The *doAnything* Smart Contract.

Timon Riechsteiner, Supervisor: Arthur Gervais

ETH Zürich, Switzerland

Abstract. This paper introduces the *doAnything* contract, a versatile framework for efficiently interacting with DeFi smart contracts in the Ethereum network. The *doAnything* contract serves as a unified interface for executing diverse DeFi operations, eliminating the need for multiple contract deployments and significantly reducing gas costs in many scenarios. This paper demonstrates its effectiveness using several manually implemented comparisons, equally divided between actual arbitrage trading strategies, previously observed exploits, real-world MEV extraction transactions and commonly used DeFi smart contract functions. By leveraging the Huff compiler, implementing contract-specific optimization strategies, and using a custom encoding scheme for passing data to our smart contract, the *doAnything* contract achieves substantial gas savings and minimal overhead across the board. With these optimizations the contract can outperform dedicated Solidity contracts. The performance incraease is both in terms of deployment costs, where the *doAnything* contract only uses 29.9 % of the gas that the Solidity implementation uses, and in terms of operation cost, with only requiring an average of 82 % gas used for each smart contract call, compared to the Solidity contract. The *doAnything* contract performs equally efficient as two smart contracts used by actual MEV extraction bots, and is able to execute many exploits that were observed on-chain, either beating the equivalent Solidity implementations, or only incurring minimal overhead. Further, the *doAnything* contract allows for layering arbitrarily many contract functions with control leaks together, and it supports returning data to static calls.

Keywords: Cryptocurrency · Ethereum · EVM · Smart Contracts.

1 Introduction

Smart contracts are programs that run in decentralized blockchain networks like Ethereum. Therefore they are crucial to provide functionality to the blockchains users. They enable decentralized exchanges [29], decentralized autonomous organisations [18], lending protocols [2] and many other financial- and non financial functionalities that benefit from the advantages that decentralized networks provide.

As the resources in such a network are not infinite, the networks implement mechanisms to allocate resources and incentivize participation. In the Ethereum network, transaction fees, also known as *gas* fees [11], are one of the main mechanisms used. The amount of fees heavily depends of the efficiency of the underlying smart contracts that are used.

As of today[1], an average of 107'000 million gas is used every day over the past six months [15]. With the current gas price [14], this is approximately half a million US-dollars spent solely on fees, every day. Therefore, it is crucial for smart contract developers and users alike, to use and develop smart contracts that are as efficient as possible, as only minor improvements can potentially save thousands of dollars.

In this paper a new smart contract is proposed, the *doAnything* contract, that functions as a framework for gas efficient interactions with other smart contracts. We outline its implementation, and provide various benchmarks to showcase its efficiency, using detailed, real-world scenarios.

It minimises gas usage in two dimensions. The first dimension is the number of smart contracts that need to be deployed. When a developer, user or advanced trader wants to perform a new set of interactions with smart contracts, a new contract usually needs to be deployed. Using our contract, many such deployments of new contracts become redundant. For example, we are able to show that the *doAnything* contract only incurs about one third of the deployment cots compared to specific Solidity implementations of concrete trading strategies.

Secondly, smart contracts that heavily depend on interactions with other smart contracts can benefit by a reduced gas cost on a per-interaction basis when using the *doAnything* contract. Our findings show that the contract is able to outperform dedicated Solidity smart-contracts in terms of gas usage by a reduction that varies between 10% and 25% per call, depending on the scenario.

2 Background

2.1 Smart Contracts and Ethereum Virtual Machine

Smart contracts are self-executing programs that run on blockchain platforms, with Ethereum being one of the most prominent [31]. These contracts are executed in a specialized runtime environment called the Ethereum Virtual Machine (EVM). The EVM is a stack-based, big-endian virtual machine designed to execute smart contract bytecode [19].

These smart contracts can be used to provide various functionality. Common examples include decentralized exchanges like Uniswap to exchange tokens [29], non-custodial liquidity protocols like Aave [2] that enables users to supply tokens, borrow tokens, and liquidate unhealthy positions, decentralized autonomous organisations (DAOs) [18] that enable users to make and participate in democratized and transparent organisations, among many others.

2.2 Gas and Gas Price

The EVM is running on many different decentralized nodes and provides computing power to its users [12]. As there is a physical limit to this shared computing

[1] 14. September 2024

power, dictated by the amount of nodes and their own computing power, there needs to be some method to distribute the networks resources. Further, running a node needs to be incentivized, as it incurs cost for example for electricity and hardware [3].

Gas is a concept fundamental to the EVM used for incentives and resource allocation. All instruction given to the EVM, for example multiplying two numbers, or loading or storing data, have some predefined constant amount of *gas* they use. Certain instructions, like invoking other smart contracts, introduce additional, dynamic gas cost, which depend on the concrete execution [11].

The amount of gas each instruction uses was defined in the Ethereum yellow paper, and does not depend directly on the actual underlying implementation. However, the values were chosen such that they do correlate with how expensive the instructions are percieved by the network [31]. For example, storing data permanently is heavily disincetvized by incurring a high fee. This is because the blockchain nodes will need to have the data available indefinitely, which, if a lot of data is stored, massively increases the physical storage required to run a node.

Interacting with the blockchain, a user needs to pay for all the gas that accrues during execution. The total fee is the gas used, multiplied by a price paid per unit of gas [11]. This price can be specified by the user sending the transaction. The price of gas is then paid in the token native to the Ethereum network, commonly known as Ether. If many users send transactions at the same time, the has to choose a subset of them to include in the current block, as the size per block is limited [9]. It can be observed, that nodes commonly include transactions that pay the highest gas fees to maximize profit. Thus, the problem of allocating the networks resources is solved by essentially auctioning the resources off to the highest bidders. At the same time, the nodes executing the transactions are able to receive part of the gas fees as payment for their participation.

2.3 Smart Contract Languages

The choice of programming language for smart contract development significantly impacts contract gas efficiency, readability, and security. Several languages have emerged to cater to different developer preferences and use cases. In this section, four key languages are outlined: Solidity, Yul, Assembly and Huff. Another prominent language is Vyper. We did not include Vyper in this paper, as due to its design principles, it does not provide low-level access to the EVM [30], which is needed to develop the *doAnything* contract.

Solidity First introduced 2014 and sponsored by the Ethereum Foundation [26], Solidity is the most widely used language for Ethereum smart contract development [7]. It is a statically-typed, contract-oriented language, which was inspired by C++, Python and JavaScript [13]. Solidity offers high-level abstractions and extensive features, making it accessible to developers familiar with object-oriented programming. The abstractions improve readability, which is crucial to write clear and bug-free code. However, high-level abstractions can sometimes lead to inefficiencies in gas usage and make it challenging for the compiler to optimise. The Solidity compiler turns the code either directly into EVM bytecode, or it can first translated it into Yul, which is then assembled into bytecode [25]. Yul, described in more detail below, can also be used as inline assembly in Solidity [24], giving the developer a way to interact more directly with the EVM.

Yul Yul is an intermediate language designed for the Ethereum ecosystem [10]. It can be compiled to EVM bytecode and is also used as an intermediate language in the Solidity compiler. Yul provides some level of abstraction, while still offering fine-grained control over EVM operations. This makes it useful for implementing complex algorithms and optimisations that are difficult to express in high-level languages like Solidity or Vyper. Further, it allows the developer to directly control the EVM, making it suitable for low-level manipulations.

Assembly EVM Assembly language provides direct access to EVM opcodes, offering the highest level of control over contract execution [4]. It allows for extreme gas optimisation and implementation of complex EVM interactions. However, Assembly code is challenging to write, read, and maintain, increasing the risk of errors and vulnerabilities. It is typically used for specific optimisations within contracts primarily written in higher-level languages.

Huff Huff is a low-level programming language designed specifically for writing highly optimised EVM contracts [21]. It provides a balance between the low-level control of Assembly and the convenience of some more high-level functionality. Huff allows for macros and other abstractions [20], enabling developers to write reusable and gas-efficient code. However, like Assembly, Huff requires a deep understanding of EVM internals and careful attention to detail to avoid introducing vulnerabilities.

3 Design and Implementation

This section provides a comprehensive overview of the *doAnything* contract's foundational concepts, design principles, and implementation details, illuminating how its flexibility and gas efficiency is achieved.

3.1 Foundational Concepts

Inter-Contract Interactions The core functionality allowing smart contracts inside the EVM to interact with one another is provided by function calls and returns. A function call takes an address of the contract that deployed the function, together with data that will be supplied to the function [23]. This data usually starts with four bytes of the function name hashed together with the argument types, which is called the signature [5]. This de facto standard allows for the smart contract that is called to determine which logic the caller wants to invoke. If the function takes arguments, those are appended to the signature. A function call starts a new call-context. After calling, the called contract executes its logic. The end of such a call-context is marked by a return, which exits the context. The return instruction can also be used to pass data back to the caller [23]. Using only these two basic operations of the EVM, smart contracts can already make complex interactions in the Ethereum smart-contract space. For example, the first call exchanges token A to token B, while the second call exchanges token B to token C, and the third call deposits the tokens C as liquidity in another exchange.

Operation Slots The main task of the *doAnything* contract is to execute any sequence of call and return operations. To achieve this, the operations are first encoded off-chain, using a custom operation encoding. The choice of operations can be done either using a fuzzer, manually selecting operations, or using other software to generate the desired operation sequence. An encoded call or return operation is what we call a *slot*. Each slot will contain meta-data about the operation, together with the data required by the specific operation. A call slot for example encodes the address that will receive the function call, the signature used to identify the function, and the arguments supplied to the function.

The chosen operations are then passed to the encoder, which turns them into a sequence of bytes. Those bytes can be decoded back into operations by the *doAnything* contract, which then executes them. Further, the last slot contains data that is used to finalise the execution. The *doAnything* contract can check if the operation sequence generated the desired amount of native tokens. Additionally, it can send a bribe to the coinbase, the address of the current block builder. This is done to further incentivize the inclusion of a transaction.

3.2 Slot Encoding

Header At the beginning of each slot (except the last slot, s_n) is a header. A header has two components. A flag field and a length field. The flag field is one byte long, and contains metadata about the encoded operation. For example the first flag bit is set to one if the slot is a `CALL` slot, otherwise it is set zero. See Table 1 for a complete list of flags used. This allows the contract to efficiently process each slot. The length field contains the the number of bytes of data that are contained in the current slot, encoded as an unsigned seven byte number. The length is used to both determine the beginning of the next slot, and to infer the size of the call- or return-data used for the operation.

CALL-Slot A `CALL`-slot is used for a call instruction[23]. The primary components (cf. Figure 2) of the `CALL`-slot encoding are the target address of the smart-contract that receives the call (*target*), the amount of native token sent to the target address (*value*), and the data passed to the called contract (*calldata*). There are two optimisations applied. First, if the value is zero, it is wasteful to include 32 bytes of zero to represent that. Thus, the value chunk is only stored

Name	Set to 1 if
`IS_CALL`	1 for `CALL` slots.
`HAS_CALLBACK`	1 if additional logic is executed inside a callback.
`REQUIRES_STORE`	1 if the slot needs to be stored in transient storage.
`HAS_DATA`	1 if the slot has any call or return data.
`HAS_VALUE`	1 if the `CALL` sends Ether
`IS_STATIC`	1 if the `RETURN` slot is used inside a static call.

Table 1: Flags that are stored in the header.

if non-zero. Otherwise, the corresponding flag indicates that there is no value chunk. The second optimisation is based on the fact that calldata usually contains $4 + n \cdot 32$ bytes, where n approximately correlates with the number of arguments to the function. This is because the first four bytes contain the function signature, and, since the EVM uses 32 byte words, the arguments are encoded in 32 byte packets (chunks). Therefore, the header size was chosen such that the signature can be stored in the first chunk, and the remaining $n \cdot 32$ bytes fit neatly in the remaining chunks of a slot.

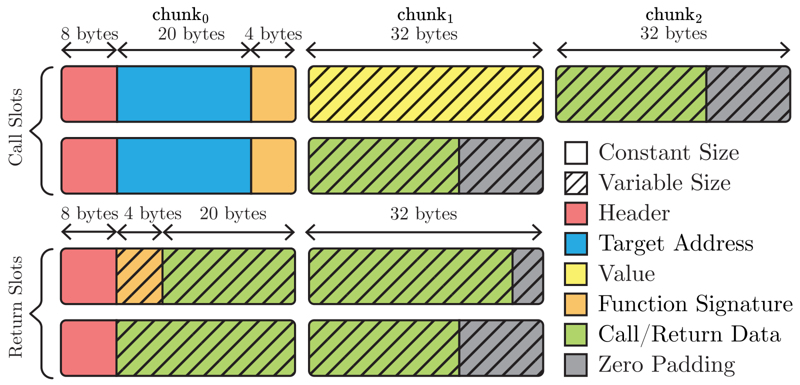

Fig. 1: The encoding of `CALL` and `RETURN` slots. The two topmost rows both depict `CALL` slots, where the upper row includes a value chunk and the lower row does not. The two bottom rows depict `RETURN` slots, where the upper row includes the signature used for a static return, and the lower one does not.

RETURN-Slot A `RETURN`-slot (cf. Figure 1) encodes a return instruction [23]. It can be used to return data to the caller, but it is not a must. The slots header encodes that it is a `RETURN`-slot and if it contains any data. If it contains data, it is appended after the header.

In the edge case where data is returned to a static-call [23] instead of a regular call, the `RETURN`-slot additionally contains a function signature, which is four bytes. The signature is put between the header and the returndata. For a more comprehensive explanation of the static-return mechanism refer to Subsection 3.3.

FINALISE-Slot The `FINALISE`-slot represents the most straightforward slot type. It stores the value `MIN_TRANSFER`, the minimum amount the operations are expected to generate, and `BRIBE`, the amount of tokens sent to the coinbase. Both `MIN_TRANSFER` and `BRIBE` represent quantities of native tokens. In the cases examined, these values did not exceed $2^{128} - 1$. This lead to the decision to encode `MIN_TRANSFER` and `BRIBE` into a single slot, where `MIN_TRANSFER` occupies the upper 128 bits of the chunk and `BRIBE` the lower 128 bits.

3.3 Execution Contexts and Return to Static Call

There are two types of execution contexts, external and internal. Which context the *doAnything* contract is running in depends on the chosen operations.

Internal Execution Context The *doAnything* contract, lets call it A in this section, always starts in an internal execution context when it is first invoked. It then proceeds to execute the encoded operations in this context. Let us consider the case where the contract calls another smart contract, which we will call B. This smart contract B can itself call an arbitrary contract C when called. If the contract C is not our *doAnything* contract, i.e. $C \neq A$, we remain in the internal execution context.

External Execution Context If the smart contract B calls our contract A again instead, it enters an external execution context. Any time the *doAnything* contract is called by another contract, instead of the initial call by the user, it will be in the external context.

For illustration purposes, let A_i be the i-th execution context, and assume that the sequence of invoked contracts is $A_1 \rightarrow B \rightarrow A_2$, where A_1 is internal and A_2 external. The differentiation between internal and external is crucial to the execution of the *doAnything* contract, because data that is specific to the first, internal context A_1, like calldata and memory, is not accessible in the second invocation A_2 [22]. Thus we need to have additional logic to ensure that data supplied to A_1 can be accessed from the external execution context A_2. This logic will copy the data from the calldata to transient storage, where A_2 can access it. After a external execution context has read all slots from transient storage and executed them, it is terminated by a return slot. In our example A_2 returns to B, and after the contract B returns back to A_1, we are in the top-level, internal execution context again.

Further, our contract allows for arbitrary layering of external execution contexts, i.e. $A_1 \rightarrow B \rightarrow A_2 \rightarrow C \rightarrow A_3 \; \ldots Z \; \rightarrow A_k$ would be a possible combination of contract executions. Each A_i for $i > 1$ is an external execution context. See Figure 2 for a graphical representation of a internal execution context and one layer of external execution context. In fuzzing literature, a contract function (method) that calls back to its caller is sometimes called a method with control leaks, as the other contract B "leaks" its control back to A during its execution. Thus all called methods with control leaks will lead to an external execution context.

Return to Static Call The EVM supports a special type of call instruction, a static call [23]. A static call is like a regular call, except that it does not allow the state of the EVM to be changed during the execution of said call. A static call provides the benefit that for example no tokens can be transferred. Thus they enable someone using a static call to have more security guarantees when invoking other contracts. As they cannot modify state, their only functionality is to retrieve information about the current state.

Such static calls pose a difficult problem to the *doAnything* contract. Lets consider the invocation sequence $X \rightarrow_i A_i \rightarrow B \rightarrow_{\text{static}} A_{i+1}$. As in the previous section, A is the *doAnything* contract, while X and B are not. The sequence differs to the previous one in that the contract B uses a static call instead of a "regular" call. Note that X can either be an externally owned account (EOA) or a contract.

Lets assume that our contract A_i is in state σ just before it calls B. In this state, the contract is already prepared to return to the static call. It then calls B, which in turn calls back and starts context A_{i+1}. In context A_{i+1}, we return the data, and exit the context. Then B finishes its execution, and returns to A_i. As a static call does not allow state changes, our contract will still be in state σ, like before it called B. Thus A_i is unable to tell if it has already called B or not.

There are two parts to our solution. Firstly, the `RETURN` slot used in context A_{i+1} has the `IS_STATIC` flag set. When this flag is read as true, our contract suppresses all state changes while executing the slot. The second part is that the function signature used by the call $\rightarrow_{\text{static}}$ is stored in the return slot. If our contract is in state σ and sees the signature corresponding to the call $\rightarrow_{\text{static}}$, it concludes that it is inside the static call. If our contract is in state σ, but the signature it sees corresponds to the call $\rightarrow_i$, it concludes that it has already returned data to the static call from B. In that case, the contract applies the necessary changes to the state that it was unable to do inside the static call, and continues execution as usual.

This solution requires that the call $\rightarrow_i$ uses a different signature than the call $\rightarrow_{\text{static}}$. The signature is equal only if both B and X call into the contexts A_i and A_{i+1} using the same function, i.e. both calls target a function of the same name and with the same argument types, which should rarely happen in practice.

3.4 Execution Contexts and Flow

The execution flow can be broken into four stages. See Figure 2 for a graphical representation of the execution flow.

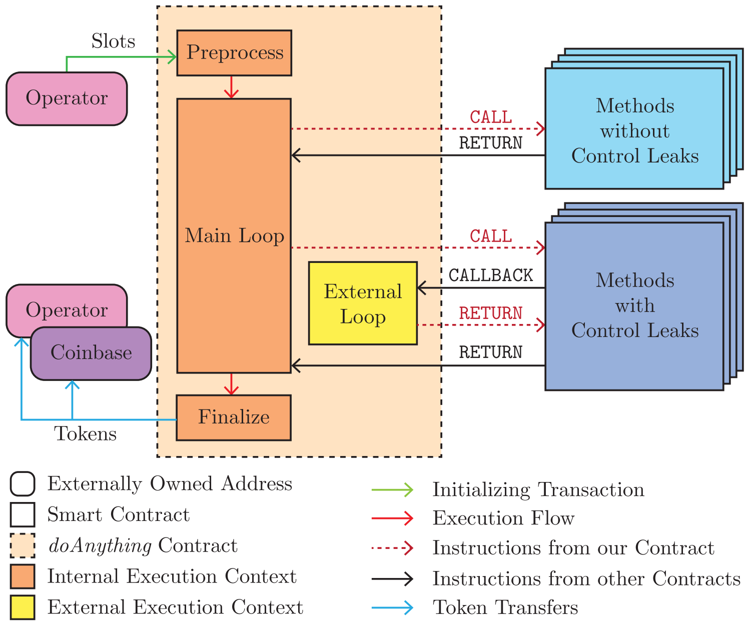

Fig. 2: The *doAnything* contract execution flow. In the first stage, the operator supplies the encoded slots to the contract. The three other stages happen inside the contract, executed in the following order: The preprocess stage, the main execution and the finalise stage.

Initiation The first stage is initiating the execution, which is done outside the contract. It comprises the operator selecting the desired operations, and encoding them into slots, using the format specified in the previous section. Our approach uses scripts written in Golang, but any software for manipulating bits can be used. The last step of the first stage is to call the contract and pass the encoded operations as an argument to the call. This contract call is no different to invoking any other smart-contract function. The operator interacting with the *doAnything* contract signs and send a transaction containing the data to the Ethereum network (or other EVM-like chains).

Preprocess The second stage is concerned with preprocessing the supplied slots. After having received the initiating transaction, the contract iterates over each provided slot. It checks if the slot is required in an external execution context, which is indicated by the `REQUIRES_STORE` flag. If this is the case, the contract stores the slot in transient storage to make it available in the external context.

Main Loop The main execution stage is where the core functionality of the contract is located. This stage operates in the two execution contexts described in Subsection 3.3.

First, the contract checks which context it needs to execute in. This check is done by looking at the sender of the contract call. If the owner called the contract, it must execute in the internal context. It can also be sure that it is executing it for the first time, as the owner can only execute one sequence of operations at a time. If the invocation originated from another address, we are inside a callback. Let us first cover the case where we are inside the call from the contract operator.

The first time the main stage is reached, it starts an execution loop that sequentially processes each slot. As returns are only needed to supply data when inside callbacks, the main-loop only issues calls to other contracts. The data needed for executing a `CALL`-slot is read from the operators supplied calldata. The contract prepares the values needed for the underlying call instruction on the stack and stores the calldata that will be passed to the called function in memory. The call instruction receives the pointer to this data together with the data length. Additionally, the contract stores a flag that is set to true if the call will require operations to be executed in an external execution context. After all flags are set and the values are stored on the stack, the call is executed.

If the called function does not call back, the main loop continues its execution with the next slot, once the previous call is completed. If the called function does call back, it ends up in our contract again. First, it reads out the flag set in storage, which informs it if it needs to execute additional operations in the external, call-back context. If this is not the case, it simply returns, and the external contract continues its execution, until it returns to the main loop.

If it does need to execute additional function calls, or return specific data, it goes into an external-execution loop. This loop is very similar to the main-execution loop. With the main difference that it reads the data needed to execute the operations from transient storage instead of calldata, as the calldata supplied to the top level context A_1 is not available in this external execution context.

As previously described, there can be an arbitrary layering of external-execution contexts. In each of these contexts the *doAnything* contract stores which slots already have been processed. This is done to ensure that all the context are synchronised, and no slot is executed twice. After having executed all the necessary calls in a external callback context, the external loop is terminated with a return. Once the contract that started the external context has finished, it returns back to either another external context inside the *doAnything* contract, or it returns to the main loop in the internal context.

Finalise When the main execution loop has executed all slots, it exits the loop and the finalisation stage starts. The finalisation stage includes three steps. First, it unwraps any remaining Wrapped Ether (WETH) that might have accrued during the execution. Second, it verifies that the final balance meets the minimum specified in the `FINALISE` slot. If specified, it also sends a bribe to the coinbase. After all these operations have finished successfully, the *doAnything* contract has finished executing the supplied operation sequence. In this case, it halts the execution, and is ready for further operation sequences.

3.5 Gas Optimisation Strategies

The *doAnything* contract implements several sophisticated strategies to minimise gas consumption, which we will detail in this section.

Efficient Operation Encoding and Flags As described in Section 3.2, the data structure is compact and only stores data essential to the call. Much information is made available by individual flag bits. This minimises the amount of data that needs to be processed resulting in reduced gas usage.

Efficient Storage Usage As describe in Section 3.3, some data needs to be stored for it to be available in the external execution context. Storing and loading data is notoriously gas intensive. Thus, using efficient operation encoding reduces the number of storage accesses for both loading and storing. Further, using transient storage [27] instead of permanent storage also drastically reduces the cost. Accessing permanent storage for the first time inside a transaction costs abound 2000 gas [31, 23] and then 200, accessing transient storage costs always 100 gas, and a calldata load only 3 gas. Our implementation uses the cheapest storage available in all scenarios.

Memory Usage Optimisation Memory utilisation in the EVM incurs gas costs similarly to storage. The gas cost for memory operations, like for permanent storage, consists of a constant and a dynamic part. The dynamic part only accrues the first time certain memory or storage is accessed, while the constant part needs to be payed on each store or load. The crucial difference is that cold storage accesses always have the same dynamic cost, independent of the storage location. In memory accesses, the cost per accessed unit of memory increases with the size of the used memory. Let a represent the highest accessed memory location as a 256-bit word pointer. The total expansion cost of memory [31] (only accrued once inside each execution context) is determined by the following formula:

$$\text{cost}(a) = \left\lfloor \frac{a^2}{512} \right\rfloor + a \cdot 3$$

Note the quadratic growth with respect to a. The *doAnything* contract only uses 512 bytes of memory, plus the additional bytes that are required to pass return or call-data. This approach minimises memory expansion costs.

4 Evaluation

To assess the effectiveness and performance of the *doAnything* contract, a comprehensive evaluation focusing on gas usage was conducted. We evaluated the contract by using it to execute different DeFi operations, and compared it with *(i)* a naive implementation in Solidity without the optimisations mentioned in the Section 3.5, *(ii)* the gas consumption of the underlying function calls, and *(iii)* dedicated implementations in Solidity. The tests can be broken down into two categories. The first category consist of performing individual function calls of commonly used DeFi functions. These benchmarks serve to analyse the performance of individual operations and see the overhead they introduce. Our second test set consists of replicating transactions that were be observed on-chain, to get a sense of the real-world performance of the *doAnything* contract.

4.1 Individual Function Calls

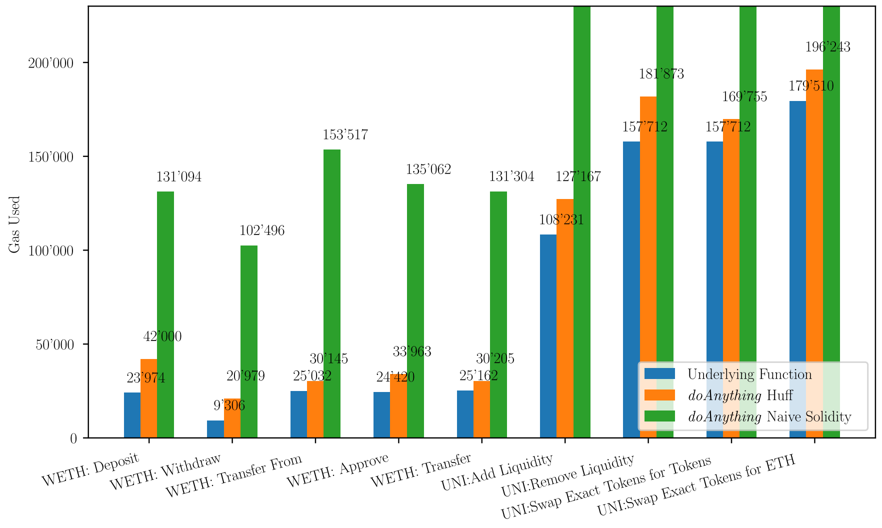

Fig. 3: Gas usage across different Uniswap V2 and WETH functions. Both `swapExactTokensForTokens` and `swapExactTokensForETH` swap over three tokens. For readability the bars were cut off at 230'000 gas.

In this section the *doAnything* contract is benchmarked using individual contract calls. The focus lies on functions from two widely-used Ethereum smart contracts: Wrapped Ether (WETH)[6], which implements the ERC20 standard [16] used for most tokens ("crypto currencies") in the Ethereum network, and Uniswap [8], which is a decentralized exchange that allows users exchange tokens implementing this standard. The gas used by the *doAnything* contract is compared to the gas used by the underlying function call and to the naive *doAnything* implementation in Solidity. The gas used in the underlying call is all the gas used by mentioned smart contracts from after they have been called, up to the point when they return. Thus this underlying gas usage is independent of the *doAnything* contract and cannot be avoided or improved on.

Our analysis reveals a small overhead compared to the underlying function call. Further, the gas usage is halved compared to the naive Solidity implementation. See (Figure 3) for a plot detailing the results. The gas savings are particularly notable for complex operations like `swapExactTokensForETH`, where *doAnything* uses only 179,510 gas compared to 436,044 gas for the naive Solidity implementation - a reduction of about 59%.

4.2 Gas Usage for Sequential Function Calls

As described earlier, gas fees can be different, for example if an address or a storage location has been used previously or not. Thus the gas fees differ for sequential calls of the same function. To get an understanding of the gas usage in sequential calls, the *doAnything* contracts performance was also tested for an increasing number of calls to Uniswaps `swapExactTokensForTokens`. The results are shown in Figure 4. In this test, the underlying cold function call requires 161'722 gas versus 54'190 gas for a warm function call. If we let n be the number of function calls, where $n \geq 0$, we can write down a function describing the underlying gas usage:

$$(n-1) \cdot 54'190 + 161722 = n \cdot 54'190 + 107'532$$

Approximating the gas used by the Huff implementation using a linear least squares fit, the gas usage is:

$$n \cdot 56'239 + 123'573$$

Subtracting the gas used by the underlying function calls from the gas used by the *doAnything* contract, we can approximate the overhead to be:

$$n \cdot 2'049 + 16'041$$

Further, the plot shows that both implementations experience linear growth in gas usage as the number of calls increases. However, the Huff version consistently remains at least one order of magnitude cheaper than the Solidity version. We also show that even for increasing number of function calls, the incurred overhead compared to just the underlying function gas usage is relatively small.

4.3 Gas Usage for DeFi Attacks

Further benchmarks of real-world scenarios were done by re-executing ten past DeFi attacks. These attacks were observed on-chain in or after January 2024. DeFi attacks are notorious for consuming more gas than average blockchain transactions due to their complex, often nested call structures. The attacks were taken from the dataset called DeFiHackLabs by SunWeb3Sec [28]. Some exploits were excluded as they did not target the Ethereum network. Further, some modification were done to the exploits to ensure that the execution was similar to what is achievable with the *doAnything* contract, and to ensure that the comparison did not unfairly favour our implementation. The modifications are as follows

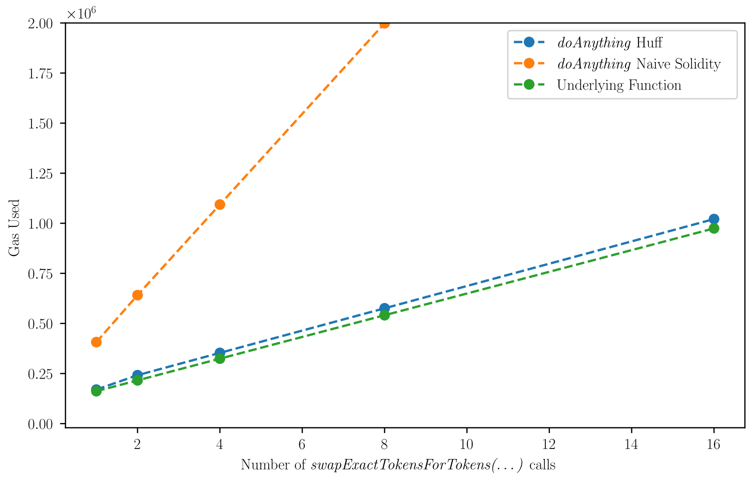

Fig. 4: A plot of number of `swapExactTokensForTokens` calls versus the gas used. All swaps involve three tokens, which remained the same for each function call. The gas used is cut off at 2'000'000 gas for readability.

1. Any logs the exploits emitted for logging values to the console were removed.
2. Setup actions, like giving the attack contract funds, were not included in the measurement.
3. Calculating a signature or other complex on-chain calculations were moved outside the exploit logic and pre-calculated, as the *doAnything* contract also requires those calculations to be done outside the transaction.
4. In general, to ensure fairness, all changes done to the exploits were only allowed to either not affect, or lower the gas used by the Solidity implementations.

The analysis of past DeFi attacks (Figure 5) shows that for all attacks the *doAnything* contract uses significantly less gas than the naive Solidity implementation. The Barley Finance attack shows the most dramatic improvement, with the *doAnything* contract using only about 23% of the gas used by the naive contract. Even the smallest difference, observed in the DAO Soul Mate attack, still shows a notable improvement of about 9% in gas usage. In part of the test cases, the hard-coded exploit in solidity used the least gas. This is to be expected, as the compiler can optimise it, and no overhead is incurred to support general functionality like the *doAnything* contract provides. Despite this, the *doAnything* contract on average beats the dedicated implementations. It uses on average 3% less gas, with a standard deviation of 9%.

These results underscore the potential of the *doAnything* implementation to reduce gas costs even for highly complex transactions like those found in DeFi attacks. The ability to beat dedicated solidity smart contracts suggests that

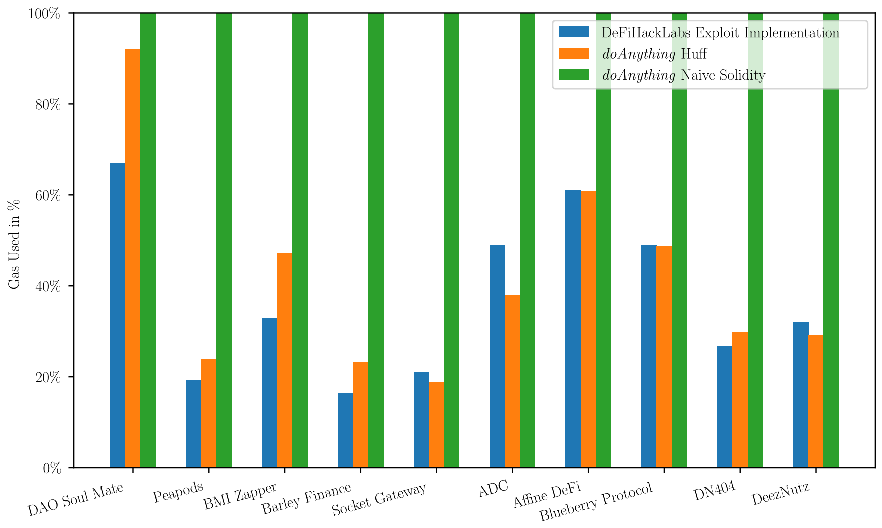

Fig. 5: Relative gas usage when comparing four DeFi exploits. For each test case, 100% is set to the implementation using the most gas, which was the *doAnything* implementation in solidity in all cases.

adopting the *doAnything* implementation can lead to cost reductions for users engaging in complex DeFi operations.

Note that these attacks were only replicated for research purposes, and that the *doAnything* contract is not intended to be used for exploits. The tests were done in test networks, using the smart contract development tool-chain Foundry [17].

4.4 doAnything vs. Dedicated Contracts

To further evaluate the efficiency of our *doAnything* contract, a comparative analysis against dedicated contracts designed for specific trading strategies was conducted. This comparison aimed to determine which approach is more gas-efficient: a single, versatile contract like the *doAnything* contract, capable of executing multiple types of trading actions. Or multiple dedicated contracts, each optimised for a specific trading strategy. For this two potential trading strategies were implemented in solidity. The first scenario is an arbitrage smart contract. It allows its user to specify a list of tokens of arbitrary length, together with a list containing the amount of token to exchange, and a list which determines which of the three predefined exchanges it should use. The contract then goes trough these lists, first approving the exchange to spend the current token, then it exchanges the token to the next token in the list. Finally, it unwraps any remaining WETH, and transfers a non-zero amount to the block builder and the message sender. The second strategy implements an arbitrage smart contract with the same functionality, with the addition that it leverages a flash loan to have more funds available for the trades [1].

Mathematical Model To formalise this comparison, the following model can be used. Let $p \in P$ be a call or return operation, and $n := |P|$ the number of operations. Further, the following notation is used

Contract	Deployment	Overhead	Underlying Cost
doAnything	d_d	$o_d(p)$	$u(p)$
Solidity	d_s	$o_s(p)$	

Table 2: The notation used for the mathematical model. Each value corresponds to the gas used by one of the two choices for a contract.

Therefore $o_d(p) + u(p)$ is the total cost incurred by executing an operation p. Note that the underlying cost $u(p)$ is independent of the implementation. Using this model the total cost of the *doAnything* contract is

$$C_d(P) = d_d + \sum_{p \in P} o_d(p) + u(p)$$

The same applies for the solidity implementation, with s instead of d as the subscript. To be able to generalise $\hat{o}_d$, $\hat{o}_s$ and $\hat{u}$ is used for the measured averages. Therefore the estimated total cost becomes

$$\widehat{C}_d(n) = d_d + n \cdot (\hat{o}_d + \hat{u})$$

When setting the inequality such that the *doAnything* contract is more gas efficient than the solidity implementation and simplifying it, we get the following.

$$\widehat{C}_d(P) < \widehat{C}_s(P) \tag{1}$$

$$d_d + n \cdot (\hat{o}_d + \hat{u}) < d_s + n \cdot (\hat{o}_s + \hat{u}) \tag{2}$$

$$d_d + n \cdot \hat{o}_d < d_s + n \cdot \hat{o}_s \tag{3}$$

Going from equation one to two follows from inserting the definition. Step two to three from the fact that the underlying cost is implementation independent.

This model allows for both average and case-by-case evaluation based on specific deployment costs, operation overheads, and expected number of executions.

If both the deployment cost of the *doAnything* contract d_d and the average operation cost $\hat{o}_d$ are less than the corresponding values of the solidity implementation, the *doAnything* contract is more efficient in any scenario, as the inequality will always hold.

If $d_d < d_s$, but $\hat{o}_d > \hat{o}_s$, i.e. the *doAnything* contract is cheaper to deploy, but incurrs a larger overhead than the dedicated Solidity contract, the *doAnything* contract is more efficient for some operations, but for large enough n, the Solidity implementation will be favourable.

If the opposite is true, i.e the the deployment cost of the *doAnything* contract is higher than the deployment cost of a Solidity contract, but the *doAnything* contract is able to execute the operations more efficient, the *doAnything* contract will be more efficient for large enough n.

The remaining case is that both the deployment and the cost per operation of the *doAnything* contract are higher. In this case, it does not make sense to use the *doAnything* contract.

Results The comparison of the *doAnything* contract versus a dedicated contract for the previously detailed strategies is detailed in Table 3. The *doAnything* contract clearly beats the Solidity implementation in the two tested scenarios. The improvements are most strongly pronounced in the size of the deployed bytecode. The size directly correlates with the deployment gas cost, as all the bytes stored on chain incur gas fees. The average operation overhead improvement is not as huge, but nevertheless notable.

The smallest improvement of 10% is seen in the flash-loan case, compared to a 25% improvement in the non-flash-loan case. The flash-loan case has a higher overhead, as the *doAnything* contract needs to make the data available inside the flash-loan callback, which requires additional storing and loading. Despite this, our contract proves more gas efficient across the board. In both scenarios, the inequality from the previous section is true for arbitrary n. In other words, no matter the total number of operations or the number of deployments, it will always be more efficient to use our *doAnything* contract than to use the dedicated Solidity implementation.

Note that this only holds for the two specific tested scenarios. If for example the strategy would use a fixed number of swaps, instead of iterating over arrays, the compiler might be able to optimise more, potentially resulting in a lower gas usage than our approach. Clearly, this needs to be determined on a strategy-to-strategy basis, which is possible using the supplied model.

4.5 doAnything vs. MEV Bots

Additional evaluation was done by comparing the *doAnything* execution to a total of ten transactions of two existing MEV bots that use simliar, "*doAynthing*"-like contracts. The MEV bots considered are *MEV Frontrunner Yoink*[2] and *c0ffeebabe*[3]. To confirm that the operations were indeed encoded in the transaction invoking the contracts, the following two steps were done. First, we determined all contract calls performed by their smart contracts to extract MEV. Second, we confirmed that the targets of the calls, the function signatures and some of the parameters passed as arguments are contained in the calldata of the MEV extracting transaction.

[2] MEV Frontrunner Yoink: 0xfde0d1...455a

[3] C0ffeebabe: 0xc0ffee...9671

Strategy	Contract	Deployed Size	d_i	$\hat{o}_i$	$\hat{u}$
non-flash-loan	*doAnything*	1462	372640	3712	35354
	Solidity	4499	1038322	4926	
	relative usage	32.4%	35.9 %	75.4%	-
flash-loan	*doAnything*	1462	372640	5526	36346
	Solidity	6627	1498852	6113	
	relative usage	22.1%	24.9 %	90.0%	-

Table 3: Bytecode size of the deployed contracts and gas usage of the different scenarios. The bytecode size is given in bytes. The relative usage is the gas used by the *doAnything* contract relative to the Solidity implementation.

After having confirmed that the calldata encoded the calls, we replicated the MEV extraction using our own contract and packing format, on the same blockchain state that the transactions were included. The MEV bot used by c0ffeebabe outperformed our method in all five transactions. The average relative performance (gas used by the *doAnything* contract divided by the gas used by the MEV bots) is 113.0% compared to c0ffeebabe, with a standard deviation of 3.0%. The bot used by Yoink only managed to outperform our method in two out of five transactions that we analysed. The average relative performance is 99.2%, with a standard deviation of 3.7%. See Figure 6 for a detailed overview of the results.

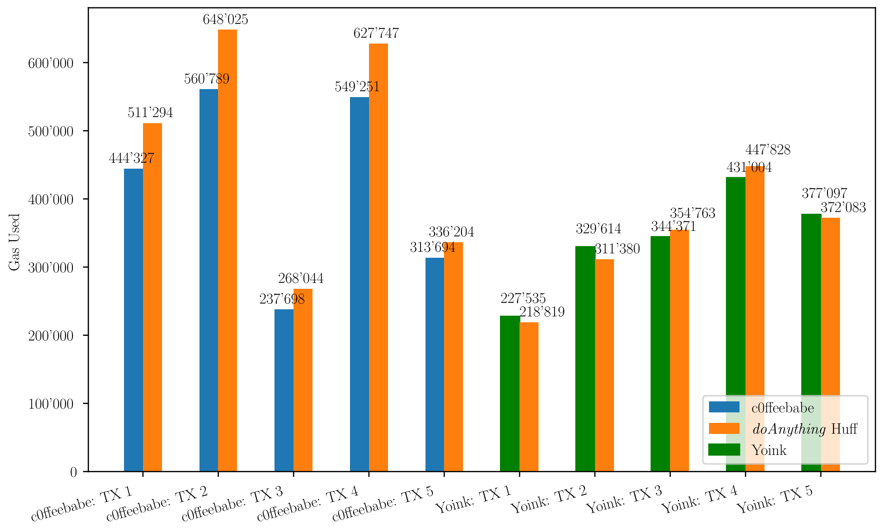

Fig. 6: The gas used by the *doAnything* contract versus MEV bots.

5 Limitation

While the *doAnything* contract is able to save gas in some scenarios, and only incurs minimal overhead in the other scenarios, there are some notable limitations to it. An individual, pure function call will always be cheaper than using the *doAnything* contract. Only in cases where a dedicated smart contract is required to be able to execute a sequence of calls (and potentially returns) the *doAnything* contract might be more gas efficient.

Further, one crucial reason that makes our *doAnything* contract gas efficient is that all call- and return-data is determined and packed off-chain. The *doAnything* contract only needs to load it in memory and can then proceed to call or return. A dedicated solidity contract, which is able to execute various calls and returns based on arguments given to it, usually needs to prepare the data with further EVM logic. For example it needs to allocate arrays, set element of the arrays, and potentially needs to copy them around in memory into the right order. Only then the dedicated contract can proceed to call the target contract. This gives dedicated implementation a clear disadvantage. However, this also points to a disadvantage of the *doAnything* contract. As the data needs to be determined off-chain, it cannot use data returned from one call as an argument to the next call, unless the data can be known beforehand.

This is especially limiting as demonstrated in the following example. Let the most recent block be n. The operator can base its actions only on the state of the block n. Assume the block builder includes the transaction invoking the *doAnything* contract as the i-th transaction in the block $n+1$, where $i > 1$. Then, transactions t_j for $j < i$ included in the same block could alter the state such that the *doAnything* contracts execution fails.

However, as our first insights into existing "*doAnything* "-like contracts used by MEV bots have shown, it would be possible to include the ability to use current, on-chain data for function calls. To which extent this is possible, and how it could affect the efficiency would be a question for further research.

Even if the limitation of using current, on-chain data for function calls is circumvented, one limitation would remain. The contract would lack the functionality to manipulate this data. For example, the current balance b of an address is queried using a function call. Assume, that the next call needs $\frac{b}{2}$ as an argument. To be able to do such calculations, it would require even more logic and functionality. To provide all this functionality, the solution could be a simple virtual machine running inside the EVM. We suspect that this would require a trade-off between functionality and gas efficiency. The feasibility and gas efficiency of such an implementation has, to our current knowledge, not been researched.

More potential gas savings could be achieved if there are different, specialised *doAnything* contracts for certain use-cases. For example, the preprocessing phase is only required because some slots need to be stored for external execution contexts. As it is known beforehand if such a external context will arise in a sequence of operations, one could use a different *doAnything* contract that does not provide this functionality, if no external context will be needed. Then one would not need to provide the `RETURN` slot functionality, and could completely remove the preprocessing calculations. This would both reduce overhead and deployment cost relative to the gas used by our *doAnything* implementation.

6 Conclusion

Despite various limitations, we were able to demonstrate savings or only minimal overhead in terms of gas usage across various, real-world scenarios. More versatile, generalised contracts, and the ability to do expensive calculations off-chain, can potentially change the approach to smart contracts for many different use-cases, for developers, regular users, more advanced traders and MEV extractors alike.

There is a lot of potential to further improve on the implementation of the *doAnything* contract. The fact that it already outperforms dedicated implementations using Solidity underlines the statement that this topic is worth further research and development. This paper can serve as a starting point for such endeavours.

Many smart contracts have existed for some time now. Revisiting their implementations, applying advanced gas saving strategies, and using low-level programming languages like Huff could improve on the user-friendliness (in terms of minimal fees) of existing, established protocols. This would have the benefit of making a protocol more accessible and enticing compared to its competitors. Especially in times of high demand for network resources, protocols that are the most gas-efficient should prove valuable to users.

Further, as we have shown, the *doAnything* contract is able to compete with contracts that MEV extractors are using today. Therefore, such an optimised and versatile smart contract is also useful to more sophisticated actors in the blockchain space. Extracting value and generating revenue requires agility and efficiency, two points that the *doAnything* contract or similar implementations certainly provide.

Bibliography

[1] Aave.com. Flash Loans. https://docs.aave.com/faq/flash-loans, 2024. Accessed: 2024-09-13.

[2] Aave.com. Introduction to Aave. https://docs.aave.com/faq, 2024. Accessed: 2024-09-13.

[3] Allie Grace Garnett. What is ETH Gas? https://www.britannica.com/money/ethereum-gas-fees-eth, 2024. Accessed: 2024-09-13.

[4] Andreas M Antonopoulos and Gavin Wood. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[5] beaconcha.in. Ethereum Signature Database. https://www.4byte.directory/, 2024. Accessed: 2024-09-13.

[6] Ethereum Community. Wrapped Ether (WETH). https://ethereum.org/en/wrapped-eth/, 2022. Accessed: 2024-09-13.

[7] Chris Dannen. *Introducing Ethereum and solidity*. Springer, 2017.

[8] DefiLlama. Uniswap Protocol. https://defillama.com/protocol/uniswap#information, 2024. Accessed: 2024-09-13.

[9] Ethereum Community. Blocks. https://ethereum.org/en/developers/docs/blocks/, 2024. Accessed: 2024-09-13.

[10] Ethereum Foundation. Yul. https://docs.soliditylang.org/en/v0.8.16/yul.html, 2022. Accessed: 2023-08-22.

[11] Ethereum Foundation. Gas and Fees. https://ethereum.org/en/developers/docs/gas/, 2024. Accessed: 2024-09-13.

[12] Ethereum Foundation. Run a node. https://ethereum.org/en/run-a-node/, 2024. Accessed: 2024-09-13.

[13] Ethereum Foundation. Run a Node. ttps://docs.soliditylang.org/en/develop/, 2024. Accessed: 2024-09-13.

[14] Etherscan. Ethereum Average Gas Price Chart. https://etherscan.io/chart/gasprice, 2024. Accessed: 2024-09-13.

[15] Etherscan. Ethereum Daily Gas Used Chart. https://etherscan.io/chart/gasused, 2024. Accessed: 2024-09-13.

[16] Vitalik Buterin Fabian Vogelsteller. ERC-20: Token Standard. https://eips.ethereum.org/EIPS/eip-20, 2019. Accessed: 2024-09-13.

[17] Foundry. Introduction to Foundry. https://book.getfoundry.sh/, 2024. Accessed: 2024-09-13.

[18] P. Hassan, S. De Filippi. Decentralized autonomous organizations. *Internet Policy Review, 10(2)*, 2021.

[19] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018.

[20] Huff. Huff by Example. https://docs.huff.sh/get-started/huff-by-example/#macros, 2024. Accessed: 2024-09-13.

[21] Huff Team. Huff Language Documentation. `https://docs.huff.sh/`, 2022. Accessed: 2024-09-13.
[22] smlXL, Inc. Ethereum Virtual Machine Opcodes Interactive Reference. `https://www.evm.codes/`, 2024. Accessed: 2024-09-13.
[23] smlXL, Inc. Evm opcodes. `hhttps://www.evm.codes/`, 2024. Accessed: 2024-09-13.
[24] Solidity Authors. Yul Documentation. `https://docs.soliditylang.org/en/latest/yul.html`, 2024. Accessed: 2024-09-13.
[25] Solidity Team. A Closer Look at Via-IR. `https://soliditylang.org/blog/2024/07/12/a-closer-look-at-via-ir/`, 2024. Accessed: 2024-09-13.
[26] Solidity Team. About Solidity Lang. `https://soliditylang.org/about/`, 2024. Accessed: 2024-09-13.
[27] Solidity Team. Transient Storage Opcodes. `https://soliditylang.org/blog/2024/01/26/transient-storage/`, 2024. Accessed: 2024-09-13.
[28] SunWeb3Sec. DefiHackLabs GitHub Repository. `https://github.com/SunWeb3Sec/DeFiHackLabs`, 2022. Accessed: 2024-09-13.
[29] Uniswap Labs. The Uniswap Protocol. `https://docs.uniswap.org/concepts/uniswap-protocol`, 2024. Accessed: 2024-09-13.
[30] Vyper Team. Vyper overview. `https://docs.vyperlang.org/en/stable/index.html`, 2024. Accessed: 2024-09-15.
[31] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151, 2014.