



Master Thesis

Gas Cost Analysis for Ethereum Smart Contracts

Author(s):

Signer, Christopher

Publication Date:

2018-11-14

Permanent Link:

<https://doi.org/10.3929/ethz-b-000312914> →

Rights / License:

[In Copyright - Non-Commercial Use Permitted](#) →

This page was generated automatically upon download from the [ETH Zurich Research Collection](#). For more information please consult the [Terms of use](#).



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Gas Cost Analysis for Ethereum Smart Contracts

Master Thesis

Christopher Signer

Wednesday 14th November, 2018

Advisors: Prof. Dr. M. Vechev, Dr. H. Ritzdorf, Dr. P. Tsankov

ChainSecurity AG, Zurich
Department of Computer Science, ETH Zürich

Abstract

Ethereum blockchain smart contract development has grown immensely with large corporations and many startups adopting the new technology for their own use. The distributed network processes transactions including smart contract code executions on the mining nodes when generating a block but also on many other nodes for validation. To prevent Denial-of-Service attacks through extensive computation, Ethereum introduced a gas system that limits the computational, memory and storage resources one contract can use. For certain Ethereum Virtual Machine (EVM) instructions, the gas cost is statically determined, while for others it is dynamic.

While the total gas cost of an executed transaction can be observed on the blockchain, the gas cost cannot generally be predicted ahead of time. Local execution tests might be incomplete, require a costly setup or can be unreliable due to outdated states. Additionally, the overall gas cost is insufficient for developer guidance, as it does not reveal where in particular within a transaction the high costs originate from.

In this thesis, we address the challenges mentioned above by providing Visualgas, a tool to visualize gas costs in depth to support the simpler, more gas-efficient development of smart contracts. Visualgas makes it easy for developers to test gas costs and explore best and worst case executions before deployment. Furthermore, we give a detailed overview how the costs relate to different parts of the code. Visualgas collects transaction traces using a fuzzer, maps the EVM instructions to the code, interprets the costs in the traces and maps them to the EVM instructions, aggregates the results to get costs in relation to code, analyses for patterns and makes the results accessible in a web interface.

Especially with how internal functions translate into EVM compared to calls to other contracts we see that the gas costs can wildly differ based on implementation details. Thus part of the development process should be following through the control-flow for the most common transactions in Visualgas to see where unexpectedly high costs can occur and possible adapt smart contracts accordingly.

Acknowledgments

I would first and foremost like to thank my advisors, Dr. Hubert Ritzdorf and Dr. Petar Tsankov, for their advice, support and feedback. Thank you to ChainSecurity AG for providing the infrastructure and the webserver code base. Thank you to Nodar Ambroladze for providing the fuzzer, a product of his master thesis. And last, I would like to thank Professor Martin Vechev for supervising the thesis.

Contents

Contents	iii
1 Introduction	1
1.1 Background	2
1.1.1 Solidity	3
1.1.2 Ethereum Virtual Machine (EVM)	3
1.1.3 Gas costs	3
1.1.4 Truffle	4
1.2 Related Work	4
1.2.1 MadMax: surviving out-of-gas conditions in Ethereum smart contracts	5
1.2.2 eth-gas-reporter	5
1.2.3 Remix	6
2 Manual Analysis	7
2.1 Packing	7
2.2 Type Choice	8
2.2.1 uint size	8
2.2.2 byte array	9
2.3 Correct Operator and Control Flow Choice	9
2.4 Tautologies	10
2.5 Constants	10
2.6 Factories	11
2.7 Refunds	11
2.8 Inlining	11
2.9 External vs Public	11
2.10 Short Circuiting	11
2.11 Reducing Computational Work and Store Operations	12
2.12 Private & Internal Functions/Variables	12
2.13 Simplifying Expressions	12

3	Methodology	13
3.1	On Static and Dynamic Program Analysis	14
3.2	Ethereum Clients in Regards to Trace and Truffle	14
3.3	Fuzzer and Webserver	15
4	Implementation	17
4.1	Collecting Data	17
4.2	Program Counter to Code Mapping	19
4.3	Parsing Transactions	19
4.3.1	Gas costs of calls	19
4.3.2	Refunds and Revert	20
4.3.3	Invalid	20
4.4	Cost Aggregation	20
4.5	Return Point Costs	21
5	Case Study	23
5.1	User Interface Layout and Running Visualgas	23
5.2	Interpreting the Visualgas Results	24
5.3	Performance	27
6	Conclusions	29
	Bibliography	31

Chapter 1

Introduction

Ethereum is after Bitcoin the most well known blockchain. It stands out for supporting already in the initial design mainly one feature, namely a nearly Turing complete smart contract framework. Currently Ethereum clients are running Ethereum Virtual Machine (EVM)[11], an environment that runs its own binary language. Multiple high level languages compile to EVM, including Solidity which is the most popular and Vyper, a more recent addition which strives to improve security. Development is also in progress for Ethereum flavored WebAssembly (ewasm)[10], a runtime environment for smart contracts with the goal to be portable but running code nearly as fast as native machine code. It is an adaption of WebAssembly and thus has a large code base to build upon that includes compilers from various high level languages to Wasm.

Gas costs are a method to prevent Denial-of-service attacks for code execution on Ethereum. Each block has a gas limit voted upon by the miners and changed according to votes and an algorithm that limits the resources all transactions of the block can use altogether. Miners can choose when building a block which transactions to add to their block and will do so also under consideration of the gas price given with the transaction. While the gas cost is a metric of how much resources the transaction needs, the gas price states how much ether the transaction sender is willing to pay per gas used. The miners keep the transaction fees as part of their reward and thus, obviously, tend to pick transactions with higher gas prices to increase their profit. The gas price is regulated by supply and demand. Senders of transaction can try to prioritize their transactions by increasing the gas price. Saving gas thus not only serves the purpose of reducing transaction fees but also allows more transactions to take place per block and as such overall.

The gas costs are defined for EVM operations in the Ethereum yellowpaper by Dr. Wood [16]. Solidity code is thus on an abstract layer where gas costs are not fully aparent. Stack management, compiler optimizations, loading of

function arguments and more are not directly visible in Solidity but added by the compiler to the EVM binary code and cost gas like any other instructions. Some of the gas costs are dynamic because they depend on the state during runtime. This means they can hardly be predicted with static analysis in most cases. While the compiler does output a source mapping to the program counter (PC), the mapping is not so easy to use as the source code assigned to certain PCs is often a full contract or function in case of putting function arguments on the stack for example. Additionally the source code of a single contract can be across multiple source files due to inheritance and how libraries work. Some PCs can not be assigned to a single place in the code as they are either compiler generated or like a hidden internal function to access for example an array element which is the same binary code for multiple array accesses in multiple functions.

Working with traces is another challenge. To collect traces we have to run the program on an Ethereum client (more to this in section 3.2). The traces themselves do not show directly where all costs arise due to how the gas counter is kept. In particular, the costs for calls, contract creates and refunds can only be read off with further work or not at all without more data. We are also interested where we enter and return from internal functions which is hard to detect within the trace.

To show the data in a graphical user interface, we have to decide on an output format and in our case we aggregate costs of statements and function signatures, thus we have to extract where these elements are located within the source code.

We hope Visualgas can help developers to better detect where high gas costs can occur to improve their code before deployment and thus write safer smart contracts.

In the following sections of this Chapter, we first introduce Solidity, Truffle and Gas costs in more depth and some related tools. In Chapter 2, we show some patterns and details to adhere to to save gas in general. In Chapter 3, we present the requirements for Visualgas and discuss in particular some of the implementation choices. Chapter 4 gives further details on the implementation while Chapter 5 illustrates Visualgas with an example, before we end the thesis with the conclusion in Chapter 6.

1.1 Background

In this section we introduce some of the relevant background for this thesis, including the smart contract language Solidity, the definitions of gas costs, the development framework truffle and previous work in the area of gas analysis.

1.1.1 Solidity

Solidity is a statically-typed programming language for writing smart contracts. It was developed by the Gavin Wood et al. from the Ethereum foundation. It is similar to JavaScript and C++. Some of the more notable language features are the support of inheritance and function modifiers, the latter which allows for checking properties on function calls. Variables can be defined to be stored in the stack, memory and storage/state within some restrictions with high gas costs associated to storage variables. There are also multiple features in regards to specialized use cases as for example Ether units, events for logging which applications can follow, block and transaction properties such as the origin (who sent the transaction), how much gas is left and more.

Many smart contracts make use of so called Ethereum Request for Comment standards (ERC) with the token standard interface ERC-20 being the most popular. Some of them are implemented services which are used widely and others like the ERC-20 are interfaces standards to allow for compatibility in applications that serve between different contracts, such as a trading platform in case of tokens.

1.1.2 Ethereum Virtual Machine (EVM)

The EVM is a deterministic virtual machine, designed to securely run untrusted code in a global blockchain network. It thus uses the gas mechanic to limit the extent of smart contract executions. Smart contract executions are run sandboxed and can only modify their own state and interact with other smart contracts by transmitting single arbitrary-length byte arrays.

The EVM has a stack, memory and a key-value storage. Stack elements, most instruction arguments and storage keys and values use 32-byte words, especially also any integer instructions. The stack is limited in size to 1024 elements, although only the top 16 32-byte words can be accessed without removing elements from the stack. This limits the amount of stack and memory variables that can be used within a function, as the memory addresses also have to be kept as stack variable. The storage is the only permanent way to store nonconstant data between program executions.

1.1.3 Gas costs

Gas costs are associated to Ethereum Virtual Machine (EVM) instructions. The detailed list can be found in Appendix G. and H. of the Ethereum Yellowpaper [16]. In general, EVM instructions which lead to the blockchain size growing, such as storing and changing state variables, creating contracts and emitting events, are very expensive, in addition to the base cost of 21,000 for any transaction. The base cost is applicable for transactions whether they

execute code or only transfer Ether and an additional cost of 4, respectively 68 is added for any included data or code per byte, with the higher rate for non-zero bytes. High costs also apply for call operations. This is mainly due to the permanence of the resources used. Memory cost is another component that has quadratic growth, yet as a general rule is less expensive than storage but more expensive than the stack.

A special case are storage variables. Writes that set a storage slot from a zero value to a non-zero value cost 20'000 gas. Writes to a non-zero storage slot are 5'000 gas. If a non-zero storage slot is set to zero a 15'000 refund of gas is added to the refund counter. A refund can also be gained through self-destructing a contract. The value of the refund counter but at most half the transaction cost is given back after the full transaction is executed, provided the transaction is not reverted or otherwise exceptionally halted. For simplification, EVM implements besides the revert operation which returns all remaining gas only the out-of-gas exception. Any other exception as for example an invalid jump or an invalid opcode will use up all remaining gas and thus exit as an out-of-gas exception. Revert and the out-of-gas exception undo any state changes.

1.1.4 Truffle

Truffle [7] is a development environment which includes a testing framework and provides various services related to private and public EVM based blockchain networks. The Truffle Suite as a whole is a very popular blockchain development suite, especially for decentralized applications (Dapps), as the suite provides further applications, particularly ganache, a personal blockchain for Ethereum development which we will discuss further in Section 3.2. The automated testing which is based on Mocha is often used extensively in Ethereum smart contract development and easy to use due to the integration with ganache. Truffle also provides debugging of transactions via command line and an easy setup of smart contract deployment to public and private networks.

1.2 Related Work

While there are many collections of 'do's and don'ts' when writing Solidity code to save gas, there are also multiple tools that are currently often used. Besides the following which are presented in more detail there is notably also a profiler from Logvinov[14] that should serve a similar purpose but is still under development or remains unfinished at 0xProject as of beginning of November 2018.

1.2.1 MadMax: surviving out-of-gas conditions in Ethereum smart contracts

MadMax by Grech et al.[13] is a static analyzer to find gas-focused security vulnerabilities. Causing an out of gas exception at a specific place can leave a smart contract in an undesired state where funds can be locked in or other undesired behavior can be produced. If an out-of-gas exception occurs, the state is reset to before the transaction, thus no progress takes place to get out of the undesired state and thus causing Denial-of-service. In comparison to normal programs no system reset or manual data modification can take place to get out of the situation unless it is purposely planned for ahead of deployment which can on the other hand create trust issues. Thus, the best measure is to prevent it with better, specific coding regarding gas exploits. To find these situations is no easy feat, thus MadMax combines a control-flow-analysis-based decompiler and declarative program-structure queries to find possible problems.

While MadMax finds vulnerabilities, it does not however analyze gas costs for the general purpose of saving gas and giving insight where the costs originate. Lower gas costs can increase limitations before out-of-gas exceptions occur and thus Visualgas complements MadMax.

1.2.2 eth-gas-reporter

The eth-gas-reporter [8] is a Mocha [4] reporter for Truffle. Mocha is a test framework for Javascript running on Node.js. Truffle makes use of Mocha for running Solidity tests. The eth-gas-reporter runs Truffle tests and collects data to match it together with the source, to provide gas costs for functions that are called. To do so it modifies the contracts which influences the execution and does slightly change the behavior in regard to gas usage. While it shows nicely what gas costs to expect for the public functions, it does not show where the gas costs originate. Gas costs are associated by the transaction input to functions, thus functions which are not or can not be called from tests are not listed separately. The data given is thus very nice for users to see how much they should expect to pay for specific transactions. The consideration of the gas price to display real currency costs helps to serve that cause. The eth-gas-reporter does not however help to determine in specific transactions why the gas costs are high and is limited to the data collected from the test cases. It also does not show the gas costs of tests where the transaction aborts which could be of interest as an out-of-gas exception could be the reason for it. Visualgas breaks the costs further down to within internal functions and shows gas costs also at return points due to execution errors.

1.2.3 Remix

Remix is an open source web IDE for Solidity smart contracts that also facilitates testing, debugging and deploying. The overall costs of transactions broken down to transaction cost and execution cost are given. Debugging allows for stepping through the debug trace where gas costs can be observed, although only statically. It is easy and quick to use for small applications, but has disadvantages over Truffle for larger applications, specifically in combination with Dapps based on Javascript. On the other hand the graphical user interface of Remix allows for easier debugging without heavy use of the command line. In regards to gas costs, Remix is well suited for investigating specific transactions, whereas Visualgas shows aggregated results of many transactions and facilitates finding specific transactions which should be investigated.

Chapter 2

Manual Analysis

This Chapter lists simple measures that can save gas which were analyzed in real contracts. We intend to provide guidance for developers and also partly as assistance what to look for in Visualgas. It was tested in Remix with the Solidity 0.4.24 compiler release and the 0.5.0 experimental features and optimization both on. Note that execution costs in the following differ from transaction costs in that the fixed cost of 21,000 gas for a transaction and the variable cost based on argument size is not included in the execution cost.

2.1 Packing

Reordering state variables and fields for structs can decrease the storage space needed. Each storage space is 256 bits. State variables and fields of structs that are in storage will share a storage slot if their size allows it according to their listed order, filling greedily top to bottom. The compiler optimization will do writes and reads to multiple state variables or fields respectively together, if they are in the same function. This can lead to less sload or sstore instructions in some situations, especially if the storage operations are written right after each other in the code. Packing should thus not only be according to size but also to the relations, whether the variables and fields are often read or written in combination. No packing is done in memory.

```
1 contract A {  
2     uint128 public a = 1;  
3     uint256 public c = 0;  
4     uint128 public b = 0;  
5     function setSlotb (uint128 arg) public {  
6         b = arg;  
7     }  
8 }
```

Listing 2.1: Inefficient

```
1 contract B {
```

2. MANUAL ANALYSIS

```
2  uint128 public a = 1;
3  uint128 public b = 0;
4  uint256 public c = 0;
5  function setSlotb (uint128 arg) public {
6      b = arg;
7  }
8  }
```

Listing 2.2: Packed

Listing 2.1 results in transaction cost of 195,454 gas to deploy and calling setSlotb has 20,409 execution cost with a uint larger than zero (excluding transaction cost in the latter case). Listing 2.2 analog is 193,485 to deploy and 5,411 to execute setSlotb. This is due to a and b being packed into one slot in listing 2.2 and thus writing to b does not set a new slot to a non zero value.

2.2 Type Choice

The choice of datatypes can change the gas costs, as we show in the following examples.

2.2.1 uint size

uint256 generally provides less deployment costs and execution costs over other uint sizes, unless it has to be stored. This is due to the additional masking from 256 bit sizes to smaller uint types. The additional cost for using more storage is immensely more significant than the masking overhead and thus smaller sizes can make sense if we can save on storage. Using temporary variables to work with uint256 instead of on smaller size uint's in that case will allow us to profit from packing and partly less overhead due to type choice. The compiler might store some storage variables temporarily on the stack anyway to prevent multiple storage load or store operations in which case they use up one stack slot of 256 bit each.

```
1  contract A {
2      function add128 (uint128 arg) public pure returns (uint128 sum){
3          for (uint128 i = 0; i != arg; ++i) {
4              sum+=i;
5          }
6      }
7  }
```

Listing 2.3: Sum up with uin128

```
1  contract B {
2      function add256 (uint256 arg) public pure returns (uint256 sum){
3          for (uint256 i = 0; i != arg; ++i) {
4              sum+=i;
5          }
6      }
7  }
```

Listing 2.4: Sum up with uint256

The uint128 variables in listing 2.3 result in intermediate internal conversions to uint256 to do instructions and back again. Thus the execution cost is 790 gas versus 662 for listing 2.4 for a function argument of eight. The deployment cost difference is at 115,627 gas for uint128 compared to 100,071 gas for uint256.

2.2.2 byte array

For arrays of bytes the type `byte[]` should practically never be used. `bytes` should be used instead as `bytes` has a more compact format where the full 32 bytes are used in comparison to `byte[]` which uses up 256 bit (32 bytes) per byte.

```
1 contract A {  
2   byte[] b;  
3   function g2(byte[] a) public returns(uint256){  
4     b = a;  
5     return uint256(b[2]);  
6   }  
7 }
```

Listing 2.5: `byte[]`

```
1 contract B {  
2   bytes b;  
3   function g1(bytes a) public returns(uint256){  
4     b = a;  
5     return uint256(b[2]);  
6   }  
7 }
```

Listing 2.6: `bytes`

Listing 2.5 ends up at 196,459 gas while listing 2.6 takes only 185,363 to deploy. For executing both functions with the argument `["0x0a", "0x0b", "0x0c"]` it's 230,667 gas versus 21,647 gas in the `bytes` case.

2.3 Correct Operator and Control Flow Choice

The unequal operation `!=` is generally cheaper than less than (`<`) or greater than (`>`). Specifically `> 0` occurs a lot which can be rewritten for uint datatypes as `!= 0`. In many cases this saves 3 gas for using unequal and another 3 gas in case it is unequal zero as the compiler can turn a greater than or less than instruction and an `iszero` check into an `iszero` check only. Another measure is to reuse stack variables. E.g. integers for iterating in for loops can not be reused if defined in the for loop and should thus not be used unless there is no other possible use for that variable. Do while will skip the first check whether the stop condition for a loop is fulfilled, thus saving gas over for or while loops in case the loop is always executed at least once.

2. MANUAL ANALYSIS

```
1 contract B {
2   function f1(uint256 b) public pure returns (uint256 sum) {
3     for (uint256 a = b; a != 0; a--) {
4       sum+=a;
5     }
6   }
7 }
```

Listing 2.7: Introducing an additional variable to loop

```
1 contract B {
2   function f1(uint256 a) public pure returns (uint256 sum) {
3     for (; a != 0; a--) {
4       sum+=a;
5     }
6   }
7 }
```

Listing 2.8: Using existing variables to loop

Not reusing the variable as in listing 2.7 leads to a fixed overhead of 5 for an additional push and pop each plus one swap more for 3 gas per iteration.

2.4 Tautologies

uint and address imply that their value is greater than or equal zero by definition. Thus any such comparisons should statically be evaluated as true and smaller than 0 always as false. This is already handled by the compiler in case that the value of the variable is known at compile time but it can be handled in general. Similarly the array length is always greater than or equal to zero, which the compiler does not optimize.

```
1 contract A {
2   function f1(uint256[] a) public pure returns (uint256 sum) {
3     if (a.length>=0) {
4       uint256 temp = 0;
5       for (uint256 i = a.length;i!=0;i--) {
6         temp = a[i-1];
7         if (temp>=0)
8           sum+=temp;
9       }
10    }
11    return sum;
12  }
13 }
```

Listing 2.9: Introducing an additional variable to loop

Removing the if clause in line 3 in listing 2.9 will save 3,630 gas in deployment and 38 gas in execution. Removing the redundant if clause on line 7 will save another 2,086 on deployment and 23 per length of the input uint[] a.

2.5 Constants

Using constant state variables can save gas as it allows for avoiding expensive load operations by placing in the value directly at compile time instead.

We should avoid state variables that are never assigned to in contracts and write them as constants instead. Due to inheritance if some child contracts are not provided it can be wrong to blindly turn any such state variables into constant though.

2.6 Factories

Factories can save a lot of gas if a contract is intended to be deployed many times. The basic structure can for many cases be done very similarly and thus an estimate of how many times it would have to be deployed so that one would profit from a factory can be calculated based on the contract code. It has to be taken into account that the deployment of factories can easily exceed the block gas limit and thus not be possible.

2.7 Refunds

Freeing up many storage slots in one transaction can quickly result in the refund counter surpassing half of the full transaction cost. In such a case or in case of self-destructs which can lead to a high refund counter as well we should evaluate how much of the refund can actually be used, as the refund can be at most half of the transaction cost. Thus freeing up storage slots or deleting contracts can make more sense in combination with other operations if possible.

2.8 Inlining

In case some internal or private function is called only at few places (although possibly many times if it is in a loop) it can save gas to inline it. The cost difference is small and inheritance as well as code duplication and readability can be reasons to choose not to do this.

2.9 External vs Public

It is advisable to use the automatic getters of public state variables over own getters if possible. Special care should be taken to avoid user defined getters in addition to an automatic getter.

2.10 Short Circuiting

Having the right evaluation order in conditions for control flow structures can save gas due to short circuiting. Changing for example the order of two comparisons with one using a storage variable such that the storage access

is after the other comparison can make a difference. The evaluation order can have an impact on correctness, thus it has to be done with care.

2.11 Reducing Computational Work and Store Operations

An article by Chen et al. [9] on a so far unpublished tool GASPER shows us a few more patterns where gas is wasted. Dead code and opaque predicates should be avoided (pattern 1 and 2). Similarly, there is improvement that can be done regarding repeated reads and writes to the same variable. The optimizer does sometimes but not consistently avoid repeated reads and writes on storage variables, especially in loops. Copying from storage to stack or memory and then working on it or building and then writing back can reduce costs immensely. Overall, multiple writes or reads to the same storage variable in a single transaction should be avoided if possible.

The constant evaluations by the Solidity compiler (solc) are not optimal. Some loops can possibly be merged or some calculations can be done out of the loop even if dependent on input (pattern 5 and 6). The last pattern 7 describes how some comparisons which cannot be determined at compile time can possibly be taken out of a loop if the outcome is unilateral.

2.12 Private & Internal Functions/Variables

Unused private or internal (state) variable should be avoided. Solc does not give a warning nor are they removed by the compiler. Private or internal variables use up storage space and are thus costly. However, unused internal and private functions are removed by solc although without any notification.

2.13 Simplifying Expressions

Sometimes, an expression can be swapped out for an equivalent expressions that is more gas efficient. As an example, computing $a + a$ requires two gas less than computing $2 \cdot a$. This particular example is not optimized by the compiler.

Chapter 3

Methodology

The following four goals guided the decision making, design and development of Visualgas:

Usability Visualgas should provide a smart contract developer with meaningful insights that complements other tools.

Simplicity It should be simple to use without a large setup or and should not require any reformatting of the project.

Coverage The coverage in terms of statements, branches and paths should be high to highlight the different costs.

Fine granularity The gas costs should be shown in detail to provide meaningful development input.

We intend to take smart contract code as input and return a visualized view of the gas costs in the code. We show the gas costs within functions at the return points, such that the cost difference of various control flows can be tracked better. As Truffle is one of or the most popular smart contract development framework we want to directly support Truffle projects as input for simplicity. We require a deployment script along with the Truffle project as the interest is mostly on the large number of transactions after deployment. The owner of the smart contract can control deployment while he can not freely control what interactions occur with the smart contract afterwards if deployed on a public network. The Solidity compiler solc is under active development including breaking changes occasionally. Also the optimization between different compiler versions changes at times, so we plan to support multiple compiler versions. Visualgas does not support the use of inline assembly in Solidity. Inline assembly can break many invariants that the solc compiler would comply with and thus analyzing contracts with inline assembly can create unexpected or false behaviors.

Visualgas has several dependencies on other tools. As usability is one of the goals, we provide a docker container. This especially isolates the solc-js installed with the node package manager for Truffle and the installation of solc through py-solc from the host system to not break any other dependent applications. The Go-Ethereum (Geth) client for the Fuzzer also has to be a certain version which is specifically patched. Through the docker image, we can keep a consistent version of these tools in case a new version of a tool has breaking changes.

3.1 On Static and Dynamic Program Analysis

While the costs of many EVM instructions are fixed, the control flow of the program execution and the input influence the cost of the other EVM instructions. A static analysis to show gas costs throughout the program is very difficult as it for one would have to be done on the binary (EVM) code, as gas costs are set for EVM instructions. The abstract interpretation would have to model many of the internals of the Ethereum Virtual Machine in great detail. Some well defined properties can be shown nicely with static analysis as done in MadMax [13], in example proving that out-of-gas exceptions can only occur at certain places where it causes no unwanted behavior in any possible transaction. While this is of great use to guarantee security aspects it is of less help to save gas and pinpoint where to save it. Static analysis can contain false positives or might have to over-approximate, while dynamic analysis is more precise but might not have complete coverage. We choose to use dynamic analysis, as it can more easily show the usual costs in various transactions as traces can provide us with the detailed information we need to observe the gas costs.

3.2 Ethereum Clients in Regards to Trace and Truffle

There are three popular Ethereum clients. Parity and Go-Ethereum (Geth) clients run the large part of the Ethereum main net. Ganache is part of the Truffle suite and a client for development purposes. Parity provides a trace of all calls, creates and suicides with some detailed information each, a list of all state changes with the values before and after the transaction and also a detailed trace of each EVM step that does not however store the state of the stack. Ganache and Geth both provide very similar debug traces to each other including the stack state but no overall state differences. In particular, the state is tracked but without knowing what is in a certain storage slot before overwriting it, unless a storage load instruction already occurred or it was already written to in the same transaction. Ganache performs much better running Truffle tests and has support for some features like going forward in time and creating snapshots that are an advantage for supporting

the use of Truffle tests and collecting traces thereof. We can run the tests and query the traces with RPC. However, during testing it turned out that Ganache traces would be incomplete or wrong. Geth on the other hand does not natively process more than one block per second, thus slowing down Truffle tests immensely. It also does not support many common Truffle tests due to the Ganache specific features. We can circumvent these limitations with the Fuzzer as detailed in the following section.

3.3 Fuzzer and Webserver

The Fuzzer by Nodar Ambroladze runs in the Go-Ethereum (Geth) client. We first run the deployment script on Ganache to record the deployment and initialization transactions. The Fuzzer replays them in Geth and then generates transactions to run all public methods and uses a feedback loop to adjust arguments. It is optimized to reach a high coverage by also creating snapshots during execution to return to a known state, for example in a case that a method that self-destructs a contract is called. The Fuzzer also extracts timestamps from the deployment transactions which it uses as block timestamps during fuzzing to possibly also execute methods which require a certain time to have passed. We can export the trace from within the Fuzzer in the processing module where in most cases the properties might be checked. The adaptations to the Geth client remove the limitation of one block per second. Out of 31,000 random real world contracts with verified sources from Etherscan [2], about 30,400 contracts that have less than 6,000 EVM instructions and the other 600 have more. Running the former in the Fuzzer with a limit of 5,000 transactions each will result in an average code coverage of about 88% with a median code coverage of over 94%. The larger contracts have a much lower code coverage with that many transactions, thus we will use a limit that depends on the project size.

The webserver is heavily based on ChainSecurity's web application for Securify [15] from Tsankov et al. The Python flask framework [3] serves the content. The Javascript CodeMirror [1] text editor implementation is used to show the Solidity code and further JavaScript to make the requests to the flask back-end and handle the polling of the result. The flask server starts a separate Linux process to run the Fuzzer and then the analyzing of the transactions. Finally, it returns the result to the front-end for creating the visual elements.

Chapter 4

Implementation

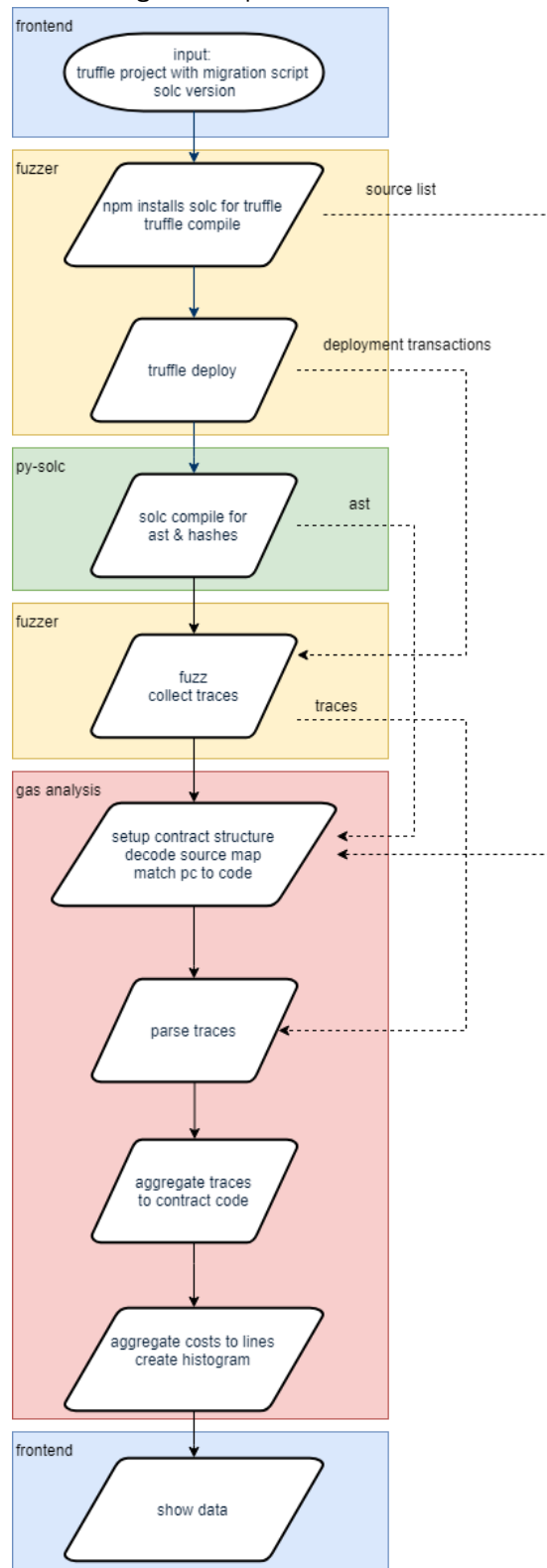
In the following sections we will go into details on how the Truffle project is processed to create the results of Visualgas.

4.1 Collecting Data

First the set solc-js compiler is installed in the node package manager for Truffle. Second the contracts are deployed to Ganache using Truffle compile and then Truffle deploy. The deployment transactions are exported from Ganache (through Truffle) and stored. The Truffle compiling process lists all source files in an order which is the exact source list used in the encoded source mapping format of solc. Truffle does not explicitly output this list in the build artifacts, thus the need to parse it from the printed output. The build artifacts which contain abi, source mapping, bytecode and more are another byproduct to be used later on. In addition we use py-solc to change the solc if necessary, to then gather the function signature hashes and solc abstract syntax tree (AST). The ASTs given in the Truffle build artifacts are unfortunately in a hardly reusable format and its format is not publicly documented. The Fuzzer is then used to replay the deployment transactions on Go Ethereum and then to collect traces by running many transactions with semirandom data. Comparing the swarm hash also allows for matching deployment addresses to the corresponding contracts during the fuzzing. The traces contain what instruction it is, which program counter (PC), the remaining gas, the gas cost of the step, the call stack depth and the stack and memory state for each step. It also contains some information on storage, specifically any slot that is written to or read from is listed in the storage dictionary from that specific step onwards. The trace does not consider refunds but the receipts can also be gathered, which include the total cost.

4. IMPLEMENTATION

Figure 4.1: process flow overview



4.2 Program Counter to Code Mapping

For easy further processing, we build an object structure for each contract. The central part is decoding the source mapping from the Truffle build artifacts and to further process it to map the program counters to specific lines. The general decoding is described in the Solidity documentation [12]. Of interest are the actual source code ranges given for specific PCs. Generally, we want to aggregate on a source code statement or source code line basis to produce a format that can be displayed well. A common case is a PC pointing to a source code range which is part of a statement. Using a visitor for the AST we look for the statement that contains the code range and then count what line the start is and over how many lines the statement spans (often just one). However, PCs related to control flow structures, contract definitions and function definitions point often to the whole element including the body. For such cases, we just aggregate per line that the element starts on. At this point, we also determine on which line in the source code functions end, to know on which line to annotate implicit returns.

4.3 Parsing Transactions

The next step is to load the transactions into their object structure and parse the traces. We notably want to eventually add to the PCs, which are mapped to the source code now, what the actual costs are as well. While for many instructions the cost is static, most of the high costs originate from the few other instructions. We know which contract is called from the receipt. We then have to keep track of the current contract with a call stack and reading off from the stack in the trace which contracts we enter in call instructions and observing with the depth when we return (call referring here also to `delegatecall`, `staticcall` `callcode` and `create` instructions). Furthermore, how the gas costs and gas remaining are shown at calls is more involved. Since we do not have to worry about book keeping to detect out of gas exceptions as the trace would indicate this, we focus on deducing the actual gas consumed by instructions.

4.3.1 Gas costs of calls

If the call is an Ethereum transfer to a non-contract address we can detect this due to the depth not becoming larger despite the call instruction. The call stipend of 2300 is included in the gas costs for the call together with possibly more gas, but fully returned. We can read off how much gas is passed along from the stack. The gas remaining and gas cost do not show that the gas is not used and returned and thus we reduce the gas cost of the call accordingly.

On the other hand, in case a call is made to a contract address, the gas cost shown at the call is again the full gas passed along. The gas remaining then shows that amount of gas that the newly called contract contains until we return from that contract. Thus, we can deduce the actual gas cost by subtracting the remaining gas right after the call from the gross gas cost of the call. It is then easiest to just adjust the gas remaining of the whole trace based on the gas costs which are now correct.

4.3.2 Refunds and Revert

Refunds are not explicitly given in the trace. We can often read off the previous value in storage before an `SSTORE` and thus together with the new value definitely say whether a refund of 15000 takes effect. But in case that the storage value has not been written to in the same transaction before, the value is not in the storage and in case we write a zero, a refund could happen. We can based on the total gas used for the transaction from the receipt deduce how many of the possible refunds actually take place. If we would keep track of when the Fuzzer reverts to a previous snapshot, further processing could allow to read storage values that were written in previous transactions. A special case are reverts. We can read off a `REVERT` from the last instruction of the trace. In case of a `REVERT`, no refund occurs at all as any storage writes are undone. Analog, we can check if the last instruction is a `SELFDESTRUCT`. If so, we have an additional refund of 24000 gas. Additionally, we have to check if the refunds exceed half of the pre-refund transaction cost and accordingly limit it to that number.

4.3.3 Invalid

Exceptional halting is done either in orderly fashion with a revert or alternatively with an invalid instruction that effectively burns all gas to create an out-of-gas exception. Returning from one contract to another or ending the trace with an invalid instruction does not show the right gas cost. We can, however, either observe how much gas remains at the next step in the trace or in case the trace ends at this step, it burns all remaining gas. No gas is refunded for a call that ends with invalid.

4.4 Cost Aggregation

We can now collect the cost from all traces at the respective PC in the contract structure. Iterating over all PCs from contracts then lets us aggregate the values to extract for the front-end. We gather them per source code line range and classify the instructions to give more details regarding where the costs originate. For PCs that are never executed we can potentially fill in static values but otherwise it is a rough estimate at most. The histogram

per line is calculated across all source code data 4.1. We add the costs of all line ranges into a Numpy array. To define the bin edges for 10 bins we sort all values. We then calculate at which indexes of the sorted data points the bin edges are by linear spacing 10 points in the range of 0 to the number of data points. And finally, we interpolate the sorted values to evaluate which values the bin edges are at.

```
1 bin_edges = np.interp(np.linspace(0, len(data_points), 10),
2                       np.arange(len(data_points)),
3                       np.sort(data_points))
```

Listing 4.1: Calculating the bin edges for the histogram

Thus, we can now classify the line ranges according to their costs and the bin edges.

4.5 Return Point Costs

Evaluating what the costs are when returning from a function has a few challenges. First, we have to find the PC positions where a function is entered. This is nontrivial as calls are only for entering another contract, but within a contract a function call is basically a JUMP instruction, almost as any other. Looking at the exact points where functions are entered and exited, we can notice a few things though. There is always a JUMP followed by a JUMPDEST instruction. Both are not compiler generated and associated to line range. When entering, JUMPDEST always points to the whole function we enter. JUMP sometimes does but not always. Exiting the function is similarly in that JUMP always points to the whole function from which we exit. Additionally, we can obviously only exit if we have a previous function to return to, or the trace ends respectively and we can also check that the PCs have to match up. The gas cost of the function execution is a simple subtraction of the gas before entering and after exiting. More challenging is to determine on which line range to associate returning from the function. We remember the last instruction that did not point to a whole function or contract before exiting, as that often occurs but is not of much use. We then check if the source identifier of it matches the source identifier at the JUMP and the indicated line is in the function given by JUMP. If so, we can assign the return costs to that line range. If not, we assign the cost to the last line of the function that we are leaving. It occasionally occurs that multiple functions are left directly one after another and in that case, the last statement not pointing to a function can be from another function that we already left, thus giving us a false result.

A trace ending with a REVERT instruction is a special case again, as at that point we return from all functions and not with the usual JUMP and JUMPDEST one at a time. Thus, we keep track of the lines where the func-

4. IMPLEMENTATION

tions are called on the way, such that we can report back these line ranges as where the functions return.

Case Study

We show Visualgas in use based on a TimedCrowdsale contract. The full code is available at [6]. It is based off the equally named contract from the OpenZeppelin Solidity library [5] with an added migration script. We describe the interface and how the results can be read off and interpreted in the following sections.

5.1 User Interface Layout and Running Visualgas

In Figure 5.1, we see the code editor with the buttons above for submitting a Truffle project. Visualgas supports ZIP upload and cloning a public Git repository. The drop-down allows to pick the compiler version, which has to adhere to the version pragma of the submitted source files for successful compilation. The Truffle project should be on the top level in the directory, in example a ZIP of the project folder content or, alternatively, on the top level of the Git repository. Also the truffle.js or truffle-config.js configuration file requires a network configuration called development with the host given as localhost and an asterix as wildcard for the network id. The optimization flag will correctly be taken into account from the Truffle configuration file and dependencies to the npm openzeppelin-solidity package are supported, but no others.

Once Visualgas is done running, the source files are listed to the left of the code editor view in a list for navigation between the different files, as shown in Figure 5.1. Below is a table that displays detailed information regarding the gas costs of all instructions that are linked to the row, where the cursor currently is, in the code editor. The instructions are split by the types stack, storage, memory, control flow, math / logic and other. The type other notably includes instructions for return and call data, logging, create and globally available variables like getting the balance of an address. The code editor now shows the code of the chosen source file. A click on the

Figure 5.1: Screenshot of the editor view of the results for a TimedCrowdsale



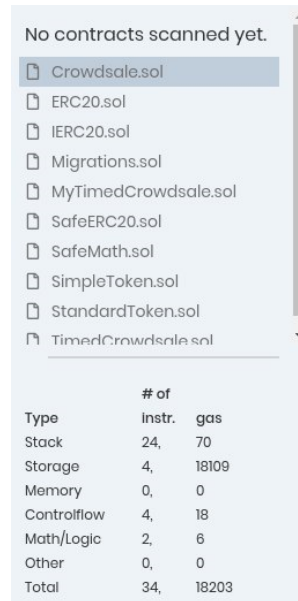
button with the graph next to the Scan button switches between viewing the gas cost histogram as shown in Figure 5.1 and info symbols to display more information on the gas costs of function return points.

5.2 Interpreting the Visualgas Results

The gas histogram gives a quick overview of the gas costs. Together with the gas table, we can find also what kind of instructions cause them. We look at the return points to investigate the costs of specific control flows. We inspect a transaction that calls function `buyTokens` on the `TimedCrowdsale` contract. See Listing 5.1 with relevant extracts from the `TimedCrowdsale` contract and Listing 5.2 for further extracts from the `Crowdsale` contract to which we will refer.

- Gas is used in the function signature mainly for unpacking call data as it is a public function. Internal and private functions do not have high gas costs associated to the signature, as no call data has to be placed into the stack and memory respectively.
- Line 116 puts the message value on the stack which can be clearly read

Figure 5.2: Screenshot of the navigation view and gas table for a TimedCrowdsale



Type	Instr.	# of	gas
Stack	24,		70
Storage	4,		18109
Memory	0,		0
Controlflow	4,		18
Math/Logic	2,		6
Other	0,		0
Total	34,		18203

off from the gas table which only has a few transactions in that regard.

- The call to `_preValidatePurchase` itself is inexpensive and consists of control flow and stack operations. We have to check the function `_preValidatePurchase` in the `TimedCrowdsale` first to see how expensive it is. The function modifier `onlyWhileOpen` has a return point with gas costs of 2,309 and another that costs 825. Looking at function `isOpen` we see that there are two storage locations at average costs of 400 total, thus, two storage load operations. With one return point at a cost of 250 and another at 463, the second comparison for the closing time is clearly not always executed. Which propagates back to different costs in the `onlyWhileOpen` modifier. We can deduce that the modifier is a return point in `_preValidatePurchase` although not marked as such.

Further following the call to `super._preValidatePurchase`, we end back in the `Crowdsale` contract. It checks two `require` statements which are both relatively inexpensive. From the return point information we can read off that `_beneficiary` is never address 0 at this point. At the second `require` statement we see two different costs for the return point, thus we can expect `_weiAmount` is zero in some cases. Back in the `_preValidatePurchase` from `TimedCrowdsale` we see two return points with nearly identical costs at the call to the super function which coincides with the super function return points. And finally back at line 117 in `buyTokens`, we see three return points. One at 325 gas which corresponds to the first `require` failing in function `isOpen`. The second is only slightly over 200 gas more expensive, and indicates

5. CASE STUDY

that it aborted in `isOpen` after checking the closing time. The third matches in number of occurrences and cost the `require` statement of `_preValidatePurchase` in the `Crowdsale` contract, which checks whether `_weiAmount` is zero.

- The call to `_getTokenAmount` follows in line 120. The function notably executes a storage load to get the `rate_` which together with the multiplication in `SafeMath` leads to a return point cost of 398 gas.
- For line 123, we see a very high cost due to the storage write and loads. `weiRaised_` has to be read first before `weiAmount` can be added and written back to.
- Looking further in less detail, we see that the logging costs at the emit are considerable and the transfer in the `_forwardFunds` function is another very expensive part.

```
1 uint256 private _openingTime;
2 uint256 private _closingTime;
3 modifier onlyWhileOpen {
4     require(isOpen());
5     _;
6 }
7 /**
8  * @return true if the crowdsale is open, false otherwise.
9  */
10 function isOpen() public view returns (bool) {
11     // solium-disable-next-line security/no-block-members
12     return block.timestamp >= _openingTime && block.timestamp <= _closingTime;
13 }
14 /**
15  * @dev Extend parent behavior requiring to be within contributing period
16  * @param beneficiary Token purchaser
17  * @param weiAmount Amount of wei contributed
18  */
19 function _preValidatePurchase(
20     address beneficiary,
21     uint256 weiAmount
22 )
23 internal
24 onlyWhileOpen
25 {
26     super._preValidatePurchase(beneficiary, weiAmount);
27 }
```

Listing 5.1: Extract from `TimedCrowdsale.sol`

```
1 /**
2  * @dev Validation of an incoming purchase. Use require statements to revert
3  * state when conditions are not met. Use 'super' in contracts that inherit
4  * from Crowdsale to extend their validations.
5  * @param _beneficiary Address performing the token purchase
6  * @param _weiAmount Value in wei involved in the purchase
7  */
8 function _preValidatePurchase(
9     address _beneficiary,
10     uint256 _weiAmount
11 )
12 internal
13 {
14     require(_beneficiary != address(0));
15     require(_weiAmount != 0);
16 }
```

```

14 | }
15 | /**
16 | * @dev Override to extend the way in which ether is converted to tokens.
17 | * @param _weiAmount Value in wei to be converted into tokens
18 | * @return Number of tokens that can be purchased with the specified _weiAmount
19 | */
20 | function _getTokenAmount(uint256 _weiAmount)
21 |     internal view returns (uint256)
22 | {
23 |     return _weiAmount.mul(rate_);
24 | }

```

Listing 5.2: Extract from Crowdsale.sol

5.3 Performance

We look at the run times when submitting the TimedCrowdsale project to Visualgas via Git on a i7-3520M 2.9GHz 2 core processor, 8GB of memory and an Intel SSD 520 Series 180GB in Linux Kubuntu 16.04. Table 5.1 shows a comparison of run times with a Fuzzer limit of 5,000 and 10,000 transactions. The Fuzzer returns a coverage of 99% for the deployed SimpleToken contract and 96% for the MyTimedCrowdsale contract in both cases. Note that the Fuzzer returns only the coverage of these contracts as these are the only ones deployed besides the Migration contract. Thus, overwritten functions from parent contracts might not be counted towards these numbers. From the coverage it follows that the difference in data is small between both cases and we can not necessarily expect much benefit from having more than 5,000 transactions in this case. However, for Truffle projects with a larger deployment a larger number of transactions can be required for a high coverage. As the Fuzzer reports statement coverage, it does not generally reflect path coverage or all possible gas costs.

Investigating the table, we see that writing the transactions to the disk in the Fuzzer and reading it during the gas analysis is what takes up a large part of the time and would have to be solved better for a production environment. The Fuzzer extract is slowed down by a large part due to the npm installation of solc, which we could solve differently as well. We could set up multiple Truffle installations with a different solc version each and then use the right one flexibly to circumvent the need for installing solc on demand.

	5001 tx	10089 tx
Websserver preprocessing	2s	2s
Fuzzer extract	20s	20s
Solc compile for AST	<1s	<1s
Fuzzing	8s	22s
Gas Analysis Contract parsing	3s	3s
Transaction reading	6s	24s
Transaction parsing	3s	6s
Return point analysis and various	3s	5s
Gas Analysis total	15s	38s
Total	45s	82s

Table 5.1: Visualgas run times for TimedCrowdsale

Conclusions

We investigate the gas costs of Ethereum smart contracts. Given a Truffle project, we use a Fuzzer to collect traces for which we map the gas costs to the source code. We specifically resolve the traces to clarify where the gas costs originate and take costs, which the trace only implicitly tracks, into account. Using the source mapping, we also examine where the return points of all functions are for specific traces and what costs the function executions have at the specific return points. The tool as a whole, Visualgas, combines the whole process in a Docker container with a flask server to provide a simple web application.

The web application serves to take the Truffle project as input and visualizes the results to help developers investigate where high gas costs arise. We provide the manual analysis as a basis to take measures and optimize the smart contract code to save gas. In the case study in Chapter 5, we show how gas costs can be examined in detail for a real world problem. We also take a closer look at the return point gas costs to see how they vary in different control flow scenarios.

Visualgas lets developers examine gas costs in detail and find ways to save gas by providing valuable insights that other tools to our knowledge do not provide. With debugging specific executions of smart contracts as the alternative, we offer visual, aggregated costs linked to the source code instead of step by step information.

While the fundamental part of Visualgas stands, it can be developed to support much more features. Many of the gas saving measures shown in Chapter 2 could be implemented to show where gas can be saved and how. Some checks could be done with static analysis, as for example detecting the use of `byte[]`, while many others could complement the current dynamic analysis. Further modifications in the Fuzzer would allow for keeping track of the storage state before each transaction which would be of interest so that

6. CONCLUSIONS

all refunds can be detected where they occur.

Another interesting case would be to provide detailed information on what input a transaction receives and what state the contract has in certain situations. We might be interested to know what caused high costs such that we can reproduce it in a debugger. This could of course be taken further to actually add stepping through the trace in Visualgas.

Additionally, more work can be done to improve the front-end and how the data is visualized. For readability, it would be especially beneficial to have the histogram span across multiple lines in case that the statement is across multiple lines. The error handling for Visualgas could also be improved, including better input validation.

Bibliography

- [1] Codemirror javascript text editor implementation. <https://codemirror.net/>. Accessed: 2018-11-11.
- [2] Etherscan ethereum blockchain explorer. <https://etherscan.io/>. Accessed: 2018-11-11.
- [3] Flask python microframework. <http://flask.pocoo.org/>. Accessed: 2018-11-11.
- [4] Mocha javascript framework. <https://mochajs.org/>. Accessed: 2018-11-05.
- [5] Openzeppelin solidity github code repository. <https://github.com/OpenZeppelin/openzeppelin-solidity>. Accessed: 2018-11-13.
- [6] Timed crowdsale github code repository. <https://github.com/ABBDVD/TimedCrowdsale>. Accessed: 2018-11-13.
- [7] Truffle suite documentation. <https://truffleframework.com/docs>. Accessed: 2018-11-10.
- [8] @cgewecke et al. eth-gas-reporter. <https://github.com/cgewecke/eth-gas-reporter>. Accessed: 2018-11-05.
- [9] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 442–446, 2017.
- [10] Ethereum foundation. Ethereum flavored webassembly design specification. <https://github.com/ewasm/design>. Accessed: 2018-11-09.

BIBLIOGRAPHY

- [11] Ethereum foundation. Ethereum virtual machine wiki. [https://github.com/ethereum/wiki/wiki/Ethereum-Virtual-Machine-\(EVM\)-Awesome-List](https://github.com/ethereum/wiki/wiki/Ethereum-Virtual-Machine-(EVM)-Awesome-List). Accessed: 2018-11-09.
- [12] Ethereum foundation. Solidity documentation v0.4.25. <https://solidity.readthedocs.io/en/v0.4.25/>. Accessed: 2018-11-03.
- [13] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proc. ACM Program. Lang.*, 2(OOPSLA):116:1–116:27, October 2018.
- [14] Leonid Logvinov. Eth-denver profiler. <https://devpost.com/software/profiler-4mahxw>. Accessed: 2018-11-10.
- [15] Petar Tsankov, Andrei Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts, 2018.
- [16] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger (byzantium version 2d0661f - 2018-11-08), 2018. Accessed: 2018-11-10.



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Gas Cost Analysis for Ethereum Smart Contracts

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

Signer

First name(s):

Christopher

With my signature I confirm that

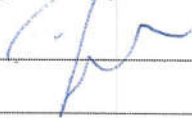
- I have committed none of the forms of plagiarism described in the '[Citation etiquette](#)' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zurich, November 14th, 2018

Signature(s)



For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.