3.

The last artifact referenced in **C2** contains additional information to specify the execution of Frama-C. For example, the OEM might state the exact Frama-C command that should be used by reviewers to reproduce the formal verification, as depicted in Listing 4.

The OEM chooses an ETHTID instance that starts (**(3)** in Fig. 3) once the software release and its review prompt are published. Next, reviewers conduct their reviews and announce their encrypted results through their respective RevLog contracts. The structure of a review result is listed in Listing 5. It contains the boolean decision for or against attestation of the claim as well as optional attachments per reviewed attribute claim. The reviewer has the option of reviewing any subset of claims by leaving the value object of unanswered claims empty. In this example, the reviewer responds to and agrees with **C2**.

Attachments allow the reviewer to explain their result and, thereby, increase trust in the review or simply provide additional information for relying parties. For **C2** the reviewer could attach the report result from Frama-C, as indicated in Listing 6.

To publish the review of Listing 5, its results are encrypted as described in Subsection V-B. The reviewer publishes the encrypted review results with additional metadata in its header, see Listing 7. The metadata links this review to its disclosure-coordinating ETHTID contract, the software release under review, and the review prompt CID, which defines the reviewed claims. In addition, the published review contains an encrypted AES key. Once the review period has been terminated (**(5)** in Fig. 3), each AES key and thus all review results can be decrypted and used by relying parties (**(6)** in Fig. 3).

VI. EVALUATION

In this section, we evaluate our implementation of ETHDPR both quantitatively and qualitatively. The proof of concept implementation is comprised of an on-chain part consisting of Ethereum smart contracts and an off-chain part in the form of a Python application to generate, encrypt, and decrypt artifacts. Gauging the practicality of the proof of concept implementation quantitatively tackles both parts differently. For the on-chain part, we measure the required contract execution effort via the monetary costs that would occur in practice by simulating corresponding transactions. Other metrics like latency between the submission of a transaction and it being validated and recorded on-chain are highly volatile in practice and are independent of our implementation, i.e. no matter how our implementations works, this latency is entirely dictated by current demand for transactions to be recorded on-chain. For the off-chain part, we focus on the cryptographic operations with encrypting and decrypting reports as the underlying elliptic curve, which is the only one Ethereum currently supports, is generally not intended for this use case. We refrain from evaluating the performance of IPFS, since we merely rely on standard IPFS client functionality and did not make any changes to such clients that would warrant an evaluation. See, for example, the work by Shen *et al.* [42] for a thorough performance evaluation of IPFS.

TABLE 1. Gas costs for publishing hashes to BHS contracts with CIDs and for publishing CIDs to reviews.

	Binary Hash		Review	
CID Length	Gas	USD[1]	Gas	USD[1]
0	78 673	17.41	-	-
46	81 383	18.01	28 411	6.29
60	81 552	18.05	28 579	6.32
66	82 082	18.17	29 111	6.44
111	83 082	18.39	30 111	6.66

[1] Conversion rates: USD 3077.74 per ETH, ETH 71.91×10^{-9} per gas.

A. QUANTITATIVE EVALUATION

Transactions on Ethereum that alter any part of its global state and must therefore be recorded on-chain, incur costs in the form of *gas*. State changes of those transactions are determined by executing relevant smart contract code in the Ethereum Virtual Machine (EVM). Each basic operation in the EVM, like adding or multiplying integers, loading from or writing to persistent storage, or conditional jumps, has a fixed gas cost attached to it. When preparing a transaction, a sender chooses how much Ether per gas they are willing to pay to have their transaction executed and recorded in a future block.

To evaluate the on-chain components, we compiled all relevant smart contracts with solc 0.8.10 using the `optimize` flag and deployed them to a local Ganache (v6.12.2) development blockchain running on the Muir Glacier hard fork. We then executed the relevant smart contract functions and recorded their gas costs for various CID lengths. For better intuition, we also report gas costs converted to USD based on the average exchange rates of 20th April 2022 as reported by Etherscan.[4]

Table 1 shows the gas costs for recording a new binary release in an unmodified Palinodia BHS contract without a CID (CID length 0) and a modified BHS contract to also record a CID as event data. Also shown in Table 1 are the costs for recording a CID for a review via a RevLog contract as described in Subsection V-B. The CID lengths were chosen to be representative of IPFS CIDs with various embedded multihashes as described by the IPFS documentation:[5]

- 46: IPFS CIDv0.
- 60: IPFS CIDv1 with sha256.

[4] https://etherscan.io/

[5] https://docs.ipfs.io/concepts/content-addressing/

TABLE 2. Mean decryption time in seconds (rounded to two decimals) of different sized review sets with different review attachment sizes over 100 repetitions.

Review Set Size	Attachment Size per Review in MB					
	0	0.2	0.4	0.6	0.8	1.0
10	0.10	0.12	0.14	0.17	0.19	0.21
20	0.21	0.25	0.29	0.33	0.37	0.41
30	0.31	0.38	0.44	0.51	0.57	0.63
40	0.41	0.49	0.58	0.67	0.75	0.84
50	0.52	0.63	0.72	0.84	0.95	1.06

- 63: IPFS CIDv1 with blake2b-256.
- 111: IPFS CIDv1 with sha3-512.

The disproportionate jumps in costs between CIDs of lengths 60, 66, and 111 are due to padding to the nearest 32 B for strings as required by the Ethereum contract ABI specification.[6] Still, the additional costs to log an IPFS CID alongside an already existing event, as in the case of publishing a hash, is very low with between 2710 and 4409 gas or USD 0.60 to USD 0.98.

Logging CIDs of reviews is comparatively more expensive but still reasonable. Deployment of the minimally viable RevLog contract, which consists of just one access-controlled function to emit an event as described in Subsection V-A, costs 252 503 gas or USD 55.88. In practice, it would probably be more sensible to include this functionality in other contracts that a reviewer would deploy for other activities. It is worth noting that, as long as the emitted events are congruent, i.e. they have the same name and arguments, the contracts that emit them can differ. Similarly, a relying party does not need to know the addresses of all RevLog or similar contracts beforehand, as consistently named and structured events enable their Ethereum clients to find both events and emitting contracts on demand.

Apart from on-chain gas costs, we also evaluate the required time to execute cryptographic operations in ETHDPR. In particular, we measured the off-chain computation time for a Python program to encrypt a review ((**4**) in Fig. 3) and decrypt reviews ((**6**) in Fig. 3) of varying sizes. These measurements are interesting because the encryption and decryption of the reviews using ECIES and ElGamal (see Subsection V-B) is neither common nor the most efficient approach. We were restricted to ElGamal due to the use of ETHTID which in turn is restricted by Ethereum currently only supporting one elliptic curve that is suboptimal for this use case. All measurements were repeated 100 times in an OpenStack virtual machine running Ubuntu 20.04 on four virtual 2.4 GHz AMD EPYC CPUs and 8 GB RAM using Python 3.8.10. The Python program randomly creates reviews analogous to the example in Subsection V-C. The rounded mean time for creation and encryption of reviews ((**4**) in Fig. 3) increases linearly from 0.02 s for reviews without an attachment to 0.18 s for reviews with an attachment of 1 MB. Table 2 depicts the mean time measurements for review decryption ((**6**) in Fig. 3), including a check for duplicate AES keys to identify obvious plagiarism. We evaluated sets of reviews in which each review consists of the same size of a random attachment, from zero to one MB. In addition, we looked at varying numbers of participating reviewers, from five to 50 reviews per review instance. Table 2 depicts a linear increase of time with increasing attachment size per review and increasing numbers of reviews per review instance. The worst case of our measurements is the decryption of 50 reviews with an attachment size of one MB each. For this task the rounded mean time measurement is 1.06 s, which is still reasonable and could be reduced further by optimizing the implementation. In summary, the required time for cryptographic operations in ETHDPR is negligible compared to the tasks of creating a review prompt, conducting reviews and evaluating review results, which could in total be on the order of hours or days.

[6] https://docs.soliditylang.org/en/develop/abi-spec.html

B. QUALITATIVE EVALUATION

In addition to the quantitative assessment of our implementations, it is also worth examining its qualitative properties based on the objectives described in Subsection III-A.

O1 (REVIEWS SHOULD BE CREATED INDEPENDENTLY)

The independence of attributions is facilitated through the use of ETHTID, but as we discuss in Section VII, the goal of strict enforcement of independence appears unattainable. For a malicious reviewer to violate this objective without cooperation of other reviewers would entail coercing or compromising a sufficient fraction of the council tasked with the safekeeping of the employed ETHTID instance's secret key. In practice, the ETHTID smart contract could be configured to economically punish council members if the instance's secret key were to be recovered prematurely, thus transforming this endeavor into a measure of economic means. A related threat to the independence of reviews is plagiarism. Due to the use of ECIES via ETHTID's temporally decoupled asymmetric key pair, plagiarism between reviewers is either immediately obvious or unpreventable. Copying an encrypted AES key and encrypted payload from any review and publishing it as one's own is immediately detectable even before disclosure. Altering both the encrypted AES key and encrypted payload without knowing the corresponding ETHTID secret key in order to avoid the above scenario is most likely not going to result in a sensible review once disclosed, if decryption is even possible. Lastly, reviewers can either share their AES keys or results of their reviews covertly, in which case it is less a case of plagiarism and more a case of voluntary collusion. As mentioned previously, such subversion of a review process are most likely not preventable by technological means.

O2 (REVIEW PROCESS SHOULD BE CENSORSHIP RESILIENT)
By virtue of being a public permissionless system, Ethereum itself already preempts many forms of censorship. Generally speaking, it is rather difficult to prevent any party in particular from writing or reading data to or from the Ethereum blockchain. It is similarly difficult to present two distinct versions of events to different parties. In our case, this means it is very difficult for any party to suppress the announcement of a software review, the publication of encrypted reviews, or their coordinated disclosure, both generally and for particular individuals. However, unforeseen spikes in transactions and subsequent congestion can lead to delays in getting transactions executed. It is therefore prudent to set time limits with enough leeway to handle such eventualities.

O3 (REVIEW PROCESS SHOULD BE TRANSPARENT AND ENABLE TRACEABILITY OF ARTIFACTS)
Similar to the point above, the public accessibility of Ethereum and interlinking of artifacts as depicted in Fig. 4 positively contribute to this objective. Additionally, all actions taken during the review process are tied to Ethereum addresses and thus attributable to respective parties, assuming all parties properly secure their Ethereum private keys, which is important regardless. Similarly, each reviewer is implicitly incentivized to publish their reviews via only one Ethereum address in order to build a reputation.

O4 (REVIEW ARTIFACTS, INCLUDING RESULTS, SHOULD BE IDENTIFIABLE, PERSISTENTLY AVAILABLE, AND TRACEABLE TO THE REVIEW INSTANCE AND THE SOFTWARE RELEASE UNDER REVIEW)
By being logged on the Ethereum blockchain, all timing-critical information and attached references to artifacts available via IPFS become part of a persistent and highly available record for each software review instance. Such a record is especially important to judge the independence of reviews by comparing their release timestamps to the start and end of the review period as enforced by an ETHTID instance. However, when it comes to the long-term availability of off-chain artifacts, the case is less straightforward. As IPFS only facilitates the dissemination of CID-addressed data but does not ensure or encourage their long-term availability, it is up to reviewers and other relying parties to keep off-chain artifacts available. Depending on the use case and context, existing institutions could perform the function of archives. Note that the integrity of any artifact is protected via the IPFS CID which is immutably logged on-chain, thus preventing modifications. Similarly, IPFS CIDs also serve to give each off-chain artifact a unique identity.

O5 (POSSIBLE ATTRIBUTE CLAIMS AND REVIEW CONSOLIDATION METHODS SHOULD BE USE CASE AGNOSTIC)
As long as attribute claims and attributions can be expressed in text form, ETHDPR supports them. As such, our implementation of ETHDPR does not further constrain the set of methods and tools that can be employed for software assessment.

VII. LIMITATIONS AND OPEN ISSUES

It is important to first acknowledge a fundamental issue with independent review processes. Technological means can enable or encourage independent work, but they can hardly enforce or guarantee it. Regardless of the methods or tools used to perform software reviews, humans are involved in some capacity to interpret, refine, transform, or combine results in order to decide whether or not to attribute a claimed characteristic of a software. In the end, it is the assessment of a human actor that makes a software review meaningful, as even fully automated tools for evaluation were created and assessed by humans. Humans also have the capacity to go against any given plan or process out of spite, ignorance, or a need to stand out. In the case of our implementation, anyone observing a software review in progress can decide to forego any coordinated disclosure mechanism and publish their results immediately, regardless of their accuracy, to diminish the independence of any properly prepared and disclosed reviews. As such, it stands to reason that any system or protocol for performing independent software reviews must assume a shared goal among participants and their voluntary compliance.

In a similar way, collusion and sybil attacks are equally fundamental and difficult to overcome challenges. Analogous to the scenario above, human participants in an independent review process can decide to collude and privately share or align their results for their mutual benefit, e.g. an overall smaller workload or increased credence of their results. Counter-collusion mechanisms have already been presented for simpler scenarios [43], but they rely on all relevant information being available on-chain, which is not necessarily feasible, as our proof of concept implementation shows. Likewise, by creating sybil identities, an individual reviewer can provide a disproportionately large amount of review results for both benign and malicious reasons. The only incidental counter measure in our implementation are the monetary costs associated with deploying RevLog contracts and using them to publish reviews. This issue emphasizes the need for robust identity management in decentralized systems, which is also an area of current research [44]–[47].

Performing a review of software attribute claims via decentralized platforms was not intended to overcome the aforementioned challenges. However, by examining the problem in general and implementing a proof of concept, we posit that the use of decentralized platforms, particularly permissionless ones like Ethereum, has both positive and negative effects. As a neutral and public infrastructure, decentralized platforms can serve as a ''common ground'' for execution and logging of such processes and thus focus the attention of all involved parties without relying on a centralized provider. While this makes a software review process more accessible to both good and bad actors

alike, it also forces any wrongdoing to be performed and persistently recorded in public view. The same persistent logging can also give greater weight to software attributes. Since there is no way to revise or rescind one's statement, it stands to reason that greater care is taken in their preparation.

It is also worth examining design decisions we took with our proof of concept implementation and their consequences. Initially, we chose to initiate a review for a software release by extending the publishing event in Palinodia with an IPFS CID of a review prompt out of simplicity and convenience. In this way, we implicitly restricted each software release to have at most one review and only the responsible maintainer could define and initiate it. In retrospect, this choice vastly simplified the overall implementation by preempting a host of difficult scenarios. For example, if one software release could undergo more than one review process, they could happen simultaneously with overlapping claims but differing ETHTID schedules. As mentioned above, such a scenario would unfold in plain view but it would nevertheless complicate both processes, including the evaluation of their respective results, and potentially diminish their independence from each other. Attributions of the review process with a later ETHTID disclosure may depend on similar attributions from the review process with an earlier disclosure, depending on when they were logged on-chain. One way to meet this challenge would be to extend software identity management systems so that anyone can initiate a review but only one review could be active at any given time.

In a similar way, attaching reviews and attributes to binary releases of a software simplified both the implementation and its presentation, but it could potentially be very cumbersome in practice. For instance, certain attribute claims may be verifiable for, and should thus be attached to, a higher level in a software identity hierarchy rather than being verified repeatedly for derived binary releases. Determining when such overarching attributes need to be re-evaluated due to significant changes to the common code base is a challenge currently being tackled [48]. Extending and adapting software identity management solutions to support such attribution and certification processes could be an interesting avenue for future work.

The execution environment of our proof of concept is also worth emphasizing. Applications on Ethereum have to deal with its unrestricted accessibility. In particular, this means that participants can create and act through as many identities as their financial means allow. However, transferring a system like the one presented in this work to a permissioned, and thus more controlled, environment should be rather straightforward. By vetting and provisioning reviewers, for example, the sybil problem mentioned above is rendered moot and consequences for inappropriate behavior can have a lasting effect.

We previously stated that a user can tell if the software binary running in their environment is the same as the reviewed software. However, this ability is based on the assumption that a user's system, e.g. their car, is not compromised. A compromised user environment may report a different software version than the one actually running on the system. ETHDPR by itself does not help in this regard but it could benefit from Trusted Execution Environments (TEE). Running an ETHDPR client inside a TEE could protect it from compromise and thus help establish a robust and secure bootstrapping chain.

Lastly, it is worth examining how a decentralized software review process relates to the concept of coordinated vulnerability disclosure, previously called responsible disclosure. If, for example, a reviewer discovers a critical security flaw as part of an attribute claim review, they can inform the responsible software maintainer just like before. However, depending on the details such a reviewer includes in their results, the previously negotiable dead line for the public disclosure of such a vulnerability is now fixed and enforced through the coordinated disclosure mechanism for the reviews.

VIII. CONCLUSION

In this work, we explored how existing tools and methods for software review and attestation could be employed on top of decentralized platforms like Ethereum. To that end, we decomposed the overall problem into distinct but interconnected modules to realize a well-structured review process. By integrating previous works based on Ethereum for two of these modules together with new additions both on- and off-chain, we provided a proof of concept implementation for a decentralized public software review. Lastly, we discuss both general and implementation-specific aspects of decentralizing such an independent review process. The confluence of ever-improving software testing and certification methods with rapidly improving decentralized platforms should provide an interesting avenue for cross-disciplinary future work. Similarly, the concept presented here may be transferable to use cases other than software attestation.

```
int encoded_floor(int soc, char factor) {
  if (soc < 0) {
    return -1;
  } else if (factor == 0) {
    return -1;
  }
  int max = 101 * factor;
  if (soc >= max) {
    return -1;
  }

  int soc_point =  soc % factor;
  int soc_rounded =  soc - soc_point;

  return soc_rounded;
}
```

LISTING 1. Example C source code of the exemplary review prompt claim artifact `source_code`. The code implements the precision-loss functionality of the example. It computes the rounded floor value `soc_rounded` of a decimal number that is represented as the integer `soc` via `factor` as described in Subsection V-C.

```
{
  "ethtid_contract_addr" : "...",
  "bhs_contract_addr" : "...",
  "hashID" : "...",
  "claim_artifacts" : {
    "source_code" : "...",
    "build_info" : "...",
    "formal_specification" : "...",
    "frama-c_info" : "...",
    "informal_specification" : "..."
  },
  "claims" : {
    "c1" : "Build is reproducible from
      ${source_code} using ${build_info}",
    "c2" : "Formal verification of the
      ${source_code}'s precision-loss
      functionality is reproducible in
      Frama-C with ${formal_specification}
      and ${frama-c_info}.",
    "c3" : "${formal_specification} of
      precision-loss functionality is valid
      with respect to the software's
      ${informal_specification}.",
    "c4" : "Potentially unwanted unspecified
      behavior was not detected in
      ${source_code} with respect to the
      software's ${informal_specification}."
  }
}
```

LISTING 2. **Example review prompt structure. A review prompt uniquely references a review instance by stating its corresponding ETHTID address as well as the BHS contract address and HashID of the software binary under review. The purpose of a review prompt is to state the claims that should be investigated in this review instance. Claims may reference additional information in the form of claim artifacts for consistency and readability. In the example we replace the values of references with ellipses for the sake of readability.**

```
/*@ assigns \nothing;
  behavior invalid:
    assumes soc < 0 || factor < 1
      || soc >= 101 * factor;
    ensures \result == -1;
  behavior valid:
    assumes soc >= 0 && factor >= 1
      && soc < 101 * factor;
    //valid result range
    ensures 0 <= \result <= 100 * factor;
    //soc_rounded is rounded
    ensures \result % factor == 0;
    //soc_rounded is floor of soc
    ensures \result <= soc
      < \result + factor;
  complete behaviors invalid, valid;
  disjoint behaviors invalid, valid;
*/

int encoded_floor(int soc, char factor);
```

LISTING 3. **Formal Specification in ACSL of the exemplary review prompt claim artifact `formal_specification`. The OEM presents this specification in the example of Subsection V-C to describe the formal verification of the expected functionality of the software binary under review and corresponding `source_code` (see Listing 1).**

APPENDIX
LISTINGS

In this appendix, we give listings for the example execution of ETHDPR in Subsection V-C.

```
frama-c -wp -wp-rte example.c \
  -then -report
```

LISTING 4. **Frama-C/WP plug-in command of the exemplary review prompt claim artefact `frama-c_info` that specifies how the OEM executed the claimed reproducible formal verification.**

```
{
  "c1" : {},
  "c2" : {
      "attestation" : true,
      "attachment" "..."
  },
  "c3" : {},
  "c4" : {}
}
```

LISTING 5. **Example review result structure stating an attestation of claim C2 while abstaining from all other claims of this review instance.**

```
[kernel] Parsing example.c
  (with preprocessing)
[rte] annotating function encoded_floor
[wp] 15 goals scheduled [...]
[wp] Proved goals:    15 / 15
 Qed:                  5  (0.68ms-7ms-60ms)
 Alt-Ergo 2.4.1:      10  (1ms-26ms-191ms)
 [...]
------------------------------------------
--- Status Report Summary
------------------------------------------
    15 Completely validated
    15 Total
------------------------------------------
```

LISTING 6. **Frama-C/WP plug-in verification report as an exemplary `attachment` explanation of a review result such as the attestation in Listing 5. Some details are omitted for readability.**

```
{
  "ethtid_contract_addr" : "...",
  "review_prompt_cid" : "...",
  "bhs_contract_addr" : "...",
  "hashID" : "...",
  "ethtid_ecies_aes_key" : "...",
  "encrypted_results" : "..."
}
```

LISTING 7. **Example review structure. Analogous to review prompts (see Listing 2), a review is uniquely referenced by the addresses of its corresponding review prompt, ETHTID contract, and software binary. The review also contains the encrypted results as well as the encrypted symmetric key to decrypt the results once the review period has been terminated.**

ACKNOWLEDGMENT

The authors thank Lionel Blatter for his feedback on their ACSL specification. They also thank all peer-reviewers for their valuable comments, as well as everyone who gave feedback on preliminary drafts of this article.

REFERENCES

[1] D. Kwan, L. M. Cysneiros, and J. C. S. D. P. Leite, "Towards achieving trust through transparency and ethics," in *Proc. IEEE 29th Int. Requirements Eng. Conf. (RE)*, Sep. 2021, pp. 82–93.

[2] E. Denney and B. Fischer, "Software certification and software certificate management systems," in *Proc. Autom. Softw. Eng. Workshop Softw. Certificate Manage.*, vol. 11, 2005, pp. 1–5.
[3] P. Heck, M. Klabbers, and M. van Eekelen, "A software product certification model," *Softw. Quality J.*, vol. 18, no. 1, pp. 37–55, Mar. 2010.
[4] P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices," in *Proc. 13th Int. Conf. Softw. Eng. (ICSE)*, 2008, pp. 541–550.
[5] O. Stengele, A. Baumeister, P. Birnstill, and H. Hartenstein, "Access control for binary integrity protection using ethereum," in *Proc. 24th ACM Symp. Access Control Models Technol. (SACMAT)*, New York, NY, USA, May 2019, pp. 3–12, doi: 10.1145/3322431.3325108.
[6] O. Stengele, J. Droll, and H. Hartenstein, "Practical trade-offs in integrity protection for binaries via ethereum," in *Proc. 21st Int. Middleware Conf. Demos Posters*, New York, NY, USA, Dec. 2020, pp. 9–10, doi: 10.1145/3429358.3429374.
[7] O. Stengele, M. Raiber, J. Müller-Quade, and H. Hartenstein, "ETHTID: Deployable threshold information disclosure on ethereum," in *Proc. 3rd Int. Conf. Blockchain Comput. Appl. (BCCA)*, Nov. 2021, pp. 127–134.
[8] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proc. 9th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, New York, NY, USA, 2013, pp. 202–212, doi: 10.1145/2491411.2491444.
[9] M. P. Jones, "Dealing with evidence: The programatica certificate abstraction," Dept. Comput. Sci. Eng., OGI School Sci. Eng. OHSU, Beaverton, OR, USA, Tech. Rep., Jan. 2002.
[10] G. C. Necula, "Proof-carrying code," in *Proc. 24th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, New York, NY, USA, 1997, pp. 106–119, doi: 10.1145/263699.263712.
[11] G. C. Necula and P. Lee, "Safe, untrusted agents using proof-carrying code," in *Mobile Agents and Security* (Lecture Notes in Computer Science), G. Vigna, Ed. Berlin, Germany: Springer, 1998, pp. 61–91, doi: 10.1007/3-540-68671-1_5.
[12] M. Alsayyari and S. Alyahya, "Supporting coordination in crowdsourced software testing services," in *Proc. IEEE Symp. Service-Oriented Syst. Eng. (SOSE)*, Mar. 2018, pp. 69–75.
[13] F. Pastore, L. Mariani, and G. Fraser, "CrowdOracles: Can the crowd solve the Oracle problem?" in *Proc. IEEE 6th Int. Conf. Softw. Test., Verification Validation*, Mar. 2013, pp. 342–351.
[14] M. Yan, H. Sun, and X. Liu, "ITest: Testing software with mobile crowdsourcing," in *Proc. 1st Int. Workshop Crowd-Based Softw. Develop. Methods Technol. (CrowdSoft)*, New York, NY, USA, 2014, pp. 19–24, doi: 10.1145/2666539.2666569.
[15] T. Long, I. Yoon, A. Porter, A. Memon, and A. Sussman, "Coordinated collaborative testing of shared software components," in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2016, pp. 364–374.
[16] S. S. Yau and J. S. Patel, "A blockchain-based testing approach for collaborative software development," in *Proc. IEEE Int. Conf. Blockchain (Blockchain)*, Nov. 2020, pp. 98–105.
[17] S. Homayoun, A. Dehghantanha, R. M. Parizi, and K.-K.-R. Choo, "A blockchain-based framework for detecting malicious mobile applications in app stores," in *Proc. IEEE Can. Conf. Electr. Comput. Eng. (CCECE)*, May 2019, pp. 1–4.
[18] Q. Hu, M. R. Asghar, and S. Zeadally, "Blockchain-based public ecosystem for auditing security of software applications," *Computing*, vol. 103, no. 11, pp. 2643–2665, Nov. 2021.
[19] H. Chen, H. Zhou, J. Yu, K. Wu, F. Liu, T. Zhou, and Z. Cai, "Trusted audit with untrusted auditors: A decentralized data integrity crowdauditing approach based on blockchain," *Int. J. Intell. Syst.*, vol. 36, no. 11, pp. 6213–6239, Nov. 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/int.22548
[20] J. Adler, R. Berryhill, A. Veneris, Z. Poulos, N. Veira, and A. Kastania, "Astraea: A decentralized blockchain Oracle," in *Proc. IEEE Int. Conf. Internet Things Green Comput. Commun. Cyber Phys. Social Comput. Smart Data*, Jul. 2018, pp. 1145–1152.
[21] J. Heiss, J. Eberhardt, and S. Tai, "From oracles to trustworthy data on-chaining systems," in *Proc. IEEE Int. Conf. Blockchain (Blockchain)*, Jul. 2019, pp. 496–503.
[22] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, J. Cappos, and B. Ford, "CHAINIAC: Proactive software-update transparency via collectively signed skipchains and verified builds," in *Proc. 26th USENIX Secur. Symp.* Berkeley, CA, USA: USENIX Association, Aug. 2017, pp. 1271–1287. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/nikitin
[23] J. Guarnizo, B. Alangot, and P. Szalachowski, "SmartWitness: A proactive software transparency system using smart contracts," in *Proc. 2nd ACM Int. Symp. Blockchain Secure Crit. Infrastructure (BSCI)*, New York, NY, USA, Oct. 2020, pp. 117–129, doi: 10.1145/3384943.3409428.
[24] M. N. Ince, M. Ak, and M. Gunay, "Blockchain based distributed package management architecture," in *Proc. 5th Int. Conf. Comput. Sci. Eng.*, Sep. 2020, pp. 238–242.
[25] A. B. Lopez, K. Vatanparvar, A. P. D. Nath, S. Yang, S. Bhunia, and M. A. Al Faruque, "A security perspective on battery systems of the Internet of Things," *J. Hardw. Syst. Secur.*, vol. 1, no. 2, pp. 188–199, Jun. 2017.
[26] A. Brighente, M. Conti, and I. Sadaf, "Tell me how you re-charge, I will tell you where you drove to: Electric vehicles profiling based on charging-current demand," in *Proc. Eur. Symp. Res. Comput. Secur.*, in Lecture Notes in Computer Science, E. Bertino, H. Shulman, and M. Waidner, Eds. Cham, Switzerland: Springer, 2021, pp. 651–667.
[27] L. Olejnik, G. Acar, C. Castelluccia, and C. Díaz, "The leaking battery—A privacy analysis of the HTML5 battery status API," in *Data Privacy Management and Security Assurance* (Lecture Notes in Computer Science), vol. 9481, J. García-Alfaro, G. Navarro-Arribas, A. Aldini, F. Martinelli, and N. Suri, Eds. Cham, Switzerland: Springer, 2015, pp. 254–263, doi: 10.1007/978-3-319-29883-2_18.
[28] L. Olejnik, S. Englehardt, and A. Narayanan, "Battery status not included: Assessing privacy in web standards," in *Proc. Int. Workshop Privacy Eng.*, in CEUR Workshop Proceedings, vol. 1873, J. M. del Álamo, S. F. Gürses, and A. Datta, Eds. CEUR-WS.org, 2017, pp. 17–24. [Online]. Available: http://ceur-ws.org/Vol-1873/IWPE17_paper_18.pdf and http://ceur-ws.org/HOWTOSUBMIT.html#REFERENCE
[29] Battery Status API. (2016). *World Wide Web Consortium (W3C)*. [Online]. Available: https://www.w3.org/TR/2016/PR-battery-status-20160329/#security-and-privacy-considerations
[30] C. Lamb and S. Zacchiroli, "Reproducible builds: Increasing the integrity of software supply chains," *IEEE Softw.*, vol. 39, no. 2, pp. 62–70, Mar. 2022.
[31] F. L. Haddi and M. Benchaïba, "A survey of incentive mechanisms in static and mobile P2P systems," *J. Netw. Comput. Appl.*, vol. 58, pp. 108–118, Dec. 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804515002106
[32] M. Allahbakhsh, A. Ignjatovic, B. Benatallah, S.-M.-R. Beheshti, E. Bertino, and N. Foo, "Collusion detection in online rating systems," in *Web Technologies and Applications* (Lecture Notes in Computer Science), Y. Ishikawa, J. Li, W. Wang, R. Zhang, and W. Zhang, Eds. Berlin, Germany: Springer, 2013, pp. 196–207.
[33] G. Ciccarelli and R. Lo Cigno, "Collusion in peer-to-peer systems," *Comput. Netw.*, vol. 55, no. 15, pp. 3517–3532, Oct. 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128611002581
[34] J. Benet, "IPFS–content addressed, versioned, P2P file system," 2014, *arXiv:1407.3561*.
[35] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Trans. Inf. Theory*, vol. IT-31, no. 4, pp. 469–472, Jul. 1985.
[36] E. Rescorla, *Diffie-Hellman Key Agreement Method*, document RFC2631, IETF, 1999.
[37] V. G. Martínez, L. H. Encinas, and C. S. Ávila, "A survey of the elliptic curve integrated encryption scheme," *J. Comp. Sci. Eng.*, vol. 2, no. 2, pp. 7–13, 2010.
[38] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, "ACSL: ANSI/ISO C specification language," CEA-LIST and INRIA, Palaiseau, France, Tech. Rep., Version 1.17, 2021.
[39] P. Baudin, F. Bobot, L. Correnson, Z. Dargaye, and A. Blanchard, *WP Plug-in Manual*, CEA LIST, Université Paris-Saclay, Paris, France, 2021.
[40] L. Merkle, M. Pöthig, and F. Schmid, "Estimate e-Golf battery state using diagnostic data and a digital twin," *Batteries*, vol. 7, no. 1, p. 15, Feb. 2021. [Online]. Available: https://www.mdpi.com/2313-0105/7/1/15
[41] L. Petrovic. (2022). *Evdash*. [Online]. Available: https://github.com/nickn17/evDash
[42] J. Shen, Y. Li, Y. Zhou, and X. Wang, "Understanding I/O performance of IPFS storage: A client's perspective," in *Proc. Int. Symp. Quality Service*, Jun. 2019, pp. 1–10.
[43] C. Dong, Y. Wang, A. Aldweesh, P. McCorry, and A. van Moorsel, "Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, Oct. 2017, pp. 211–227, doi: 10.1145/3133956.3134032.

[44] S. Friebe, I. Sobik, and M. Zitterbart, "DecentID: Decentralized and privacy-preserving identity storage system using smart contracts," in *Proc. 17th IEEE Int. Conf. Trust Secur. Privacy Comput. Commun.*, Aug. 2018, pp. 37–42.
[45] S. Friebe, P. Martinat, and M. Zitterbart, "Detasyr: Decentralized ticket-based authorization with sybil resistance," in *Proc. IEEE 44th Conf. Local Comput. Netw. (LCN)*, Oct. 2019, pp. 60–68.
[46] O. Poupko, G. Shahaf, E. Shapiro, and N. Talmon, "Building a sybil-resilient digital community utilizing trust-graph connectivity," *IEEE/ACM Trans. Netw.*, vol. 29, no. 5, pp. 2215–2227, Oct. 2021.
[47] T. Rathee and P. Singh, "A systematic literature mapping on secure identity management using blockchain technology," *J. King Saud Univ.-Comput. Inf. Sci.*, vol. 6, no. 5, pp. 86–91, Mar. 2021.
[48] S. Dupont, G. Ginis, M. Malacario, C. Porretti, N. Maunero, C. Ponsard, and P. Massonet, "Incremental common criteria certification processes using DevSecOps practices," in *Proc. IEEE European Symp. Secur. Privacy Workshops (EuroS&PW)*, Sep. 2021, pp. 12–23.

OLIVER STENGELE received the degree in applied computer science from Heidelberg University, Germany, in 2016. Since 2017, he has been working as a Staff Member with the Decentralized Systems and Network Services Research Group, Karlsruhe Institute of Technology. His research interests include decentralized software identity management, access control, threshold cryptography, and game theory.

CHRISTINA WESTERMEYER received the B.Sc. degree in information systems and the M.Sc. degree in computer science from the Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany, in 2018 and 2021, respectively. Since 2021, she has been a Staff Member with the Decentralized Systems and Network Services Research Group, Karlsruhe Institute of Technology. Her current research interests include decentralized software identity and access management within the context of automotive software systems.

HANNES HARTENSTEIN (Member, IEEE) received the Diploma degree in mathematics and the Ph.D. degree in computer science from Albert-Ludwigs-Universität, Freiburg, Germany. He was a Senior Research Staff Member with NEC Europe. He was appointed as the Chair of decentralized systems and network services (DSN), in October 2003, and has directed the DSN Research Group, Karlsruhe Institute of Technology, since 2003. His research interests include decentralized and distributed systems, information security, and information technology management.