Coupling Smart Contracts: A Comparative Case Study

Sebastian Friebe, Oliver Stengele, Hannes Hartenstein and Martina Zitterbart
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
friebe@kit.edu, oliver.stengele@kit.edu, hannes.hartenstein@kit.edu, zitterbart@kit.edu

***Abstract*—When software systems become more complex, it can be advantageous to partition their code into multiple, separate components. In this work, we examine how multiple smart contracts can be coupled to work together. When coupling smart contracts, different design approaches are possible with their own advantages and disadvantages. As an example, we couple two smart contract applications on the Ethereum blockchain: Palinodia and DecentID. Palinodia can be used to ensure the integrity of downloaded executable binaries by checking their hashes against the hashes stored in the blockchain. To make sure that not everyone can modify the data stored on the blockchain, an identity management system is required. This task is fulfilled by DecentID, which provides decentralized identities stored as smart contracts on the blockchain. We evaluate approaches of coupling these two applications and discuss their benefits and drawbacks for this use case.**

***Index Terms*—smart contract, Ethereum, coupling, blockchain**

I. Introduction

Over the past years, smart contracts have found applications in many different fields.The currently most popular platform by market cap for such applications is Ethereum[1]. Similar to conventional software engineering, it has since become clear that not every application can or should be entirely self-contained and independent. A reason for this that is particular to platforms like Ethereum is the fact that storage for both contract logic and data incurs immediate costs. Consequently, patterns such as library smart contracts or defined interfaces have been established. However, due to the speed at which this development environment evolves, consensus over such interoperability measures can lag behind the need for them.

One such opportunity for interoperability concerns identity management in access control applications. It would be rather costly and cumbersome for every application to deploy and maintain their own identity management. Additionally, it seems plausible that externally vetted, reusable and modular identity management solutions will emerge, not only to save costs but also to provide some resilience against Sybil attacks. However, a standardized contract interface for identity management systems has not been defined yet.

This work was supported by funding of the Helmholtz Association (HGF) through the Competence Center for Applied Security Technology (KASTEL).

[1]https://coinmarketcap.com/historical/20210704/

978-1-6654-3924-4/21/$31.00 ©2021 IEEE

In this paper, we examine the case of two independently developed smart contract applications that originated from research projects: Palinodia [1], a role-based access control application to manage the publication and revocation of integrity-protecting information for software binaries; and DecentID [2], a modular, self-sovereign identity management system. We assume that both applications are deployed by the same user[2], and should be modified to interact with each other. In the absence of a standardized interface, we explore how these smart contract applications can be coupled and examine how these solutions compare on various evaluation criteria.

In Section II, we present the projects Palinodia and DecentID, which our case study is based on. Afterwards, we examine how these two smart contract applications can be coupled in Section III. The advantages and disadvantages of the coupling approaches are evaluated in Section IV. After going over related work in Section V, we discuss our findings in Section VI and conclude in Section VII.

II. Palinodia and DecentID

Palinodia [1] is a blockchain-based system to verify the integrity of downloaded binaries. Developers publish hashes of their binaries on Ethereum, ensuring that users can access and verify them before running the binaries. The identities of developers are currently maintained in a rudimentary identity management system internal to Palinodia. Consequently, these identities can only be used with Palinodia instances. To improve upon this state, the identity management system *DecentID* [2] should be coupled with Palinodia. It manages its identities on the blockchain, allowing Palinodia as well as other blockchain-based systems to use and rely on them.

A. Palinodia

The integrity of binaries is of paramount importance to the integrity and safety of systems that execute them. While centralized solutions for binary integrity protection exist, they require trust into and availability of a service provider. To avoid this single point of failure, Palinodia uses the Ethereum blockchain to allow software developers to store hashes of their published binaries in a self-sovereign way.

[2]Within our use case, the term “users” refers to software developers and maintainers using DecentID as their identity representation and Palinodia to establish and manage identities for their software. For the purposes of this work, end users are out of scope.

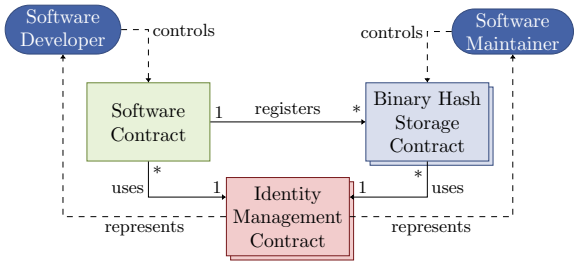

Fig. 1. Overview of Palinodia smart contracts and their mutual relations.

1) Design: Palinodia consists of three kinds of smart contract as depicted in Figure 1: The *Software* contract establishes a software identity, a number of *BinaryHashStorage* contracts that store hashes of a particular subset of binaries of said software, and a rudimentary identity management system using *IdentityManagement* contracts. Together, these contracts codify role-based access control on a per-software hierarchy from a root software identity, through one or more intermediary identities, down to individual binaries.

For the sake of completeness, it should be noted that all Palinodia contracts employ the concept of a special root owner as a recovery mechanism of last resort. The associated private keys are meant to be stored offline for emergency cases only. This work mainly focuses on the identity management for normal day-to-day use.

From the use case, it should be clear that the usefulness of the information stored on the Ethereum blockchain is inherently tied to the corresponding access control. A binary hash representing an endorsement is only useful if the rightful party authored it. Consequently, identity management as a basis for access control is indispensable.

a) Software contract: For every software using Palinodia, one Software contract serves to establish its root identity. It is the central management for the software on the blockchain and stores references to BinaryHashStorage contracts. Additionally, developers for this software are registered in a linked IdentityManagement contract. Developers registered in this way can add and remove BinaryHashStorage contracts to convey or revoke the ability to distribute binaries of the software to the corresponding maintainers.

b) BinaryHashStorage contract: Multiple BinaryHashStorage contracts can be used for each software project to represent different software versions for different device types or operating systems, for example. Similar to the developers in the Software contract, a number of maintainers are registered in the respectively linked IdentityManagement contracts. Maintainers are able to add new binary hashes to the list stored in the BinaryHashStorage contract. When they do so, users of the software can verify the hash of the downloaded binary and ensure that the binary has not been modified by the distributing server. If a particular binary is no longer safe to use, e.g., because critical security bugs were found, its hash can be revoked, ensuring that future validations of the binary by the software user will fail.

c) IdentityManagement contract: Basically, the IdentityManagement contracts each store a list of Ethereum public keys. Each public key represents an authorized user in Palinodia. Depending on which contract the IdentityManagement contract is linked from, the users are authorized either as software developers or maintainers. In this paper, we present and evaluate two approaches to obviate IdentityManagement contracts in Palinodia and replace them with DecentID.

B. DecentID

For many services on the Internet, digital identities are needed. Management of these identities on the Internet is normally done by identity providers. These store all data linked to the identity of the user and can allow them to use the identity in different contexts. *DecentID* [2] is a decentralized identity management system based on Ethereum smart contracts. It leverages security guarantees provided by the underlying blockchain to create a trustworthy system, without relying on a centralized service provider. In the following, only the parts of DecentID relevant for this paper are described.

DecentID allows its users to create so-called *shared identities* by themselves. These identities are stored on Ethereum and consist of a smart contract, representing the identity itself, and a number of attributes augmenting the identity with additional information. For example, in combination with Palinodia, such an attribute could describe the identity as belonging to a developer of a certain software. Initially, the created identity and its attributes are only accessible to the user itself. However, the user is able to share the identity with other users, allowing them to read the attached attributes and attach further attributes as well.

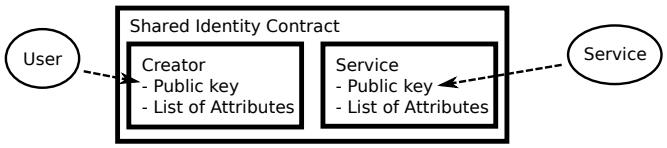

Fig. 2. Shared Identity Contract of DecentID

Within DecentID, a user is represented by its *Shared Identity Contract*, a simplification of which is presented in Figure 2. Inside of the contract, the public key of an asymmetric key pair representing the user is stored. With this key pair the user can digitally sign data. In combination with the shared identity contract, this allows other users or services with access to the shared identity contract to verify the signature and make sure that it was actually signed by the creator of the shared identity contract. To make the identity useful, a list of arbitrary attributes can be attached to it. Since storing data within Ethereum is expensive, these attributes can also be stored externally and referenced by the smart contract. If a service is allowed to access a shared identity contract, the service can add further attributes to the shared identity as well. This way, the service can, e.g., grant the user permissions to use certain features of it.

		Palinodia	
		In development	Deployed
DecentID	In development	Proxy (B) Adapting Palinoda (C) Adapting DecentID (D)	Proxy (B) Adapting DecentID (D)
	Deployed	Proxy (B) Adapting Palinoda (C)	No Coupling (A) Proxy (B)

Fig. 3. Possible approaches for coupling Palinodia and DecentID based on their deployment status.

III. Coupling Palinodia & DecentID

When coupling two smart contracts, using a standardized interface is the preferable approach. However, such a contract interface has not yet been defined for identity management on the Ethereum blockchain. Until such an interface is defined, contract developers have to define their own interfaces while taking advantages and disadvantages of the possible approaches into consideration.

Our use case includes an access-control application and an identity management system, both of which are employed by the same user. As such, we are not concerned about users exploiting their own applications through a coupled contract. In our case, users have a vested interest in their application and their identity representation working together properly. We expand on this aspect further in the discussion.

Generally speaking, the property of smart contracts being immutable once deployed leads to a strong concept of precedence when it comes to coupling them together. The contract that is deployed first basically defines the interface to be used, and the contract application being deployed afterwards will either have to adhere to it or do without it. In the case that two contract applications are already deployed, the coupling has to be achieved through a newly deployed proxy contract that mediates between incompatible interfaces. In our case, since neither Palinodia nor DecentID are deployed in a practical environment, we can therefore examine these scenarios on equal ground.

In this case study, we exemplarily investigate four coupling approaches for Palinodia, as seen in Figure 3: Using the existing identity management integrated into Palinodia (i.e., no coupling); using a proxy to communicate with DecentID indirectly; modifying Palinodia contracts to work directly with DecentID; or modifying the contracts of DecentID to offer the functionality required by Palinodia. The second and third approaches are depicted in Figure 4 and are evaluated in the next section and compared to the first approach. As reasoned below, modifying DecentID is not evaluated. The code used for authorizing users in the evaluated approaches can be found in the appendix.

When DecentID is used to manage the identities, the permission to use Palinodia is stored as an attribute within a SharedIdentityContract. Which users are granted authorization and how to ensure that a pseudonymous blockchain identity really is controlled by a trustworthy user, is out of scope for this paper.

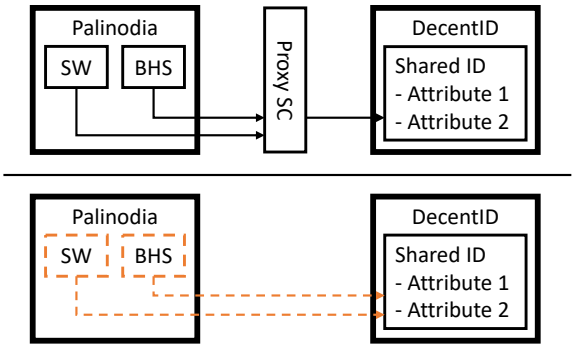

Fig. 4. Coupling per proxy (top) or by modifying Palinodia (bottom) when coupling Palinodia and DecentID. Modified parts are depicted with dashed orange lines.

A. No Coupling

In its original design, Palinodia contains a rudimentary integrated identity management system. Its functionality is rather limited, since its sole functionality is to store a list of permitted public keys. No further data can be assigned to the registered public keys, and using the "identities" in other contexts is not a design goal. As such, it presents the most efficient option to provide the identity service required for Palinodia, but the identities are not available to be used for other purposes, making it inefficient in a multi-application context.

B. Using a Proxy Contract

If both contracts are already deployed or are not supposed to be modified, a proxy contract can be used. The interface of the proxy contract offers the same functions as the integrated identity management of Palinodia, but instead of implementing the functionality itself, it calls the functions of an external identity management system to achieve the necessary functionality. If required, multiple function calls can be executed by the proxy contract to fulfill a request, and manipulating the format of the data before passing them on is possible as well. As such, a proxy contract could be used as a solution if the other contracts cannot be modified. Through different proxy contracts, Palinodia could be coupled with different identity management systems, as long as they support the desired functionality in some way.

It is worthwhile to note an unfortunate naming collision between the Ethereum smart contract community and conventional software engineers: What we call a proxy contract corresponds to the adapter pattern as described by the gang of four [3] whereas their proxy pattern would not be useful in a smart contract environment due to cost constraints. In a sense, established knowledge in software engineering comes into conflict with the new constraints and challenges of blockchains and smart contracts. As the capabilities of blockchains and the complexity of smart contracts grow over time, more and more conventional software engineering patterns will likely become relevant or applicable.

C. Modifying Palinodia

To use another identity management system, e.g. DecentID, with Palinodia directly, modifications are required. In most cases, the function calls Palinodia needs to execute as part of an access control decision are not the same as the function calls the identity management system supports. For example, Palinodia expects from its identity management system to support the function *checkIdentity(address _addr) returns (bool)*. Without the additional domain knowledge that the identity management system only maintains identities of permitted software developers, the expected functionality of this function is unclear. Thus, it is not a useful function interface for a general purpose identity management system, where identities for multiple purposes are maintained. Especially when the identity management system has already been deployed, its interface can no longer be modified. Consequently, Palinodia has to be modified to call the more generalized functions offered by the identity management system.

D. Modifying DecentID

Another approach would be to modify the identity management system instead of modifying Palinodia. One use case of this approach would be if Palinodia would have already been deployed and the identity management system could still be tailored to match. However, this would result in application-specific code to be integrated into the identity management system. In our case, specific functions that check whether a user is a developer or a maintainer would have to be added. These functions would not be useful for other applications but would clutter its function interface and increase the deployment costs for these contracts. Additionally, this approach is not possible when coupling Palinodia and DecentID: DecentID uses one shared identity contract per user, while Palinodia's contracts store the contract address of a single contract that manages all permitted identities. Consequently, this approach is not evaluated in this case study.

IV. Evaluation

In the following, the design approaches described in Section III are evaluated. We implemented two design approaches for coupling Palinodia and DecentID, and are evaluating the respective influence of these approaches as well as of the integrated identity management system on the following criteria:

- **Security dependency** Calling functions on other contracts can introduce new vulnerabilities, even when the calling contract itself can be considered secure.
- **Cost** Induced financial costs, which can be divided into deployment and operational costs of additional code.
- **Implementation** While reducing the implementation effort by using existing contracts, coupling them also requires additional code to be written.
- **Interoperability** When coupling contracts, an interface should be designed that can be used with other contracts as well.

A. Security dependency

In general, the properties of the blockchain ensure that a smart contract is executed as written. Therefore it is crucial to ensure that contracts are correct and secure before deploying them to the blockchain. However, every increase in complexity bears the risk of introducing vulnerabilities. As with conventional software engineering, simpler solutions for coupling smart contract applications are desirable.

To call another contract, its address has to be known. One variant is that the address of the called contract is stored in the calling contract itself, either already added in the source code, at deployment, or later on by a specific setter method. The other variant is that the address is given to the contract as a function parameter specifically for that execution of the function. This corresponds to the dependency injection pattern in conventional software engineering: external code is provided to a function as a requirement to execute its functionality. Where and by whom the address is provided has a significant influence on the security of the system.

The manner in which the address of the identity management system is stored in our case study depends on the design approach. When Palinodia uses its integrated identity management system, the address of the used contract is set when deploying Palinodia but can also be reset by authorized users at a later time. Since the integrated identity management system is only a list of permitted public keys, this is sufficient. Consequently, no security vulnerabilities can be introduced by a user provided contract address.

When coupling with DecentID directly, i.e., modifying Palinodia, the address of the used shared identity contract is provided on calling a function of Palinodia, since each user is expected to use their own contract. The Solidity language offers the ability to retrieve the hash value of the smart contract code residing at an address. This way, it can be checked that a valid shared identity contract resides at the called address. Afterwards, it is checked that the owner of the shared identity contract is the caller of the function. This way, the DecentID identities created by different users can be used to access Palinodia while it is still ensured that the contract at the given address behaves as expected and the data returned by it can be trusted.

When using a proxy contract, the address of the shared identity contract is looked up by using the address of the function caller, based on a previously registered mapping in the proxy contract. Palinodia was not supposed to be modified when using a proxy contract, so the address of the shared identity contract could not be passed along to the proxy contract as a function parameter. Still, the hash value of the provided contract address can be checked the same as with direct coupling.

B. Costs

"Cost" in this section refers to the gas costs of deploying and executing contracts on the Ethereum blockchain. A simple scenario has been selected for the evaluation: The necessary contracts of Palinodia are deployed, two users are

created and authorized (one developer and one maintainer), and their authorization is verified. The last step, verifying their authorization, can happen several times over the lifetime of a Palinodia instance, while the other steps are comparatively rare. Whenever a new software version is published, the maintainer needs to publish new hashes, and possibly revoke old hashes, stored within the smart contracts of Palinodia. Before they are permitted to do so, the authorization of the users is checked by the smart contracts. Contrary to that, the creation of the smart contracts for Palinodia is only done once per software identity that should be managed with Palinodia. Similarly, each developer or maintainer using Palinodia only has to create a single DecentID identity for using the system. However, deploying smart contracts to the blockchain incurs much higher gas costs than executing code.

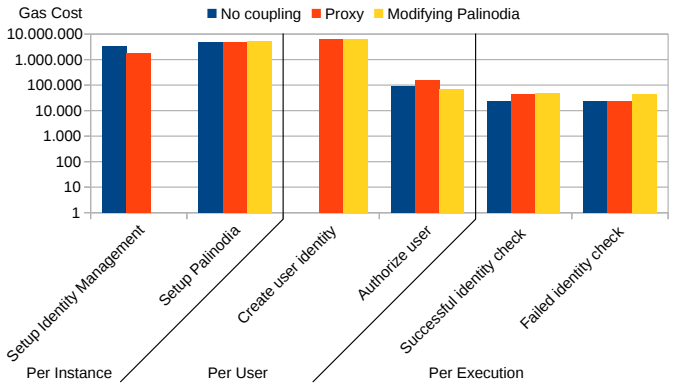

Fig. 5. Measured costs of different actions performed within the system

Figure 5 shows the measured costs of different actions performed within the system. The two measurements for setup operations represent costs that only occur once per instance of Palinodia.

- **Setup Identity Management** The cost to setup the identity management, either the integrated system of Palinodia or the proxy to use DecentID. Most of the costs are the smart contracts being deployed to the blockchain. DecentID itself does not incur any costs at this time, since there are no central management smart contracts. Smart contracts representing user identities are created when a user representation is required.
- **Setup Palinodia** The costs of deploying the smart contracts of Palinodia, i.e., one instance of the Software contract and the BinaryHashStorage contract each. The costs are slightly higher for direct coupling between the systems, since additional code is required to adapt to DecentIDs interface.

Authorizing users to modify the state of Palinodia can be divided into two steps: Creating the digital identities and authorizing them. As such, both have to be done each time a new user should be authorized to use Palinodia.

- **Create user identity** Creating a dedicated user identity only needs to be done when DecentID is used. In that case, a new smart contract representing a shared identity between the user and Palinodia is deployed. If there already exists a shared identity that should be used for Palinodia, these costs do not apply. When instead of DecentID the integrated identity management system of Palinodia is used, no explicit identities are used for unauthorized users, resulting in no costs.
- **Authorize user** Users need to be authorized to modify the state of Palinodia. For the integrated identity management system, this means that the public key of the user is added to a list of permitted users. When DecentID is used the authorization is represented by an attribute attached to the shared identity, which could also be used in other contexts. The approach using a proxy is slightly more expensive, since for technical reasons the mapping between the user's public key and its shared identity has to be stored in the proxy.

Checking the authorization of users happens each time a user wants to manipulate the state of Palinodia, e.g., to add a new binary hash. As such, these costs appear many times per participating user and should therefore be reduced as much as possible.

- **Successful identity check** A successful identity check is a requirement for manipulating the state of Palinodia. For the integrated identity management system, the check only requires determining whether the public key in question is registered in a list in the smart contract. When DecentID is used, a shared identity controlled by this public key is required which must have an attribute confirming the authorization.
- **Failed identity check** These costs occur when the authorization fails, e.g., due to providing wrong parameters or not being authorized. When trying to authorize a new user an identity check is executed first, which has to fail for the user to become authorized (to avoid duplicated authorization). In our case, this is slightly cheaper when using the proxy compared to the direct coupling, since the previous registration of users in the proxy allows faster negative authorization decisions.

Complete setups: The previous measurements regard the costs for each action separately. Based on these measurements, the deployment costs of complete Palinodia setups have been calculated. Included is the deployment of two contracts for Palinodia (one software and one binary hash storage contract), the contracts for the identity management system (either the integrated one or the proxy) and the contracts for the DecentID identities.

As can be seen in Figure 6, the costs for the integrated system are only slowly growing since authorizing another user only requires adding them to a list. When DecentID is used, the costs are mostly linked to the creation of the identities if they need to be created for the users. As seen in the columns regarding "existing users", if the shared identity contracts already exist and the authorization is added to them, the costs are drastically reduced. When modifying Palinodia, the costs are even lower than those of the integrated system.

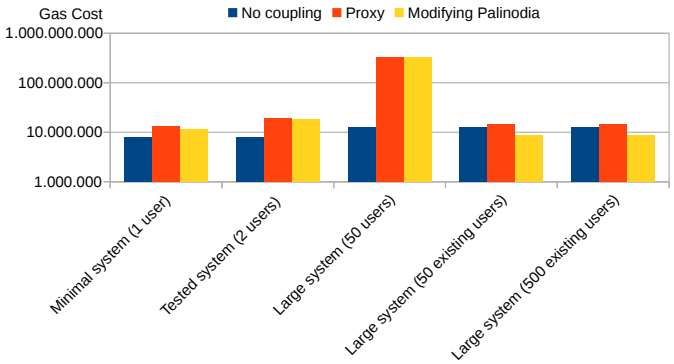

Fig. 6. Costs of using the systems with different amounts of users

C. Implementation

Depending on how many contracts have to be modified, the implementation effort varies. If only the source or the destination contract has to be modified, less work is required than if both contracts have to be modified. On first glance, creating a proxy contract requires more work than modifying existing contracts. However, the additional work required is relatively small since the largest part, the modification between the non-fitting interfaces, has to be done for all approaches.

When modifying Palinodia to work with DecentID, part of the functionality of the integrated identity management system had to be moved to Palinodia itself. Specifically, the mapping between roles in Palinodia and the generic attribute storage in DecentID has to be implemented. Additionally, the code had to be extended to interact with the more complicated data storage provided by DecentID. When implementing the proxy contract in our use case, more work had to be done instead of simply moving the adapting code from Palinodia to the proxy contract. Since the code of Palinodia was not supposed to be modified, state had to be held within the proxy contract to link the caller addresses forwarded by Palinodia to the shared identity contracts used by DecentID.

D. Interoperability

Considering the aforementioned costs for deploying additional contracts, it is desirable for a coupling solution to be reusable and flexible. Similarly, any changes to contracts should serve to increase options for users and not sacrifice one option for another.

In the original design of Palinodia, the identity management contract contains a specific function that checks the authorization for a given address. This is the minimal interface required for Palinodia's authorization check, but requires support of the identity management system for the specifics of Palinodia. Contrary to that, DecentID only offers generic access to stored attributes. In the design approaches of our case study Palinodia, respective the proxy contract, had to be modified to support this generic interface. While this requires a modification of the smart contract, it results in a more generic interface between the contracts that could also be used for interacting with other identity management systems.

E. Summary

The results of the evaluation are depicted in Figure 7. As can be seen, an ideal approach for coupling does not exist, each approach comes with its own advantages and disadvantages with regard to our evaluation criteria. However, we also observed that the outcome of our evaluation strongly depends on the specific use case and the implementation used. As such, the same evaluation criteria applied to other contracts would most likely lead to a different assessment.

V. Related Work

Similar to traditional software engineering, design patterns have been developed for smart contracts. Those are supposed to address frequent design problems or security vulnerabilities by using established approaches. Also, patterns for interacting with external data sources are considered. Some design patterns for Ethereum smart contract are listed, categorized and described in [4], [5]. However, these patterns only deal with designing single smart contracts.

Only few design patters deal with the interactions between multiple contracts. One such design pattern mentioned in [6] is the *Contract Mediator*, which, similar to the proxy contract used in our evaluation, is used to avoid tight coupling between interacting contracts. Since the interactions are mostly application specific, most patterns for coupling only deal with certain problems, e.g., the secure transfer of Ether between contracts or accounts.

A standardized smart contract interface designed for a specific use case is described in EIP-20 [7]. There, an interface is specified that can be used to implement a standard compliant token on the Ethereum blockchain. To the best of our knowledge, a similar EIP for identity management does not exist yet. A number of drafts exist [8]–[10], which are proposing different interfaces for identity management systems, each with different goals and shortcomings. It remains to be seen whether one of these or another approach becomes a standard.

Unrelated to blockchains, decentralized identifiers (DIDs) are an emerging standard to allow users to maintain self-sovereign identities [11]. Basically, a user creates a DID document which collects attributes describing them, e.g., public keys to authenticate the user. The idea is that the user keeps control of their identity and can prove ownership of it without relying on third parties. For identities on Ethereum, the public key of a key pair can be used for both identification and authentication, and can be augmented further with additional data stored in smart contracts. As such, DecentID identities are similar to DID documents.

VI. Discussion

While smart contract capable blockchains like Ethereum present a unique environment for designing and deploying applications due to their inherent cost structure, our case study shows that some conventional software engineering paradigms regarding modular designs and reusable components still hold. In our case study, identity management neatly demonstrates that individual, custom built solutions to common problems

		Criteria			
		Security dependency	Cost	Implementation	Interoperability
Approaches	No Coupling	No malicious addresses possible	Low deployment cost, low execution costs	Implementing own identity management system required	Coupling only with own system
	Proxy Contract	Hash of contract at target address can be checked	Low to high deployment cost, low execution costs	Adaption needed	Coupling with any system possible
	Modifying Palinodia	Hash of contract at target address can be checked	Low to high deployment cost, low execution costs	Adaption needed	Coupling with other identity management systems possible

Fig. 7. Evaluation results

may be efficient in a per application sense, but extending one's view to include multiple applications that could reuse these solutions shows their downsides. Since neither Palinodia nor DecentID are currently in practical use, modifying the former to best work with the latter is the preferable option in our case. Ideally, if a common interface for identity management on Ethereum would exist, both applications should be adjusted to adhere to it in order to provide the greatest amount of convenience and flexibility to users while keeping operational costs minimal.

In the meantime, proxy contracts appear to be a viable stopgap solution with certain advantages and drawbacks to consider. Their biggest strength is that they can be used to couple two already deployed applications. The downsides to this broad applicability are their additional deployment and execution costs as well as their lack of flexibility. In our case, the proxy contract between Palinodia and DecentID has to keep a mapping from the calling address to the target shared identity contract. As such, multiple instances of the same proxy contract would have to be deployed for different Palinodia instances. This mapping is a critical component in the access control mechanism of the application in question. For that reason, developers or maintainers of distinct Palinodia instances are not likely to trust each other with the management of multi-tenant proxy contracts.

With proxy contracts in mind, smart contract application developers could adapt their designs such that stateless proxy contracts could be used. In this case, one proxy contract could serve to couple multiple instances of applications and identity management systems together, thus greatly amortizing deployment costs. This could be a very cost efficient and flexible approach in case no dominant identity management interface emerges.

As mentioned previously, our case study presents a rather simple case where both the application and identity management are under the control of the same user, who consequently has a vested interest in the coupling working correctly. This basic assumption simplifies the coupling in certain ways. For example, on the side of Palinodia, the smart contract does not have to check whether or not a compatible identity management contract is actually deployed at the supplied address, since the user should have already taken care of that beforehand. However, if the application is autonomous, like the notoriously attacked DAO contract, or under the control of a separate party, security becomes paramount. A malicious user could attempt to attack the application through a coupling with a deliberately crafted identity management contract to steal funds or gain control of the application.

Ultimately, we can only echo the competing advances in the Ethereum ecosystem [8]–[10] to agree on a well-defined interface that both applications and identity management solutions can support. Such an interface would offer the greatest amount of efficiency, flexibility, and user convenience in addition to long-term stability.

While we focus on a very particular coupling issue with smart contracts, the problem generalizes in at least two ways: Coupling two or more smart contract applications will likely become relevant in contexts other than identity management and access control. Similarly, other challenges that have already been tackled and solved in conventional software engineering [3] will have to be solved again in the environment of blockchain systems with their cost structures and other limitations.

VII. Conclusion

In this paper, we examined the problem of coupling a smart contract application with an appropriate identity management system in a case study. This is a problem that is most likely going to become increasingly relevant as smart contract applications grow in number and complexity and the demand for more versatile, robust, and convenient identity management solutions increases. From our case study, one can draw insights for designing both future applications and identity management systems with coupling as a priority rather than an afterthought. We showed that proxy contracts can serve as a stopgap solution until a common interface is defined and broadly adopted. It will be interesting to see how the tendency to define comprehensive and broad common interfaces clashes with the cost structure of Ethereum, where unnecessary code to satisfy an interface causes higher deployment costs.

Acknowledgments: We would like to thank the anonymous reviewers for their very helpful feedback.

References

[1] O. Stengele, A. Baumeister, P. Birnstill, and H. Hartenstein, "Access control for binary integrity protection using ethereum," in *Proc. of the 24th ACM SACMAT*. New York, NY, USA: ACM, 2019, p. 3–12. [Online]. Available: https://doi.org/10.1145/3322431.3325108

[2] S. Friebe, I. Sobik, and M. Zitterbart, "DecentID: Decentralized and Privacy-Preserving Identity Storage System Using Smart Contracts," in *2018 17th IEEE Int. Conf. On Trust, Security And Privacy In Computing And Communications/ 12th IEEE Int. Conf. On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, 2018, pp. 37–42.
[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.
[4] M. Wöhrer and U. Zdun, "Design patterns for smart contracts in the ethereum ecosystem," in *2018 IEEE Int. Conf. on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018, pp. 1513–1520.
[5] X. Xu, C. Pautasso, L. Zhu, Q. Lu, and I. Weber, "A pattern collection for blockchain-based applications," in *Proc. of the 23rd European Conf on Pattern Languages of Programs*, ser. EuroPLoP '18. New York, NY, USA: ACM, 2018. [Online]. Available: https://doi.org/10.1145/3282308.3282312
[6] Y. Liu, Q. Lu, X. Xu, L. Zhu, and H. Yao, "Applying design patterns in smart contracts," in *Blockchain – ICBC 2018*, S. Chen, H. Wang, and L.-J. Zhang, Eds. Cham: Springer International Publishing, 2018, pp. 92–106.
[7] V. B. Fabian Vogelsteller. (2015-11-19) Eip-20: Erc-20 token standard. Ethereum Improvement Proposals, no. 20. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20
[8] T. Y. Fabian Vogelsteller. (2017-10-02) Erc-725: Smart contract based account. [Online]. Available: https://github.com/ethereum/EIPs/issues/725
[9] J. T. Pelle Braendgaard. (2018-05-03) Erc-1056: Lightweight identity. [Online]. Available: https://github.com/ethereum/EIPs/issues/1056
[10] P. Braendgaard. (2019-03-03) Erc-1812: Ethereum verifiable claims. [Online]. Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1812.md
[11] W3C. (2021-06-16) Decentralized Identifiers (DIDs) v1.0. [Online]. Available: https://www.w3.org/TR/did-core/Overview.html

APPENDIX

In the following, code segments relevant for authorization checks are displayed for the three approaches evaluated in this paper. Reasoning why these approaches have been selected can be found in Section III.

A. No coupling

```
function checkIdentity(address _addr)
    public view returns(bool) {
  return (arr_idents[map_idents[_addr].array_index]
      == _addr);
}
```

Fig. 8. Code in Palinodia's IdentityManagement contract used to check the authorization of users.

In the original implementation of Palinodia, as displayed in Figure 8, a special IdentityManagement contract was used for maintaining a list of authorized users. The array *arr_idents* contains a list of all authorized public keys, while the map *map_idents* maps the public keys to an array index as well as potentially to other small attributes. During an authorization check, the given public key is looked up in the map to retrieve the array index. If the array contains the given key at the retrieved index, the user is authorized to perform actions within Palinodia. If the index stored in the map does not point to a matching array entry, e.g., because the entry in the array was removed, the authorization fails.

```
function checkIdentity(address _addr)
    public view returns(bool) {
  SharedIdentityContract sic = map_idents[_addr];
  if (sic == SharedIdentityContract(address(0))) {
    return false;
  }
  if (sic.getCreator() != _addr) {
    return false;
  }
  bytes memory attr = sic.getAttribute(admin,
      addressToHexString(msg.sender));
  return attr.length == 1 && attr[0] == role;
}
```

Fig. 9. Code in the proxy contract for checking authorization with DecentID.

B. Proxy Contract

The code used within the proxy contract is displayed in Figure 9. Since Palinodia was not supposed to be modified for usage of the proxy, the first action is to retrieve the address of the SharedIdentityContract that has been linked to the public key beforehand. Afterwards, it is checked whether the creator of the SharedIdentityContract is the given public key. This protects against attacks where a random public key is registered with an authorized identity contract to gain access to Palinodia. For a successful authorization, the administrator of the proxy contract and Palinodia (stored in *admin*) needs to grant an attribute to the user, which uses the address of the calling Palinodia contract (available in *msg.sender*) as a key. If such an attribute exists and contains the right value (configurable depending on whether developer or maintainer access is required, stored in *role*), the authorization succeeds.

C. Modifying Palinodia

```
function checkIdentity(SharedIdentityContract sic,
    address sender) internal view returns (bool) {
  if (getContractHash(address(sic_root)) !=
        getContractHash(address(sic))
      || sic.getCreator() != sender) {
    return false;
  }
  bytes memory attr = sic.getAttribute(
      sic_root.ownerAddrs(0),
      addressToHexString(address(this)));
  return attr.length == 1 && attr[0] == byte(0x02);
}
```

Fig. 10. Added code in Palinodia (BinaryHashStorage and Software contracts) for checking authorization with DecentID.

When modifying Palinodia to work with DecentID (as displayed in Figure 10), the address of the SharedIdentityContract is passed as an additional parameter. At first, the hash of the contract code stored at the passed address is retrieved. This hash is compared to the known hash of a SharedIdentityContract registered within Palinodia by its administrator. If the hashes match, the passed address really belongs to a SharedIdentityContract and the required behavior can be expected from the contract. The following authorization check is equivalent to using the proxy.