Porthos: Macroprogramming Blockchain Systems

Adrian Mizzi
Department of Computer Science
University of Malta
Msida, Malta
adrian.mizzi.00@um.edu.mt

Joshua Ellul
Department of Computer Science
University of Malta
Msida, Malta
joshua.ellul@um.edu.mt

Gordon J. Pace
Department of Computer Science
University of Malta
Msida, Malta
gordon.pace@um.edu.mt

***Abstract*—The rise of blockchain technology has paved the way for an increasing number of blockchain systems, each having different characteristics. The need for distributed applications that span across multiple blockchain systems is increasing. However, it is currently not possible to write a single-description smart contract which can be compiled to span across multiple blockchain systems.**

In this paper we present PORTHOS, a macroprogramming framework and domain specific language for writing commitment-based smart contracts that span multiple blockchain systems. The language allows programmers to write smart contracts at a higher level of abstraction by composing together contract blocks, without the need to specify how logic should be split across different blockchain instances. A runtime framework, including both on-chain and off-chain functionality, harmonises the features of different blockchain systems as well as enables communication across the smart contracts. A proof of concept, built on the Ethereum and Hyperledger Fabric blockchain systems and extendible to other systems, illustrates the technique and framework. We also show how the PORTHOS language is expressive enough to define a variety of applications.

***Index Terms*—macroprogramming, blockchain, DSL**

I. Introduction

Blockchain technology, a type of Distributed Ledger Technology (DLT), has attracted widespread interest in recent years — different blockchain systems will co-exist and will be used for different purposes by individuals, businesses and institutions. Today's smart contracts are intended to execute on a single blockchain system. We predict that the need for multi-chain distributed applications (DApps) spanning across multiple blockchain systems is going to increase as blockchain technology continues to gain popularity. Interactions between blockchains (interoperability) will be needed to implement new types of applications where assets may be exchanged between participants across different blockchain systems. A single application may handle payments on the Bitcoin network [1], use Ethereum [2] for public interactions and Hyperledger Fabric [3] for specific private point-to-point interactions. Implementing such multi-chain DApps is non-trivial. Good knowledge of at least one smart contract language on each of the target underlying blockchain systems is required, together with a good understanding of features and characteristics. Further, blockchain interoperability is not straightforward and the use of relays, notaries or atomic swaps [4] is required.

The research work disclosed in this publication is partially funded by the Endeavour Scholarship Scheme (Malta). Scholarships are part-financed by the European Union - European Social Fund (ESF) - Operational Programme II Cohesion Policy 2014-2020. *"Investing in human capital to create more opportunities and promote the well-being of society"*.

What is currently missing and desirable, is the ability to write a single smart contract which spans across multiple blockchain systems. This would replace the need to write several smart contracts (one for each blockchain system) and the handling of the communication between them.

We propose a way of addressing this gap through the use of macroprogramming — a technique often used in the domain of IoT and sensor networks [5]–[9]. With macroprogramming, the level of abstraction is increased and the network is programmed as a whole rather than each component individually. The higher level of abstraction allows the programmer to focus on the logic, rather than the details of communication between components. We propose a domain specific language (DSL) for defining commitment-based smart contracts [10]. DSLs provide a higher level of abstraction than general purpose programming languages and are ideal to make it possible to write secure smart contracts in a quick and efficient way.

Using a technique of embedding a domain specific language, we present a framework called PORTHOS[1], to define and execute multi-chain smart contracts. In our proof of concept, we show Ethereum and Hyperledger Fabric as two diverse and interacting blockchain systems — a technique that can be extended to other systems. The Turing-incomplete language allows a programmer to describe commitment-based smart contracts that may span multiple blockchain systems. Our language is inspired from financial contracts of Peyton Jones [11] and the work done with Marlowe [12], but we extend this to contracts that span across multiple chains.

The contribution of this paper is to provide a model in which a commitment-based smart contract can be translated to execute safely on one or more interacting blockchain systems. Our aim is to (i) provide a mechanism to split contract logic on different blockchain systems according to asset location (ii) design a safe and restricted DSL for composing commitment-based smart contracts (iii) define an extensible mechanism to generate code in different target smart contract languages (iv) propose a simple runtime framework to enable chain interoperability.

[1]Available at https://github.com/adrianmizzi/porthos-1

II. Background and Related Work

A smart contract is a program that runs on a blockchain system. The contract can encode a set of rules which determine when and under what conditions transfer of assets may occur.

Languages – The Ethereum Virtual Machine (EVM) is a simple but powerful Turing-complete virtual machine on which EVM byte code can be executed. Solidity [13], the most popular language on Ethereum, is an imperative-style language, where intermediate state is managed explicitly by the programmer. The added expressivity increases the risk of bugs as programs become more complex to reason about and implement as has been shown by high profile heists [14]. Different paradigms have emerged to address the risks associated with unrestricted languages. Explicit state transition languages [15]–[17], use concepts from finite state machines and automata. Functional programming paradigms are used in [18]–[20]. Other techniques used include the use of DSLs [12, 21, 22] where code is generated in existing smart contract languages.

Macroprogramming is a technique mostly used in the domain of sensor networks where the network is programmed as a whole. Three main approaches exist: (i) the programmer has a centralised view of the network and each node can be addressed individually [7, 9] (ii) macroprograms are written from the perspective of individual nodes and all nodes get a copy of the same code [5, 8], and (iii) a macroprogram is written from the network perspective and generated code is different for individual nodes [6, 23].

Interoperability is becoming increasingly important to execute DApps across multiple blockchain systems. Buterin [4] identifies three strategies: (i) hashed time-locks (ii) relay chains (iii) centralised or multisig notary schemes. Hashed time-locks are ideal for swapping assets across separate blockchain systems [24]. Relays or notaries both rely on the presence of a trusted entity. With relays, such as Cosmos and Polkadot, blocks are copied from one blockchain system to another and the receiving blockchain has the capability of inspecting incoming blocks to trigger actions as needed [25]–[27]. In notary schemes, such as the Aion Transwarp Conduit [28], a trusted entity triggers an operation on a blockchain when an event is detected on another blockchain.

III. The Porthos Framework

Traditionally, smart contracts are written to be executed on a specific blockchain system. Interactions between smart contracts located on different systems require complex mechanisms to be implemented. Using a macroprogramming model, we propose to program a network of blockchain systems as a whole, where code is automatically generated to be executed on each blockchain system. A higher abstraction level ensures that the programmer need only focus on the overall logic of the smart contract using only one programming language.

We use techniques from the field of embedded languages to define a domain specific language. We embed our language in Haskell, a pure functional language which gives us several useful features, such as polymorphism, higher-order functions and a strong type system. Our model supports both the macroprogramming aspect of writing smart contracts that run across multiple diverse blockchain systems, and also inherently the ability to generate code for different target contract languages.

Figure 1 illustrates our framework. A macro smart contract is written in our DSEL in a form that can be analysed, translated and deployed to different blockchain systems. The smart contract description first generates an internal representation of the intended contract. With additional information about asset location mapping, the internal representation can be transformed into chunks that need to be placed on the individual blockchain systems according to the assets being used. A first-stage compilation process generates code for each of these chunks into a smart contract language supported by the underlying blockchain system. For example, for Ethereum, the first-stage compilation process generates Solidity and for Hyperledger, Go Chaincode is generated. During a second stage process, the standard compilation and deployment tools for each of the target languages are used to deploy the generated code to the intended blockchain systems.

The ultimate goal of Porthos is to allow programmers to safely write multi-chain smart contracts that are easy to read and hide away the complexities of blockchain interoperability.

A. Multi-chain Support

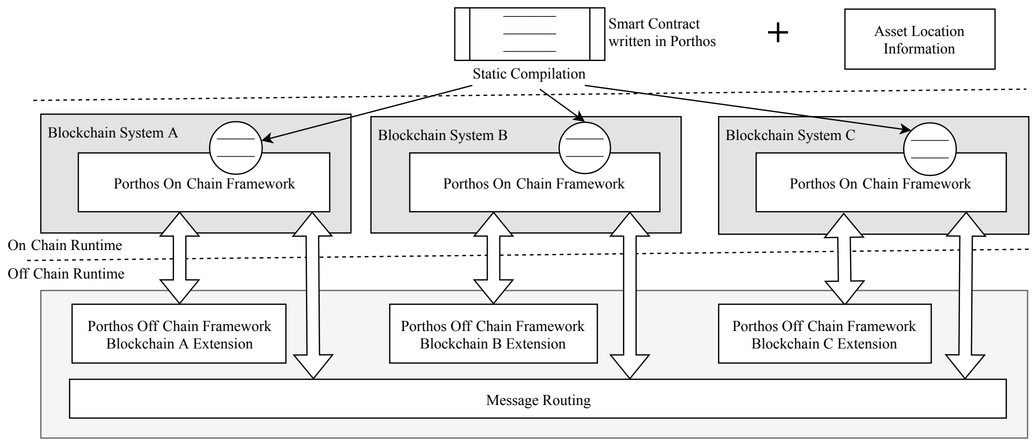
The proposed macroprogramming approach highlights two key challenges — heterogeneity and passiveness.

1) Requirements and Extensions: Porthos supports blockchains which satisfy a minimal set of requirements:

- Smart contracts must have an address and must be capable of ‘holding’ assets transferred to them by users
- Participants must be able to interact with a blockchain system through smart contract functions
- Asset registers must be supported and implementable to track fungible or non-fungible assets

Blockchain systems such as Bitcoin are not supported in Porthos. Bitcoin follows the unspent transaction output (UTXO) model and a smart contract capable of holding assets is not supported. Blockchain systems which satisfy the requirements are supported, however different systems have different features and a solution is needed to harmonise these differences. To address this in Porthos, we use blockchain extensions — a solution made up of on-chain and off-chain components which addresses gaps or differences in the required functionality. The Porthos abstraction model requires a callback-on-timeout mechanism to be able to resume execution in case an expected user interaction is not performed in time. This feature is not natively available on Ethereum and other target blockchain systems although third party extensions exist with this functionality [29, 30].

2) Message Routing: Blockchain systems are unable to communicate in the traditional way as normal systems do. Due to the nature of being passive, a common characteristic



Smart Contract written in Porthos
+
Asset Location Information
Static Compilation
Blockchain System A
Porthos On Chain Framework
Blockchain System B
Porthos On Chain Framework
Blockchain System C
Porthos On Chain Framework
On Chain Runtime
Off Chain Runtime
Porthos Off Chain Framework Blockchain A Extension
Porthos Off Chain Framework Blockchain B Extension
Porthos Off Chain Framework Blockchain C Extension
Message Routing

Fig. 1. Proposed Architecture

of current DLTs, the blockchain systems that we are interested in are unable to actively react to events from other systems. The use of an external party is therefore needed to provide a communication layer between blockchain systems. The PORTHOS framework makes use of an external message router to relay messages between one blockchain system and another. The message router is in the spirit of a notary scheme, where events of interest are captured and actioned upon. The communication layer is lightweight in the sense that there is no knowledge of the smart contract logic being executed — the router listens for events generated by the blockchain systems and triggers other contract functions as instructed by these events. Messages are signed by the originating blockchain system, and validated by the receiving blockchain system before being processed. Duplicate messages are not processed multiple times. This mechanism removes the dependency on the off-chain framework, in that the routing mechanism can be performed by any intermediary or interested party.

B. Code Cuts

Application logic is sliced into different smart contracts and placed on an underlying blockchain system. It may be possible to use different strategies to slice code: (i) an execution-cost optimised strategy — executing code on some blockchain systems may be more expensive than others (ii) a location-based strategy — contract logic is placed on the same blockchain system according to where the asset being handled is located (iii) a programmer tag-based strategy, where the programmer instructs which logic should be placed on which blockchain.

In our proposal, we use a location-based placement strategy as this avoids the added burden on the programmer for tagging code. In the future, we envisage enhancing the strategy to consider both execution cost and user compiler hints.

IV. PORTHOS AS A SMART CONTRACT LANGUAGE

PORTHOS is a domain specific language for composing commitment-based smart contracts [31] — a contract is viewed as a business exchange of commitments which are released or cancelled depending on contract criteria. The abstraction model includes these concepts:

- When a participant *makes a commitment* of an asset towards another participant, the ownership of that asset is transferred to a smart contract and held temporarily.
- A commitment is *released* when the contract transfers the ownership of the held asset to the intended recipient.
- A commitment is said to be *cancelled* when the contract returns a committed asset back to the original owner.

PORTHOS is a continuation-based language embedded in Haskell. Basic language constructs are connected together to form a contract. Haskell's strong type system ensures that only valid contracts can be constructed. Contracts are made up of other contracts in a compositional manner.

As a simple example to introduce the language we show how a simple savings-plan contract is implemented by composing constructs together. The committed assets are released after a specific amount of time.

```
savings :: Participant -> Time -> Contract
savings recipient expiryTime =
    repeatCommit "save" (ETH, isCommitTo recipient)
        (onTimeout expiryTime (releaseAll end))
```

The implementation is made up by combining three basic constructs: `repeatCommit` followed by `releaseAll` and finally `end`. Since PORTHOS is embedded in Haskell, contracts look like Haskell programs. The construct `repeatCommit` causes the progression of the contract to suspend to allow contract participants to make commitments. A filter is used to determine which commitments are accepted by the contract. In this example, valid commitments are in the cryptocurrency Ether (denoted by `ETH`) and must be in favour of a *recipient*. The same contract can be recompiled with different parameters (i.e. recipient) to generate different contracts. Once the timeout period elapses, no more commitments are accepted and execution continues spontaneously — in this example, the contract continues with `releaseAll`, that is, all commitments held in the contract at that point in time are released.

Listing 1 Asset Swap

```
swap :: (Participant, Asset Currency) ->
            (Participant, Asset Currency) -> Contract
swap (p1, a1) (p2, a2) =
          onUserCommit "p1Commit"
            (ETH, (isCommitTo p2 .&. isAsset a1))
            doP2Commit (onTimeout 10 end)
  where doP2Commit =
          onUserCommit "p2Commit"
            (XYZ, (isCommitTo p1 .&. isAsset a2))
            (releaseAll end)
            (onTimeout 20 (cancelAll end))
```

Listing 2 Group Pay

```
groupPay :: [(Participant, Asset Currency)] ->
                Participant -> Contract
groupPay yy recipient =
    allOf (userCommits yy) .>>>.
    ifThenElse
      (countC(allCommitments) .==. liftN(length yy))
      (releaseAll end, cancelAll end)
  where
    userCommits =
        map (\x -> onUserCommit (name (fst x))
          (ETH, txFilter x) end (onTimeout 100 end))
    txFilter (a, b) = isCommitTo recipient .&.
                      isCommitBy a .&. isAsset b
```

The language provides two distinct basic constructs for accepting commitments. The first, `repeatCommit` described earlier, accepts any number of commitments (zero or more) in a given time-window, and the second expects one-and-only-one commitment (`onUserCommit`). In the latter, when a valid commitment is received, the contract continues execution immediately, or if no commitment is made in time, then the contract resumes with a time-out continuation. An atomic swap contract (Listing 1) allows participants to swap assets safely where assets are released once both commitments are made.

Commitments are stored in the smart contract state and can be filtered, counted and summed to determine whether enough assets have been committed. Specific commitments can be cancelled or released — for example, by specific asset type or for a quantity which is smaller than a specific amount.

One of the key benefits of embedding the DSL in Haskell, our host language, is that a smart contract can make use of standard Haskell combined with our contract constructs. In a group pay contract (Listing 2), participants agree to transfer an agreed amount to one participant. Funds are released to the recipient once all commitments have been made. This example shows the use of Haskell's map and lambda expressions to build complex contracts concisely.

V. Use Case: Property Sale

To illustrate the effectiveness of PORTHOS as a multi-chain smart contract language, we present a property sale agreement where a buyer and a seller make an agreement to transfer property in exchange for payment. As described earlier in Section III-A, in PORTHOS, information about asset location is kept separate from the smart contract. This means that contract logic is clearer to the reader, and simpler to write for the programmer. The higher level of abstraction completely omits the details of inter-chain communication.

In our use-case, the property sale is a two-stage process. During the first stage, a buyer and a seller engage in a promise-of-sale agreement — the buyer confirms interest by committing a deposit amount, and the seller promises to sell the property to the buyer. Within a few weeks or months, the buyer must obtain funds to be able to pay the remaining balance. If the buyer is unable to make the payment, the deposit amount is forfeited in favour of the seller and the seller is freed to find a new buyer. However, before the deed is completed, a public notary must submit approval. Should the notary reject the deed, then the promise-of-sale is cancelled and the buyer receives back his deposit.

We identify three participants: the buyer, the seller and the public notary; and three types of assets: (i) a currency asset for payments (ii) a property asset to represent the asset being transferred from the seller to the buyer, and (iii) the public notary's decision of approval or rejection is also modeled as an asset. A complete implementation of the smart contract is shown in [32]. Asset location information is provided during the compilation stage such that the contract logic is placed in the generated code according to which blockchain an asset type is located. Assets may all be located on the same blockchain (in that case, only one smart contract is generated), or alternatively located on different blockchains.

```
propSale (seller, property)
        buyer deposit balance notary
```

The contract is instantiated by providing input values for participants and assets traded. The generated code can then be deployed and instantiated on the respective blockchain systems. If the same contract is to be reused for another property sale between other participants, then the PORTHOS contract is re-compiled from first stage with new input values.

At runtime, the contract progresses through different states — seller commits property, buyer pays deposit and balance, and finally notary approves or rejects the transfer. Commitments cannot be made out of sequence or at the wrong time, and once a commitment is made, it cannot be cancelled. Timeout continuations remind the programmer to define actions in case a commitment is not made.

VI. Evaluation

Expressiveness of Abstraction — We have implemented a range of applications in this paper and in [32] including group payments, asset swapping, property sale, crowd-funding and voting. The model is best used for processes with a finite number of steps. In PORTHOS, user interactions with smart contracts are limited to commitments. Other interactions, such as cancelling or redeeming a commitment, are currently not possible. We believe our limitation on user interactions is not too restrictive on the variety of smart contracts that can be described. The language is Turing incomplete, so applications involving loops are not describable — this is an intentional design decision to ensure all contracts terminate.

Security Analysis — Blockchain systems have a high level of security due to their decentralised and immutable characteristics. However, these are still not completely immune to attacks or weaknesses [33]. In addition, multi-chain applications may introduce new weaknesses:

Single Point of Failure (SPOF) – One of the strengths of blockchain systems is that due to the decentralised nature, no SPOF exists. However, the introduction of an intermediary to route messages may create a SPOF. To mitigate this, the communication layer itself must be decentralised with multiple copies running concurrently. As this may cause the same contract function to be triggered multiple times, a mechanism for filtering duplicate messages must be used.

Third-Party Interference – Intermediaries route messages between blockchain systems. It may be possible for intermediaries to withhold, modify or forge messages. As long as one honest intermediary is available to relay messages, then messages cannot be withheld. To mitigate the risk of modified or forged messages, all communication is signed by the originating system. During initiation stage, cryptographic keys are exchanged such that messages can be verified.

Distributed Transactions – Transactions spanning multiple systems are susceptible to partial failure, causing other parts of the same transaction to be reversed on other systems. If the transaction is not reversed correctly, the systems may end up with inconsistent state. Our framework does not currently handle such errors gracefully, and this is an area to be developed further.

Extensibility — The PORTHOS framework is extensible in different directions. Different asset types are supported as long as a contract interface is implemented. The model can also be extended to support new systems by adding code generation into the target language, and extending the on-chain/off-chain framework. The framework currently generates Solidity code to be executed on multiple Ethereum instances, as well as Go Chaincode for Hyperledger Fabric.

VII. Discussion and Conclusions

In this paper we presented PORTHOS, an embedded DSL framework for describing commitment-based smart contracts that span across multiple blockchain systems. This is to our knowledge the first attempt at providing a macro-level approach for specifying multi-chain smart contracts in a single specification. The closest work to that being presented within this paper include: D'ARTAGNAN for programming IoT devices at a network level [6] and for writing a single macroprogram for blockchain connected IoT devices [23]; Marlowe for specifying financial contracts on Cardano [12].

PORTHOS is designed with safety in mind such that the smart contract programmer is aware of timeout scenarios and must define what happens in these situations. Although the language cannot avoid all types of 'bugs', it does help the programmer to significantly reduce easy-to-forget cases. PORTHOS has shown that by raising the abstraction level, it is possible to separate the complexities of placement and communication from the contract logic such that the programmer needs only to focus on the contract. Through the use of composition there is much to be gained as complex contracts can be made up of simpler contracts.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[3] C. Cachin, "Architecture of the hyperledger blockchain fabric," in *Workshop on Consensus Ledgers*, vol. 310, 2016.

[4] V. Buterin, "Chain interoperability," *R3 Research Paper*, 2016.

[5] G. Mainland, G. Morrisett, and M. Welsh, "Flask: Staged functional programming for sensor networks," in *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '08. New York, NY, USA: ACM, 2008, pp. 335–346.

[6] A. Mizzi, J. Ellul, and G. Pace, "D'artagnan: An embedded dsl framework for distributed embedded systems," in *Proceedings of the Real World Domain Specific Languages Workshop 2018*. ACM, 2018, p. 2.

[7] R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming wireless sensor networks using kairos," in *International Conference on Distributed Computing in Sensor Systems*. Springer, 2005.

[8] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," in *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, ser. IPSN '07, 2007.

[9] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan, "Reliable and efficient programming abstractions for wireless sensor networks," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. ACM, 2007.

[10] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, 2005.

[11] S. Peyton Jones, J.-M. Eber, and J. Seward, "Composing contracts: an adventure in financial engineering (functional pearl)," in *ACM SIGPLAN Notices*, vol. 35, no. 9. ACM, 2000, pp. 280–292.

[12] P. L. Seijas and S. Thompson, "Marlowe: Financial contracts on blockchain," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2018, pp. 356–375.

[13] D. Harz and W. Knottenbelt, "Towards safer smart contracts: A survey of languages and verification methods," *preprint arXiv:1809.09805*, 2018.

[14] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 254–269.

[15] I. Sergey, A. Kumar, and A. Hobor, "Scilla: a smart contract intermediate-level language," *arXiv preprint arXiv:1801.00687*, 2018.

[16] "Bamboo," https://github.com/pirapira/bamboo.

[17] M. Coblenz, "Obsidian: a safer blockchain programming language," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 97–99.

[18] "Vyper." [Online]. Available: https://vyper.readthedocs.io/en/latest/

[19] R. O'Connor, "Simplicity: A new language for blockchains." [Online]. Available: https://blockstream.com/simplicity.pdf

[20] S. Popejoy, "The pact smart contract language (2016)." [Online]. Available: http://kadena.io/try-pact/

[21] J. Pettersson and R. Edström, "Safer smart contracts through type-driven development," Ph.D. dissertation, Chalmers University, Sweden, 2016.

[22] C. K. Frantz and M. Nowostawski, "From institutions to code: Towards automated generation of smart contracts," 2016.

[23] A. Mizzi, J. Ellul, and G. Pace, "Macroprogramming the blockchain of things," in *Proceedings of The 1st International Workshop on Blockchain for the Internet of Things*. IEEE, 2018, pp. 1673–1678.

[24] M. Dan and C. Arlyn, "The blocknet: Design specification," https://www.blocknet.co/wp-content/uploads/2018/04/whitepaper.pdf.

[25] "Btc relay," https://github.com/ethereum/btcrelay.

[26] "Cosmos," https://cosmos.network/docs/resources/whitepaper.html.

[27] "Polkadot," https://polkadot.network/PolkaDotPaper.pdf.

[28] R. K. Shidokht Hejazi-Sepehr and A. Sharif, "Transwarp-conduit: Interoperable blockchain application framework." [Online]. Available: https://aion.network/media/TWC_Paper_Final.pdf

[29] "Ethereum alarm clock," https://www.ethereum-alarm-clock.com/.

[30] "Oraclize." [Online]. Available: https://docs.oraclize.it/

[31] J. de Kruijff and H. Weigand, "Ontologies for commitment-based smart contracts," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2017, pp. 383–398.

[32] A. Mizzi, J. Ellul, and G. Pace, "Porthos: Macroprogramming blockchain systems," University of Malta, Tech. Rep., 2019.

[33] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International Conference on Principles of Security and Trust*. Springer, 2017, pp. 164–186.