Trotting Lightly: Reducing the Hoofprint of SmartPiggies Storage on Ethereum

Alexander Lee (ahl256) and Edward Forgacs (elf353)
New York University, Distributed Systems, Fall 2021

Abstract

SmartPiggies is a financial services protocol targeting deployment on the public Ethereum platform. Early versions of SmartPiggies utilized a significant amount of storage in the global state of Ethereum, which has become much costlier to use since the initial development of the protocol. This research describes and implements a technique for abstracting state storage from the SmartPiggies protocol and using a hash of the abstracted state data as an identifier for a financial derivative agreement on Ethereum.

1 Introduction

The SmartPiggies protocol was initially developed in early 2019 as a means for supporting financial derivatives on a suitable public network infrastructure. The protocol utilizes the functionality of a platform such as Ethereum for computing the settlement of derivative agreements and escrowing collateral without the need for a trusted third party. As the cost of computation and storage on such platforms has risen substantially in the intervening years, we sought ways in which to reduce the footprint of the protocol on Ethereum, the platform for which the initial version of SmartPiggies was built.

This paper describes a technique for abstracting a portion of the data representing a financial agreement from storage on Ethereum, and instead using a hash-based fingerprint of those data as an identifier for the agreement itself. We chose to target storage specifically for our optimizations, as this is one of the costliest operations to perform on Ethereum, and SmartPiggies makes heavy use of these operations for managing state related to financial agreements.

The remainder of the paper gives a brief overview of Ethereum and its virtual machine (Section 2), describes the SmartPiggies protocol in further depth (Section 3), illustrates the cost concerns over storage on Ethereum (Section 4), describes our fingerprinting technique, implementation takeaways, and cost savings results (Sections 5-7), briefly surveys related research (Section 8), and notes the extensibility of our technique to other applications targeting Ethereum or similar systems.

2 Ethereum and the EVM

Ethereum [6] is a public replicated state machine (RSM) allowing for the transmission of a native unit of value (ether) and the deployment of arbitrary computational logic in the form of programs called smart contracts. The program code of these smart contracts is executed by the Ethereum Virtual Machine (EVM), which is a 256-bit Turing-complete stack-based machine with special VM instructions facilitating handling of ether and computation of certain cryptographic primitives.[1]

The EVM is run by all replicas participating in the Ethereum RSM to compute updates to the world state. Ethereum is distributed but logically single-threaded, and proof-of-work is used to solve consensus for ordering batches of transactions ("blocks") that run on the EVM and update global state.[2] Because the system is public and anyone may deploy smart contracts to it, a mechanism exists to disincentivize spurious computation and abuse, such as forcing all replicas to execute infinite loops: each instruction in the EVM has an associated cost in a unit called gas. The amount of gas (i.e., computation) permitted to be consumed in a single block is limited by global parameters of the RSM, and all gas consumed by any given transaction must be paid for in ether, presenting an economic disincentive for abuse and an overall limitation on computation that replicas may execute per block.

[1] Computation in the Ethereum system is "Turing complete" but with limited execution extent; arbitrary logic may be deployed, but may only execute for a limited number of computational steps, meaning all smart contract execution halts with certainty.

[2] For more detail on proof-of-work based consensus, see [12].

3 SmartPiggies Protocol

The SmartPiggies protocol (described in detail in [2]) defines a model for the creation of derivative financial contracts, targeting deployment on a decentralized public RSM infrastructure such as Ethereum to facilitate a global marketplace for both sides of such arrangements.[3] The SmartPiggies design requires the capability for the targeted RSM to support non-fungible tokens (NFTs) which represent the actual derivative agreements, fungible tokens representing the collateral for the agreements, and an oracle service for fetching price data outside of the RSM in order to settle the agreements. As Ethereum provides broad support for all three of these components, it was the initial target for a SmartPiggies deployment.

A SmartPiggies NFT in the original protocol design is a parameterized object representing a collateralized financial option, implemented as a series of smart contracts. The collateral for any given SmartPiggies NFT must be in the form of fungible tokens defined on the same RSM platform (i.e., by a separate smart contract deployed to Ethereum), which the protocol can automatically lock. In this way, the protocol can offer a guarantee to counterparties to the agreement that the underlying funds are available. The oracle service (another set of smart contracts combined with functionality external to the RSM) is used to fetch settlement prices for the security on which the option is being struck. Because the parameters representing the agreement are associated with each NFT in the original design, a substantial amount of parameter data must be stored as part of the world state of the RSM.

4 The rising cost of storage on Ethereum

As described in Section 2, the EVM prices all computation in gas, which must be paid for in ether. Both the cost of gas in terms of ether and the price of ether itself in terms of external currencies such as the US dollar float on open markets. When there is greater demand for computation to be performed on Ethereum, the price of a single unit of gas in terms of ether may be bid up, and similarly, demand for ether in the financial markets can drive up its cost in dollar terms.

Since the original development of the SmartPiggies protocol in early 2019, both prices have increased dramatically, compounding to more than two orders of magnitude.[4] As noted in Section 3, SmartPiggies as originally designed makes heavy use of RSM world state storage for parameters of the NFT. The EVM opcode for such storage (SSTORE) is one of the costliest opcodes in terms of gas, as state storage must be replicated everywhere, potentially permanently.[5] In addition, the SSTORE and corresponding SLOAD opcodes were repriced in gas terms to be slightly costlier during a recent upgrade to the Ethereum system [5]. The effect of these changes on SmartPiggies has been to drastically increase the cost of using the protocol on Ethereum in dollar terms, incentivizing research into the gas cost reduction technique described in Sections 5 and 6.

5 Fingerprinting via static state abstraction

Due to the increasing cost of storage use on Ethereum described in Section 4, and the relatively lower cost of performing in-memory operations on the EVM, we sought to redesign a minimal set of the SmartPiggies protocol optimizing for reduced storage cost.

We sought to approach this problem by first identifying static portions of the state of a generic NFT, which could potentially be stored outside of Ethereum as long as their parameterized versions were verifiably associated with a specific NFT within Ethereum. Such portions of the state represent a large cost drag on the use of the protocol if they are write-once, read-only during the lifecycle of an NFT. When those parameters are needed for performing computation as part of a function call on the smart contract, they can instead be passed into the function and manipulated in the EVM memory rather than being read from EVM storage. We identified nine such stored immutable fields associated

[3] The word "contract" is overloaded in both the world of Ethereum and the financial world; we will try to keep the usage clear in context.

[4] According to data from Ycharts [7], [8], average gas price in units of gwei was between 10 and 20 in early 2019; in late 2021, the gas price has risen an order of magnitude to between 100 and 200. One gwei is 1×10^{-9} ether. Similarly, price data for ether itself illustrates a rise from near 100 US dollars to over $4000 today.

[5] The complete formulation for SSTORE costs is described in [16], and as can be seen in [13], is substantially costlier than MSTORE operations for manipulating objects in the EVM memory only.

with every SmartPiggies NFT and used these as the basis for our cost savings mechanism.

The savings mechanism uses the Keccak-256 [3] hash of the nine fields plus a newly introduced per-user-account nonce that tracks the number of NFTs created by any given account as an identifier for a given NFT. This allows for unique identifiers with the same set of parameters for the nine static fields (because the creator's account address is itself one of the fields), enabling users to create unique NFTs with the same set of parameters if they wish.[6]

5.1 Self-validating NFT identifiers using fingerprints

Importantly, the hash of all 10 fields (including the nonce) becomes a self-validating identifier when calculated on a smart contract deployed to the EVM (in our implementation's case, the same smart contract defining SmartPiggies itself). The owner of a particular NFT can be mapped from the identifying hash, guaranteed by the nonce to be unique per token. The original implementation identified NFTs via a globally unique 256-bit integer field stored directly on the smart contract; this is no longer strictly required, as the identifiers map uniquely into a 256-bit space by virtue of the hash algorithm.

Any function needing to index and validate the owner of a SmartPiggies NFT can accept the fingerprinting parameters (along with any others required by the function), validate them on the smart contract using cryptographic functions implemented directly in the EVM, and then perform logic based on those parameters or any other dynamic parameters which still need to be stored in the world state of the RSM.

5.2 Storage vs. computation tradeoffs

The fundamental tradeoff being made by our alternative implementation is SSTORE and SLOAD operations against any other type of operation in the EVM. As noted in Section 4, these storage manipulation operations are among the most expensive in the EVM. While hashing is generally a relatively computationally expensive operation, the EVM provides a virtual machine instruction (SHA3) for obtaining the Keccak-256 hash of arbitrary byte strings, with a gas cost that is multiple orders of magnitude lower than storing a word in a previously empty storage location. Thus, our use of a hash-based identifier can take advantage of optimized VM operations to validate fingerprints within Ethereum, rather than needing to develop custom smart contract code for these cryptographic operations.

While the need to pass in multiple parameters for fingerprint validation to any function call requiring the identifier of an NFT introduces memory manipulation costs (via MSTORE and MLOAD), these are substantially cheaper than the corresponding storage operations. In many cases, the fingerprinting parameters may be a superset of (or intersection with) the parameters that a function would require in the original implementation, or with parameters that would need to be loaded from storage (via a costly SLOAD) during invocation of that function, helping to offset the new memory manipulation costs.

5.3 User / developer experience tradeoffs

Another set of tradeoffs with our storage-optimized implementation relates to the experience of using and developing against the revised protocol. One obvious tradeoff of abstracting the static state per NFT away from the smart contract is that this data must now be stored somewhere else, as it is no longer part of Ethereum's global state. If the data identifying a particular NFT are lost, the fingerprint will no longer be able to be reproduced, and any functionality requiring validation of the fingerprint data will be rendered unusable. Fortunately, these data can be considered non-sensitive (in the original implementation, they are stored publicly on Ethereum with no privacy whatsoever), and thus they may be stored in any (potentially replicated) external system(s). Any party who obtains the fingerprint data will not be able to make use of them in our implementation if they cannot also sign a transaction using those data with the private key of the owner associated with the fingerprint itself. Thus, the security of the original ownership model is preserved, and the abstracted state data may be stored publicly if desired.

While the identifying state data may be stored publicly without fear of theft, the user is still faced with

[6] This additional state data to track the per-account nonce for the number of NFTs created does in fact introduce additional SSTORE costs, but this is more than offset by the savings of the mechanism as shown in Section 6.

a potential application usability hurdle if they must include these data with an Ethereum transaction manipulating one of their NFTs. However, specifically because the data are non-sensitive, any application offering services on the SmartPiggies protocol may itself store a copy of these identifying data, transparently passing them through to Ethereum when requested. This of course implies somewhat greater difficulty for developers of such applications, though nothing insurmountable.

One other difficulty for developers of applications built on top of the SmartPiggies protocol is that if they wish to aggregate metadata about NFTs (to create a marketplace application, for example), they must store and read the parameters of those NFTs externally in some fashion, as they will not be available to read directly in the global state of our alternative implementation. In practice, however, this approach is taken by many application developers already; reading from Ethereum directly is much slower than reading a cached copy of data stored in some other form of database. In practice, while this would add some friction for application developers, we again do not consider it to be an unmanageable hurdle.

6 Implementation and evaluation

The implementation approach for the storage-optimized functionality involved an initial analysis of the original SmartPiggies implementation code targeting Ethereum to determine the subset of (static) storage fields associated with NFTs to use for the hash-based fingerprint. A subset of the original functionality to represent common lifecycle activity of a SmartPiggies NFT was then ported to a new baseline smart contract in a development framework (Hardhat) supporting more robust gas usage data collection, and an alternative JavaScript framework for interacting with deployed smart contracts (ethers.js, where the original code was developed using web3.js).

Following the baseline port, a challenger smart contract was implemented in Solidity using the hash-based fingerprinting described in Section 5, and tests using the Hardhat framework were written to collect gas costs of core functionality for both the baseline and challenger smart contracts. Functional tests were also implemented for core functions of the challenger code to validate that the rewritten logic still satisfied protocol needs.

6.1 Testing results and evaluation

Initial test results were collected for the five core lifecycle functions on both the baseline and challenger smart contracts. Due to the complexity of the original implementation (involving interaction with external fungible token and oracle smart contracts, as described in Section 3), certain functionality was mocked or elided where interactions had been tested in the existing SmartPiggies codebase. In applicable cases, gas cost tests were performed on an independent simple fungible token smart contract, and gas costs for certain portions of elided code in the challenger smart contract were inferred from the baseline contract to provide a more direct comparison.[7] The gas costs before and after these corrections are shown in Figure 1.

	Baseline	Corrected Baseline	Challenger	Corrected Challenger
createPiggy	324,950	384,191	74,814	134,055
transferFrom	86,041	86,041	29,909	29,909
reclaimAndBurn	103,756	103,756	52,915	96,319
settlePiggy	129,826	129,826	27,708	115,962
claimPayout	33,111	84,592	33,111	84,592

Figure 1: Gas costs of core lifecycle functions in baseline and challenger smart contracts, before and after adjustments for mocked code.

Our results demonstrate significant cost savings for certain core lifecycle functionality, particularly the creation of an NFT and its transfer, which both accrued a savings of approximately 65% after correcting for cost measurements in the challenger smart contract. Other common lifecycle functions still obtain more modest cost savings, except for the functionality to claim any payout owed by the smart contract, as this was already optimized in the original protocol design via a set of internal records to net out potential transfers until requested by users.

[7] Specifically, for any calls to functions where the challenger smart contract had elided code due to dependencies on the nature of the NFT identifier (a bytes32 object in the challenger implementation vs. a uint256 in the baseline implementation), the baseline code was run with and without the corresponding lines commented out, gas measurements were taken in both cases, and the difference was applied back into the challenger smart contract results.

We also look at cost savings for two common lifecycle patterns, outlined in Figure 2 below. In Scenario 1, a new NFT is created, sold one or more times (incurring a call to transferFrom), and ultimately settled as a financial option, with at least one counterparty claiming a payout. In Scenario 2, an NFT is created but not sold (due to lack of demand until market conditions become unfavorable to the seller, perhaps, or an error in the initial parameterization of the NFT), and then is burned so that the creator may reuse the collateral.

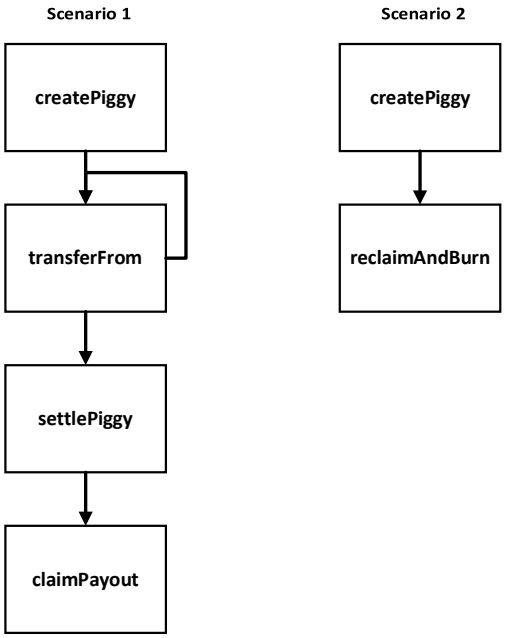

Figure 2: Common usage scenarios and function calls (as implemented on both baseline and challenger smart contracts).

As SmartPiggies NFTs are by design bespoke financial contracts, it is reasonable to assume the cost of a single transferFrom call in Scenario 1, though of course re-sale of the instrument is possible. If we assume a single transfer (reflecting the sale of the created NFT) and a split of the collateral with both counterparties claiming their respective share after the settlement of the option, we obtain the gas cost comparisons shown in Figure 3.

	Baseline	Challenger	Challenger % savings
Scenario 1	769,242	449,110	42%
Scenario 2	487,947	230,374	53%

Figure 3: Gas cost savings for common lifecycle functionality as implemented in baseline and challenger smart contracts (using corrected costs from Figure 1, for scenarios depicted in Figure 2).

Given approximate gas prices (in ether) and average ether prices in dollars as of this writing, the dollar savings for the challenger functionality are approximately $142 for Scenario 1 and $114 for Scenario 2. If additional smart contract functionality beyond the five core functions can be optimized using similar patterns, savings may potentially be greater.

7 Open implementation issues

The primary open issues related to the implementation of the storage-optimized functionality are the portions of the smart contract code which were mocked to facilitate testing of the developed prototype, and the portions of the original implementation which were not tested. As described in Section 6, the challenger smart contract functionality was itself tested for core functions that differed from the baseline implementation (claimPayout is effectively the same across both implementations due to earlier optimizations) and gas cost corrections for any elided functionality were made to attempt an “apples-to-apples” cost comparison as described in Section 6.1.

For functionality not tested in our implementation, certain features appear more amenable to the hash-based fingerprinting model (such as the functionality to “split” an NFT into two new NFTs, with part of the collateral associated with each), and others (primarily auction-related functionality) appear more challenging. The challenge arises particularly from the stack-based nature of the EVM, as there is a limit to the stack depth and thus functions that use many internal variables may ultimately run up against the limitations of the stack. When the stack depth is exceeded, the VM will revert the function call and no update will be made to the global state of the RSM, though gas used for the (speculative-but-failed) execution will still be consumed. There are a variety of potential approaches to this issue, such as an internal virtual function table on the smart

contract for dispatching a validated fingerprint to another function (in certain circumstances), or functionality that may only be called by the owner of an NFT (which would avoid the need for passing through the hash preimage variables in order to validate the data, as the hash itself is associated directly with the owner on the challenger smart contract, and the caller of any function can be verified by their signature on the Ethereum transaction itself). The question of what specific approach might enable functionality like the original protocol's auctioning mechanism to be used alongside the storage optimizations described in this paper is an area left to future research.

8 Related research

Several approaches for reducing gas consumption on Ethereum (a "Layer 1" RSM) have been described and developed to varying degrees. Some are simple design patterns, as outlined in [11], which advocate for minimization of on-chain state in general but do not necessarily prescribe any specific approaches, as these may be application-specific. Other general approaches may include compiler tools for precise gas cost estimation and optimization, as in [1].

Another broad class of approaches is that of "Layer 2" solutions, which are a class of secondary systems that can interface with Ethereum (or another Layer 1 RSM) to provide greater application scalability for applications seeking to leverage the security and functionality of Layer 1. A prevalent form of Layer 2 systems for Ethereum are transaction rollups, which may be zero-knowledge rollups or optimistic rollups (described in more depth in [9]). Newer hybrid systems such as Validiums and Volitions allow for storage of data outside of Layer 1, and in the latter case allow for per-transaction specification of where associated data should be stored, rather than using the batched approach of rollups that store data on Layer 1. These approaches using storage outside of Layer 1 (e.g. [4], [10]), in which users may potentially take custody of a copy of their own transaction data, resemble our approach though they do not necessarily use Ethereum, and do rely on Layer 2 prover systems, which we do not.

Another approach to minimizing the usage of Layer 1 Ethereum is the concept of state channels, which are similar to payment channels in Bitcoin [14] but allow generalized computation in the ideal case. Interestingly, and unbeknownst to the authors at the time this research was conducted, the State Channels open-source project which conducts research in this area related to Ethereum used an approach nearly identical to our own for Layer 1 data reduction (storing hashed data only on Layer 1, and "rehydrating" it by accepting the hash preimage as input; see [15] for details). Where our approach appears to differ, and offers what we believe is a novel contribution for Layer 1 application development, is in the use of the hashed data as a self-validating identifier for the object parameterized by the hash preimage.

9 Extension to other applications on Ethereum

While the development of the storage optimization technique described in this paper was driven primarily by the involvement of one of the authors in the design and development of the SmartPiggies protocol, the technique itself should be more broadly applicable to smart contracts deployed on Ethereum (or any other Layer 1 public RSM that similarly charges high fees for persistent state storage). The technique of using a hash on Layer 1 as an identifier for the object parameterized by its preimage may be most appropriate for NFT patterns (where the typical ownership data structure is a mapping from unique NFT identifiers to owners' addresses), but any other identifiable object used in a smart contract implementation could be a good candidate as well.

Cost savings realized by adopting this technique will be dependent to an extent on the application in question (and particularly the number of SSTORE / SLOAD instructions executed by its core functionality), and the stack limitation issues identified for the SmartPiggies implementation in Section 7 may apply to other Ethereum-based applications (as may other issues not uncovered in our specific research on the SmartPiggies protocol).

Importantly, this technique may be considered orthogonal to Layer 2 optimizations as outlined in Section 8, potentially facilitating compound cost savings for applications targeting Layer 2 on top of Ethereum.

10 Conclusion

This paper has presented a technique for reducing the storage footprint of the SmartPiggies protocol on Ethereum by abstracting portions of the state of a

SmartPiggies NFT and using the hash of those data as a self-validating identifier. In our tests, these optimizations result in gas cost savings of approximately 65% for two core functions of the protocol (including the costliest, to create an NFT), and over common lifecycle scenarios can result in aggregate cost savings of approximately 40-50%. Porting additional functionality of the existing smart contracts to this new storage-optimized model may result in further savings for common use cases. Furthermore, these techniques may be considered orthogonal to other forms of cost optimization, such as using Layer 2 solutions for portions of the application deployment. The techniques described in this paper, while illustrated in the context of the SmartPiggies protocol, should also be generally applicable to other forms of stateful data abstractions on Ethereum and similar platforms.

References

[1] E. Albert, J. Correas, G. Román Díez, P. Gordillo and A. Rubio, "GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts," TACAS Artifact Evaluation, 2020.

[2] M. Arief, T. J. Algya and A. Lee, "SmartPiggies: An open-source standard for a free peer-to-peer global derivatives market," 7 March 2019. [Online]. Available: https://bit.ly/3dxwHAW.

[3] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche and R. Van Keer, "Keccak Implementation Overview," 29 May 2012. [Online]. Available: https://bit.ly/3DGXAx4.

[4] T. Brand, A. Levy and U. Kolodny, "Volition and the Emerging Data Availability spectrum," 14 June 2020. [Online]. Available: https://bit.ly/307yg5E.

[5] V. Buterin and M. Swende, "EIP-2929: Gas cost increases for state access opcodes," 1 September 2020. [Online]. Available: https://bit.ly/31ypi2h.

[6] V. Buterin, "Ethereum Whitepaper: A Next-Generation Smart Contract and Decentralized Application Platform," 2013. [Online]. Available: https://bit.ly/3DHuVYB.

[7] "Ethereum Price," December 2021. [Online]. Available: https://bit.ly/33eDTAg.

[8] "Ethereum Average Gas Price," December 2021. [Online]. Available: https://bit.ly/3EDlvyF.

[9] "Layer 2 Rollups," [Online]. Available: https://bit.ly/3IC9goK.

[10] A. Levy, "Adamantium - Power Users," 24 May 2021. [Online]. Available: https://bit.ly/3DCiIVh.

[11] L. Marchesi, M. Marchesi, G. Destefanis and G. Barabino, "Design Patterns for Gas Optimization in Ethereum," Conference: 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), 2020.

[12] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2021. [Online]. Available: https://bit.ly/31K1RT0.

[13] "Opcodes for the EVM," [Online]. Available: https://bit.ly/3oCPoK1.

[14] J. Poon and T. Dryja, "The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments," 2016.

[15] "State Channels Gas Optimization on Ethereum (revisited)," 15 March 2021. [Online]. Available: https://blog.statechannels.org/gas-optimizations/gas-optimizations/.

[16] Wolflo, "Appendix - Dynamic Gas Costs," November 2021. [Online]. Available: https://github.com/wolflo/evm-opcodes/blob/main/gas.md#a7-sstore.