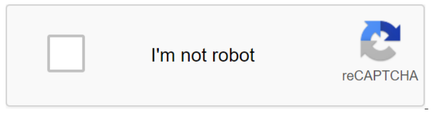
Mastering ethereum o'reilly pdf



Continue

Andreas is a passionate technologist who is well versed in many technical disciplines. He is a serial tech entrepreneur, starting a business in London, New York and California. He holds degrees in computer science and data and distributed systems from UCL. With experiences ranging from hardware and electronics to high-level business and financial systems to technology consulting and years as a CTO/CIO/CSO in many companies - it combines credibility and deep knowledge with the ability to make complex
subjects easy to understand. More than 200 of his articles on security, cloud computing and data centers have been published in print and syndicated around the world. His experience includes Bitcoin, cryptocurrencies, information security, cryptography, cloud computing, data centers, Linux, Open Source and software development for robotics. It has also been a CISSP certified for 12 years. As a bitcoin entrepreneur, Andreas founded three bitcoin businesses and launched several open source community
projects. He often writes articles and blog posts about bitcoin, is a regular host at Let's Talk Bitcoin and a prolific speaker at technology events. Andreas is a member of the advisory boards of several bitcoin startups and is the chief security officer of Blockchain. When Gavin first read about Bitcoin in 2011, he was largely uninterested, too much attention paid to the currency aspect rather than technology. However, when he returned to it in early 2013, he began to understand the new possibilities between the itC
and game theory areas, and the inevitable social changes that this would bring. A mutual friend made an introduction to Vitalik Buterin (Founder of Ethereum) in late 2014 and Ethereum has dominated Gavin Wood's life ever since. Gavin encodes the first functional implementation of Ethereum, developed the solidity contract language, and wrote Yellow Paper, the first official specification of any blockchain protocol and one of the key ways in which Ethereum differs from other blockchain systems. Prior to
Ethereum, Gavin earned a master's degree and a doctorate in computer science. He has advised Microsoft Research on the technical aspects of embedded domain languages and has participated in several startups in semantic extraction and real-time presentation for music, visual metaprogram systems, and automated legal text analysis. Title Mastering Ethereum: Building Smart Contracts and DApps Author (s) Andreas M. Antonopoulos, Gavin Wood Publisher: O'Reilly Media; 1 edition (December 23, 2018);
eBook (GitHub) License (s): CC BY-NC-ND 4.0 Paperback: 424 pages eBook HTML Language: English ISBN-10/ASIN: 1491971940 ISBN-13: 978-1491971949 Share This: Description The book is intended to work both as a reference book and as a cover to cover the Ethereum study. If you want to get started with protocol - or are among the many open source developers, integrators and system administrators already working with this platform - Mastering Ethereum is the final book on the topic. Ethereum is
the gateway to a global decentralized computing paradigm. This platform allows you to run decentralized applications (DApp) and smart contracts that have no central points of failure or control, integrate with the payment network and work on the open blockchain. With this hands-on guide, Andreas M. Antonopoulos and Gavin Wood provide everything you need to know about creating smart contracts and DApps on Ethereum and other virtual machine blockchains. Find out why IBM, Microsoft, NASDA and
hundreds of other organizations are experimenting with Ethereum. This important guide shows you how to develop the skills needed to be an innovator in this growing and exciting new industry. About the authors Andreas M. Antonopoulos is a critically acclaimed bestselling author, speaker and educator, and one of the world's leading Bitcoin and open blockchain experts. Andreas makes complex objects accessible and easy to understand. It is well known for conducting electrical negotiations that come out of the
abstract and the real world. Gavin Wood is the co-founder and former vice president of Ethereum and the inventor of the contract-oriented language solidity. He is also the founder and president of the Web3 Foundation, founder and vice president of Parity Technologies, and an advisor and founder of organizations such as Grid Singularity, Blockchain Capital, Polychain Capital and Melonport. Reviews, Ratings and Recommendations: Amazon Amazon (Mastering Bitcoin: Programming Open Blockchain) Related
Book Category: Read and Download Links: Related Books: Mastering Ethereum is a book for developers, offering a guide to working and using Ethereum, Ethereum Classic, RootStock (RSK) and other compatible EVM-based open blockchains. Reading this book To read this book, see book.asciidoc. Click on each of the chapters to read in your browser. Other parties may choose to release the book's PDF files online. Content Content status COMPLETE. The first edition of this book was published on
December 1, 2018. This edition is available in print and electronic format in many popular bookstores. It is flagged as first_edition_first_print in the development industry of this repository. Currently, only requests to correct errors are accepted. If you find an error, run a problem, or better yet, fix the problem with the pull request. We will begin work on the second edition at the end of 2019. The source and license of the first edition of this book, printed and sold by O'Reilly Media, is available in this repository.
Ethereum mastery released under Creative license CC-BY-SA. This Free Culture license has been approved by our publisher O'Reilly Media Media who understands the value of open source. O'Reilly Media is not only the world's best publisher of technical books, but also a strong supporter of this open culture and knowledge sharing. Mastering Ethereum Andreas M. Antonopoulos, Gavin Wood is licensed under creative Commons Attribution-ShareAlike 4.0 International License.Based on work in . If you are
interested in translating this book, please join our volunteer team on: Page 2 At the heart of the Ethereum protocol and operation is the Ethereum Virtual Machine, or EVM for short. As you might have guessed from the name, this computational engine is not very different from Microsoft 'NET Framework virtual machines, or translators of other integrated programming languages such as Java. In this chapter, we will describe EVM in detail, including a set of instructions, structure, and work, in the context of
Ethereum status updates. EVM is part of Ethereum, which deploys and executes smart contracts. Simple cost transfer transactions from one EOA to another should not include it, practically speaking, but everything else will include a state update calculated by EVM. At a high level, EvM, powered by the Ethereum blockchain, can be seen as a global decentralized computer containing millions of completed objects, each with its own permanent data store. THE EVM is a quasi-Turing-full state machine; quasi,
because all execution processes are limited to the finite number of computational steps by the amount of gas available for any given smart performance contract. Thus, the problem of stopping is solved (all program execution will be stopped) and a situation where the execution can (accidentally or maliciously) work forever, causing the Ethereum platform to stop in full, avoid. THE EVM has a stack-based architecture that stores all memory values in the stack. It works with a word measuring 256 bits (mostly to
facilitate native hashing and elliptical operations curves) and has several address data components: the immutable ROM code, downloaded using the smart contract by-agent code, which will run unstable memory, with each location explicitly initiated to zero Permanent Storage, which is part of the state of Ethereum, also with zero initials There is also a set of variables and data. We'll go through them in more detail later in this chapter. Figure 1. Ethereum Virtual Machine (EVM) Architecture and Execution
Context Term Virtual Machine is often applied to the virtualization of a real computer, usually a hypervisor such as VirtualBox or SEMU, or the entire operating room instance, such as Linux KVM. They have to provide programmatic abstraction, abstraction, actual equipment, system calls, and other kernel functions. EVM works in a much more limited area: it's just a computational engine, and as such provides abstraction only of computing and storage, like the Java Virtual Machine (JVM) specification, for
example. From a high-level perspective, JVM is designed to provide a performance time environment that is agnostic of basic OS or host equipment, allowing compatibility between a wide range of systems. High-level programming languages such as Java or Scala (which use JVM) or C --(which uses .NET) are compiled into a set of instructions on the You due codes of the corresponding virtual machine. In the same way, EVM performs its own set of code instructions (described in the next section) that compile
higher-level programming languages such as LLL, Serpent, Mutan, or Solidity. EVM thus has no planning capability because the execution order is organized from outside to it-Ethereum customers go through the verified transaction of the unit to determine which smart contracts need to be executed and in what order. In this sense, the world's Ethereum computer is as one-stranded as JavaScript. EvM also has no system processing interface or hardware support - no physical machine to interact with. Computer
World Ethereum is completely virtual. The EVM set of instructions offers most of the operations you might expect, including: Arithmetic and bitumen logical operations Context execution of Stack queries, memory, and access management operations to Logging flow storage, call, and other operators In addition to typical bycode operations, EVM also has access to account information (e.g. address and balance) and a block of information (such as block number and current gas price). Let's start our EVM study in
more detail by looking at available opcodes and what they do. As you would expect, all operands are taken from the stack, and the result (where applicable) is often put back at the top of the stack. The full list of opcodes and their corresponding gas cost can be found in the evm_opcodes. Available opcodes can be divided into the following categories: Arithmetic operations Arithmetic instructions on the rivet: ADD /Add two top stacks of MUL elements /Multiply the top two stacks of SUB elements /Subtract the two
top stacks of DIV elements /Integer division SDIV/Signature integrator division MOD (Modulo (residue) MULMOD /Multiplying modulo any number EXP /Exponential operation SIGNEXTEND /Extend the length of the two add-ons signed integer SHA3 (Compute Keccak-256 memory block hash Note that all arithmetic is performed modulo 2256 (unless otherwise specified) and that zero power is zero, 00, accepted to be 1. Operation Memory and Storage Management Instructions: POP/Remove the top item from
the MLOAD stack /Load the word from the memory of MSTORE /Save the word in memory MSTORE8 /Save the SLOAD memory byte /Load the word from the SSTORE store /Save the word for storage MS-WINNER /Get the size of active memory in PUSHx/Place x can be any integer from where x can be any integer from ! 1 to 16 inclusive SWAPx/Exchange 1st and (x'1)-th stack items, where x can be any // integer from 1 to 16 inclusive process flow operations Instructions for flow management: STOP //Halt
execution JUMP //Set the program contrary to any value JUMPI //Conditional change the oncoming PC program //Get the program counter value (before increment //compliant with this instruction) JUMPDEST //Mark a valid destination for System jumps System : LOGx/Append journal entry with x themes where x any integer ///from 0 to 4 inclusive CREATE //Create a new account with the associated CALL code //Message-call to another account, i.e. run another //CALL account code //Message-call to this
account with another //return account RETURN //Halt execution and return of output DELEGATECALL // Message-call to this account //Message-call in this account //Message-call in this account //Message-call in this account // but by retaining current values for /sender and staticCALL values/Static message-call to REVERT account/Halt execution, return of status changes, but return of /data and remaining INVALID gas /Designated invalid instruction SELFDESTRUCT /Halt execution and account registration to
delete logical Opcodes operations for comparison and bit logic: LT/less-than-comparison than the equalizer comparison /Comparison of equality IS'O /Simple NOT operator AND /Bitwise AND operation or (Bitwise OR operation XOR/Bitwise XOR operation NOT (Bitwise NOT operation BYTE /Retrieve one side by side from the full 256-bit word Environmental Operations Opcodes, Regarding Compliance Information: GAS //Get the amount of available gas (after the reduction //this instruction) ADDRESS //Get the
address currently fulfilling the BALANCE account //Get the account balance of any given ACCOUNT ORIGIN //Get the address of any given account ORIGIN //Get the address of this EVM // The execution of the caller // Get the address of the caller immediately responsible //Get the address of the current account BALANCE /Get the account balance of any given account ORIGIN //Get the account address of any given account ORIGIN //Get the address of the caller sent by the subscriber responsible for (this is
the execution of CALLDATAS-E)/Get the size of the input callDATACOPY /Copy input in memory CODES'E /Get the size of the code, in the current codecopY environment/Copy/Copy works in the current environment up to/memory GASPRICE /Get the gas price indicated by what is happening /transaction EXTCODESIZE /Get the code size of any EXTCODECOPY account /Copy code of any RETURNDATAS-E memory account /Get the output size from the previous call /in the current returndCOPATAY /Copy
data output from the previous call in the memory unitCodes to access information about the current block: BLOCKHASH /Get hash of one of the 256 recently completed /COINBASE blocks /Get the address of the block beneficiary of the reward block TIMESTAMP /Get the block timestamp NUMBER /Get the block number DIFFICULTY /Get the difficulties of the block GASLIMIT /Get the gas limit block Work EVM is a status update as defined by the Ethereum protocol. This aspect leads to The description of
Ethereum as a transaction-based government machine, reflecting the fact that external entities (i.e. account holders and miners) initiate state transitions by creating, accepting and ordering transactions. At this point, it's helpful to consider what Ethereum is. At the highest level we have a world state Of Ethereum. A world state is the display of Ethereum addresses (160-bit values) to accounts. At a lower level, each Ethereum address is an account consisting of an ether account (stored as the amount of wei owned
by the account), nonce (representing the number of transactions successfully sent from that account, if it is EOA, or the number of contracts it has created if it is a contract account), account storage (which is a permanent data store used only by smart contracts), and account code (again, only if the account is a smart account). EOA will never have code or empty storage. When a transaction results in a smart contract code being executed, the EVM instantly tosses with all the information needed in connection
with the current block being created and the specific transaction being processed. In particular, the EVM ROM code is loaded with the code of the so-called contract account, the program counter is set to zero, the storage is loaded from the contract account store, the memory is set up for all zeros, and all variable blocks and environments are installed. The key variable is the gas supply for this execution, which is set on the volume of gas paid by the sender at the beginning of the transaction (see Gas for more
details). As the code is implemented, the gas supply is reduced depending on the cost of the gas of the operations performed. If at some point the gas supply is reduced to zero, we get an exception from gas (OOG); execution immediately stops and the transaction is abandoned. No changes to Ethereum state except that the sender's heavy increment is incrementally, and the ether balance falls to pay the block beneficiary for the resources used to run the code to a stop point. At this point, you can think of an
EVM running on a sandbox copy of the state of the ethereum world, with this sandbox version discarded completely if the execution can't be completed for any reason. However, if the execution is indeed successful, then the actual state is updated in the matched sandbox version, including any changes in the storage data of the so-called contract, any new contracts created and any translations of the ether balance that have been initiated. Note that because a smart contract itself can effectively initiate
transactions, executing the code is a recursive process. The contract can trigger other contracts, with each call causing the other EVM to instantly infect around a new call goal. Each instantiation has its sandbox world status initiated from the EVM sandbox at the level above. Each moment is also given a certain amount of gas for its gas supply (not exceeding, of course, the amount of gas left above), and therefore can stop, except that it is given too little gas to complete its execution. Again, in such cases the
sandbox state is discarded, and the execution returns to EVM at a higher level. The compilation of the Solidity source file into the EVM integrid can be done in several ways. In intro_chapter we used the online compiler Remix. In this chapter, we'll use solc on the command line. For the list of options, run the following command: $solc --Help Raw Flow Generation opcode of the Solidity source source file is easily achieved with the command-line option --opcodes. This opcode thread leaves some information
(option-asm gives full information), but that's enough for this discussion. For example, compiling the example of the Solidity file, Example.sol, and sending the release of the opcode to a directory called BytecodeDir is done with the following command: $ solc -o BytecodeDir --opcodecodes Example.sol $solc-o BytecodeDir --asm Example.sol The next team will produce binary code for our approximate program: $solc-o BytecodeDir Our simple Solidity Example.sol file has only one contract, named by example:
the hardness of the pragmatism is 0.4.19 euros; An example of a contract is a contract-owner address; Example function () - contract owner and msg.sender; As you can see, all this contract does is hold one insistent state variable that is set as the address of the last account to run that contract. If you look at bytecodeDir, you'll see the opcode example.opcode file that contains EVM as a contract example. Discovery Discovery File example.opcode in the text editor will show THE following: PUSH1 0x60 PUSH1
0x40 MSTORE CALLVALUE IS'ZERO PUSH1 0xE JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST CALLER1 0x0 DUP1 PUSH2 0x100 EXP DUP2 S DUP2 PUSH20
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
0x60 PUSH10x40 MSTORE PUSH1 0x0 DUP1 REVERT STOP LOG1 PUSH6 0x627A7A7A723058 KECCAK256 JUMP 0xb9 SWAP14 0xcb 0x1e 0xdd RETURNDATACOPY 0xe0 0x1f 0x27 0xc9 PUSH50x9C5ABCC14A NUMBER 0x5e INVALID EXTCODESIZE 0xdb 0xcf EXTCODESE 0x27 EXTCODESE 0xe2 0xb8 SWAP10 0xed 0x Image-asmcode compiles a file called example.evm in our byte.evm catalog. This contains a slightly higher-level description of the instructions for EVM integration, as well as
some useful annotations: / Example.sol:26:132 Contract example :tag_1... iszero (callvalue)) 0x0 dup1 return tag_1: / Example.sol:115:125 msg.sender // caller / Example.sol:99:112 contractOwner No/ 0x0 dup Example1 / 99:99:4125 contractOwn and msg.sender s / 0x100 exp dup2 sload dup2
0xff/example.sol:26
Example of contract No... s/ dataSize (sub_0) dup1 dataOffset (sub_0) 0x0 codecopy 0x0 return stop sub_0: build // Example.sol:26:132 Example of mstore contract (0x40, 0x60) 0x0 dup1 return auxdata: 0xa165627a7a777230582056b99dcb1edd3eece01f27c9649c5abcc14a435efee 3b... --be-run option produces machine-readable hexagonal book-sageimal integral code: 60606040523415600e57600080fd5b333600008008061010 ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff908373
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff908373 ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff 16021790550606030305b... PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE Here we have PUSH1 followed by an unprocessed tote cost 0x60. This EVM instruction takes one side by side after the opcode in the program code (as a literal
value) and pushes it to the stack. You can push sizes up to 32 bytes per stack, as in: PUSH32 0x436f6e6e67726177756c6174696f6e736f6f6e20207446f206d617374657221 Second OPcode PUSH1 from example.opcode stores 0x40 on top of the stack (0x60 already there is one slot down). Next up is MSTORE, which is a memory store operation that retains value EVM. Two arguments are required and, like most EVM operations, gets them out of the stack. For each argument, the stack popped up; i.e. the top
value of the stack is washed away, and all other values in the stack are moved up one position. First First For MSTORE it is the address of the word in memory where the meaning to be saved will be delivered. For this program we have 0x40 at the top of the stack, so it is removed from the stack and used as a memory address. The second argument is the value that needs to be maintained, which is 0x60 here. After MSTORE, our stack is empty again, but we have a 0x60 (96 in decimal point) at 0x40. The next
opcode is CALLVALUE, which is an environmental opcode that pushes the amount of ether (measured in the wei) sent to the top of the stack, sent with the call message that initiated this execution. We could continue to step through this program this way until we had a complete understanding of the low-level state changes that this code effects, but that won't help us at this stage. We'll get back to it later in the chapter. There is an important but subtle difference between the code used to create and deploy a
new contract on the Ethereum platform and the contract code itself. To create a new contract, you need a special transaction that must be installed at a special 0x0 address and a data field configured for the contract initiation code. When processing such a contract-building transaction, the new account code of the contract is not the code in the transaction data field. Instead, EVM is instantly downloaded to its ROM software code using the code in the transaction data field, and then the release of that deployment
code is counted as code for the new contract account. This is to ensure that new contracts can be programmatically initiated using the global State of Ethereum at the time of deployment, setting values in contract storage and even sending ether or creating new contracts. When you're making a contract offline, for example, using solc on a command line, you can get a deployment code or a run time bytecode. The deployment code is used for every aspect of the initialization of a new contract account, including
the bytecode code that will actually run when the transactions of this new contract (i.e. bytecode run time) are called, and the code to initiate the entire contract constructor. On the other hand, the youth code of run time is exactly the code that is eventually executed when a new contract is called, and nothing more; it does not include the code needed to initiate the contract during deployment. Take the simple Faucet.sol contract, which we created earlier as an example: / A version of the Solidity compiler this
program was written for the hardness of the prague of 0.4.19 euros; Our first contract tap! Contract Faucet - / Give the airwaves to anyone who asks to remove the feature (uint withdraw_amount) public q ... The amount of lifting the limit requires (withdraw_amount qlt; Send the amount to an address that this is msg.sender.transfer (withdraw_amount); Accept any incoming amount function () external payable to get the deployment code, we run solc-bin Faucet.sol. If we instead wanted only time running bytecode,
we would run solc-ben-runtime Faucet.sol. If you compare the output of these commands, you can see that the run time code is a subset of the deployment code. In other words, the time code is fully contained in the deployment code. Disassembling an EVM code by the way is a great way to understand how high strength works in EVM. There are several developers you can use for this: Pig is a popular open source decompiler. Ethersplay is an EVM plug-in for Binary Ninja, a showdown. IDA-Evm is an EVM
plug-in for IDA, another developer. In this section we'll use the Ethersplay plugin for Binary Ninja and start disassembling the tap while running bytecode. After getting the time running bytecode Faucet.sol, we can feed it to Binary Ninja (after downloading the Ethersplay plugin) to see what the EVM instructions look like. Figure 2. Disassembling the crane time to run bytecode When you send a transaction to an ABI-compatible smart contract (which you can assume are all contracts), the transaction first interacts
with the manager of that smart contract. The dispatcher reads out the transaction data box and sends the relevant part to the appropriate function. We can see the example of the dispatcher at the beginning of our disassembled Faucet.sol running time bytecode. After the familiar MSTORE instructions, we see the following instructions: PUSH1 0x4 CALLDATASIZE LT PUSH1 0x3f JUMPI As we have seen, PUSH1 0x4 places 0x4 on top of the stack, which is otherwise empty. NALDATASSE receives the size of
the data bytes sent with the transaction (known as calldata) and pushes that number onto the stack. Once these operations have been completed, the stack looks like this: Stack of calldata from tx'gt; 0x4 The following LT instruction, shortening from less than. The LT instruction checks if the top item in the next item stack is smaller. In our case, it checks if the result of CALLDATASIZE is less than 4 bytes. Why does EVM verify that transaction call data is at least 4 bytes? Because of how feature identifiers work.
Each feature is identified by the first 4 Bytes of the Keccak-256 hash. By placing the function name and what arguments it takes into the hash function keccak256, we can deduce its function ID. In our case, we have: keccak256 (take off (uint256)) - 0x2e1a7d4d... Thus, the function ID for the withdrawal function (uint256) is 0x2e1a7d4d, as it is the first 4 bytes of the received hash. The function ID is always 4 bytes long, so if the entire data field is transaction, в договор, меньше </length> </length> 4
bytes, then there is no function with which the transaction could communicate, unless the return function is defined. Because we've implemented this callback feature in Faucet.sol, EVM switches to this feature when calldata is less than 4 bytes. LT pushes the top two values out of the stack and if the transaction data field is less than 4 bytes, presses 1 on it. Otherwise, he pushes 0. In our example, let's assume that the transaction data field sent to our contract was less than 4 bytes. The PUSH1 0x3f instruction
pushes the 0x3f tote onto the stack. After this instruction, the stack looks like the following JUMPI instruction, which means jumping if. It works like this: jumpi (label, cond) / Go to the label if Cond is true In our case, the label 0x3f, which is where our back function lives in our smart contract. Argument cond 1, which was the result of the LT instruction earlier. To put this whole sequence in words, the contract goes back to playback if the transaction data is less than 4 bytes. The 0x3f should only be a STOP
instruction, because although we announced the back-up feature, we kept it empty. As you can see in the JUMPI instruction leading to the return function, if we hadn't implemented the back-up feature, the contract would have thrown the exception instead. Figure 3. The JUMPI instruction results in a return function let's look at the central control unit. Assuming we got calldata, which was more than 4 bytes in length, the JUMPI instruction wouldn't jump on the back-back feature. Instead, the code will go to the
following instructions: PUSH1 0x0 CALLDATALOAD PUSH29 0x100000... SWAP1 DIV PUSH4 0xffffffff and DUP1 PUSH4 0x2e1a7d4d equalizer PUSH1 0x41 JUMPI PUSH1 0x0 pushes 0 on a stack that is now otherwise empty again. CALLDATALOAD takes as an argument the index within calldata, sent to a smart contract, and reads 32 bytes from this index as such: calldataload (p) /load 32 bytes calldata, starting with a byte position p Since 0 was an index transferred to him from the push1 0x0 team,
CALLDATALOAD reads 32 bytes calldata, starting with byte 0, and then pushes it at the top of the stack (after popping the original 0x0). After PUSH29 0x100000... Instruction, stack then: Stack 0x1000000 ... (29 bytes in length) zlt;32 bytes of calldata' starting at byte 0'gt; SWAP1 switches the top element on the stack with the i-element after it. In this case, it changes 0x100000... with call data. New stack: Stack of calldata starting at q byte 0'gt; 0x100000... (29 bytes in length) The next DIV instruction that works
is as follows: div (x, y) // integrative division x/y In this case, x 32 bytes calldata, starting with byte 0, and y 0x10000000 ... (29 bytes in total). Can you почему диспетчер делает разделение? </32></32> </32></32> Hint: we read 32 bytes from calldata earlier, starting with index 0. The first 4 bytes of this calldata is the function ID. 0x10000000... We pushed previously 29 bytes long, consisting of 1 at the beginning and then all 0s. Splitting our 32 bytes calldata on this value will leave us with
only the very top 4 bytes of our calldata boot, starting with the index 0. These 4 bytes - the first 4 bytes in the call, starting with index 0 - is the function identifier, and that's how EVM extracts that field. If this part is not clear to you, think about it this way: in the database 10, 123400 / 1000 and 1234. In base 16 it is no different. Instead of each seat being multiples of 10, it's a few 16. Just as the division of 103 (1000) in our smaller example retained only the top digits, dividing our 32-minute baseline 16 to 1629
doing the same. The result of the DIV (function ID) gets pushed onto the stack, and our stack is now: Stack C zlt;function identifier sent in data and PUSH4 0xffffffff and instructions are redundant, we can ignore them completely as the stack stays the same after they are made. The DUP1 instruction duplicates the first item in the stack, which is a function ID. The following instruction, PUSH4 0x2e1a7d4d, pushes a pre-checked id of the withdrawal function function (uint256) onto the stack. Stack now: Stack
0x2e1a7d4d, the next instruction pops out of the top two elements of the stack and compares them. This is where the dispatcher does his main job: whether the function ID sent to the transaction msg.data field matches the withdrawal field (uint256). If they are equal, the equalizer presses one on the stack, which will eventually be used to move to the withdrawal function. Otherwise, the equalizer presses 0 on the stack. Assuming that the transaction sent to our contract really started with the id of the removal
function (uint256), our stack has become: Stack 1 of the 0x41, which is the address at which the recall function (uint256) lives in the contract. After this instruction, the stack looks like this: The Stack 0x41 1 ID sent to msg.data by jumpi, again accepts the top two elements of the stack as arguments. In this case, we have a jumpi (0x41, 1) that tells EVM to perform a jump to the place of removal (uint256) function, and the code of this function can continue. As we have touched, to put it simply, turing's system or
programming language is complete if it can run any program. This feature, however, comes with a very important caveat: some programs take forever to run. Important этого является то, что мы не можем сказать, просто глядя на</function> </function> </function> </function> </function> whether it will take forever or not to accomplish. We have to actually go through with the running program and wait for it to finish to find out. Of course, if it takes forever to accomplish, we'll have to wait
forever to find out. This is called a stop problem and would be a huge problem for Ethereum if it wasn't solved. Because of the stop problem, the Computer World Of Ethereum is in danger of being asked to run a program that never stops. It can be accidental or malicious intent. We discussed that Ethereum acts as a single-dark machine, without any planner, and so if it's stuck in an endless loop, it will mean that it will become unusable. However, there is a solution with gas: if the execution is not over after the
preliminary maximum amount of calculations, the program is suspended by EVM. This makes EVM a quasi-turing-complete machine: it can run any program you feed into it, but only if the program completes within a certain amount of computation. This limit is not fixed in Ethereum - you can pay to increase it to the maximum (the so-called gas block limit) and everyone can agree to increase this maximum over time. However, at any given time, there is a limit in place, and trades that consume too much gas
during the run are suspended. In the following sections, we'll look at the gas and explore in detail how it works. Gas is an Ethereum division for measuring the computing and storage resources needed to perform actions on the Ethereum blockchain. Unlike Bitcoin, whose transaction fees only take into account the size of the transaction in kilobytes, Ethereum must take into account every computational step performed by transactions and the execution of the smart contract code. Each transaction performed by a
transaction or contract is worth a fixed amount of gas. Some examples from the Ethereum Yellow Paper: Adding two numbers worth 3 gas Calculation hashing Keccak-256 costs 30 gas and 6 gas for every 256 bits of data hashed By sending transactional costs of 21,000 gas is a critical component of Ethereum, and serves a dual role: as a buffer between the (volatile) price of Ethereum and reward miners for the work they do and as protection against service. To prevent random or malicious endless cycles or
other computational losses on the network, the initiator of each transaction is required to set a limit on the number of calculations they are willing to pay for. Thus, the gas system does not encourage attackers to send spam transactions, as they have to pay proportionately for the computing resources, bandwidth, and storage resources they consume. When EVM is needed to complete first of all, it is provided with a gas supply equal to the amount specified by the gas limit in the transaction. Every opcode that has
a cost of gas, and therefore the EVM gas supply is reduced as the EVM passes through the program. Before each operation, EVM checks whether there is enough gas to pay for the operation. If you don't have enough gas, the execution is suspended and the transaction is returned. If EVM successfully reaches the end of execution without running out of gas, the gas cost used is paid to the miner as transaction fees converted to the ether based on the price of gas specified in the transaction: the miner's fee and
the cost of gas, the price of the gas left in the gas supply, returned to the sender, converted back to the ether depending on the price of the gas specified in the transaction: the remaining gas limit - the cost of the gas is returned - the remaining gas price is returned . , the operation is immediately terminated, lifting the exception from the gas. The transaction returns and all changes in the state are rolled back. Despite the failed transaction, the sender will be charged for the transaction, as the miners have already
done the computing work up to this point and must be compensated for it. The relative gas costs of the various operations that EVM can perform have been carefully selected to best protect the Ethereum blockchain from attacks. A detailed gas cost table for the various EVM OPCOTs can be seen in evm_opcodes_table. More calculatedly intensive operations cost more gas. For example, PERFORMING a SHA3 function is 10 times more expensive (30 gas) than an ADD operation (3 gas). More importantly, some
operations, such as EXP, require additional payment depending on the size of the operation. There is also the cost of gas to use EVM memory and to store data in the storage of a contract on a chain. The importance of comparing the cost of gas with the real cost of resources was demonstrated in 2016, when an attacker discovered and exploited a cost discrepancy. The attack spawned transactions that were very computationally expensive and made the main Ethereum network almost shut down. This
discrepancy has been eliminated by hard fork (codenamed Tangerine Whistle), which tweaked the relative cost of gas. In executing the transaction, the sender determines the price of gas that he is willing to pay (on the air) for each unit of gas, allowing the market to determine the relationship between the price of the ether and the cost of computing (measured in gas): transaction fees - the total gas price used (on the air) When building a new block, Ethereum network miners can choose among the pending
deals, choosing those that offer to pay a higher price for gas. , a higher gas price will encourage miners to include your deal and its Faster. In practice, the sender of the transaction will set a gas limit that is higher or equal to the volume of gas that is expected to be used. If the gas limit is higher than the amount of gas consumed, the sender will receive an excess amount refund, as miners are compensated only for the actual work they do. It is important to clearly understand the difference between the cost of gas
and the price of gas. The price of gas is the amount of ether you are willing to pay for a unit of gas when you send a transaction to the Ethereum network. Although gas has a price tag, it cannot be owned or spent. Gas exists only inside the EVM as a calculation of how much computing work is performed. The sender is charged for the on-air transaction, which is then converted into gas to account for EVM, and then back to the airwaves as transaction fees paid to the miners. Ethereum encourages the removal of
used variables and storage accounts by returning some of the gas used in the execution of the contract. There are two operations in EVM with negative gas costs: Removal of the contract (SELFDESTRUCT) is worth the return of 24,000 gas. Changing the storage address from non-grain to zero (SSTORE-x 0) costs 15,000 gas. To avoid using the refund mechanism, the maximum return for the transaction is set at half of the total gas used (rounded). Limiting a gas block is the maximum amount of gas that can
be consumed by all transactions in a block, and limits the number of transactions that can fit in a block. For example, we have 5 transactions with gas limits of up to 30,000, 30,000, 40,000, 50,000 and 50,000. If the gas block limit is 180,000, then any four of these deals can fit in the block, while the fifth will have to wait for the future block. As mentioned earlier, miners decide which transactions to include in the block. Different miners are more likely to choose different combinations, mainly because they receive
transactions from the network in a different order. If the miner tries to turn on a transaction that requires more gas than the current block gas limit, the unit will be rejected by the network. Most Ethereum customers will stop you from issuing such a transaction by giving a warning over the line of the transaction exceeding the gas block limit. The gas block limit on the main Ethereum network is 8 million gas at the time of writing according to the , meaning that about 380 basic transactions (each consuming 21,000
gas) can fit into the unit. Network miners collectively decide to limit gas. to the customer Geth or Parity Ethereum. The Ethereum protocol has a built-in mechanism, mechanism, can vote on limiting the gas so that the power can be increased or reduced in subsequent blocks. Block miner may vote to adjust the block gas limit to 1/1,024 (0.0976%) in any direction. The result is a adjustable block size based on the needs of the network at the time. This mechanism is accompanied by a default production strategy,
where miners vote on the gas limit of at least 4.7 million gas, but which is aimed at 150% of the average total gas consumption per block (using the exponential moving average in 1024 blocks). In this chapter, we explored the Ethereum virtual machine, tracking the performance of various smart contracts and looking at how EVM performs bytecode. We also looked at gas, the EVM accounting mechanism, and saw how it solves the stop problem and protects Ethereum from denial-of-service attacks. Next, in the
consensus, we look at the mechanism used by Ethereum to achieve decentralized consensus. Consensus. mastering ethereum o'reilly pdf