Hellhound Red Paper

Abdelhamid Bakhta (MS Cryptography and Security, Blockchain Protocol Engineer),
Amira Bouguera (MS Applied Mathematics, MS Cryptography and Security),

Overview

HELLHOUND VIRTUAL MACHINE

Hellhound Virtual Machine (HHVM) is a bytecode virtual machine. Bytecode is a form of instruction set designed for efficient execution by a software interpreter. Unlike human readable source code, bytecodes are compact numeric codes, constants, and references (normally numeric addresses) that encode the result of compiler parsing and performing semantic analysis of things like type, scope, and nesting depths of program objects.

Unlike the Ethereum Virtual Machine (EVM), the HHVM is not Turing Complete. HHVM enable use cases where privacy is required. However, no censorship is expected on business domains served by HH computations. The HHVM byte code is generated by the user wanting to perform a computation on the HH Network. We expect multiple wrapper implementation in different languages to generate HHVM byte code. HH nodes are able to run HHVM byte code if they support the version specified in the HHVM header. If a node is selected and does not support the current HHVM byte code version, an update mechanism is triggered.

Further details will be given in the node selection section. There are no restriction about the HHVM version during the selection process. This will cause the HH nodes to update their code frequently. HHVM is a Register Based Virtual Machines. In the register based implementation of a virtual machine, the data structure where the operands are stored is based on the registers of the CPU. The instructions need to contain the addresses (the registers) of the operands. That is, the operands for the instructions are explicitly addressed in the instruction, unlike the stack based model where we had a stack pointer to point to the operand.

We also consider that a well-crafted register based virtual machine is faster both overall in execution time and in instruction dispatch than a typical stack-based virtual machine.

A typical instruction will be formed as follows :
• **instruction** : The identifier of the instruction.
• **operands** : The list of operands for this specific instruction.

An example for a simple addition instruction will be as follows :
ADD R1, R2, R3 ;
Add contents of R1 and R2, store result in R3
Unlike the stack based VM, we need to explicitly mention the addresses of the operands as R1, R2, and R3. The advantage here is that the overhead of pushing to and popping from a stack is non-existent, and instructions in a register based VM execute faster within the instruction dispatch loop. Another advantage of the register based model is that it allows for some optimizations that

cannot be done in the stack based approach. One such instance is when there are common sub expressions in the code, the register model can calculate it once and store the result in a register for future use when the sub expression comes up again, which reduces the cost of recalculating the expression. Let’s define the Virtual Machine structure. The HHVM is composed of the following elements:
• **register set:** A set of registers.
• **keystore**: A set of cryptographics keys
• **instruction dispatcher**: The component that decode and dispatch instructions.
• **heap**: The heap of the virtual machine.
• **heap pointer**: A pointer to the current heap index.
• **stack**: The stack of the virtual machine.
• **stack pointer**: A pointer to the current stack index.

A. Composition
1) Register Set:
The register set enables the storage of any data.
Basically, it is an array of byte-array. The size — number of slots is specified in the header of a computation payload (See. section HHVM Code for more information). This register set is dynamically created and filled with zeros at the start of the virtual machine. Data values are addressed by their slot number. The condition to store a value at a given slot number is the following, given S the desired slot number and MaxSlot the maximum number of slots for the current VM instance, 0 <= S < M axSlot
Register set is a general purpose storage component, any data could be stored into it, represented as byte array.
b) Load a value in the register set:
Load registry instruction.
LOAD REGISTRY OP1, OP2, OP3 ;
This is the definition of the instruction that load a given value into a slot of the register set.
The operands are defined as follows:
• OP1: The slot number. Must comply with intrinsic validity rules. See Intrinsic Validity section for more details.
• OP2: The size of the value. 2 bytes indicating the number of bytes to store in the register set.
• OP3: The value represented as a raw byte-array. No restriction about the format of the value.

2) Keystore:
The keystore enables the storage of cryptographic keys. It is worth mentioning that no private keys should be stored in the keystore. Only public keys used for homomorphic encryption should be stored. The size — number of slots is specified in the header of a computation payload (See. section HHVM Code for more information). This keystore is dynamically created and filled with zeros at the start of the virtual machine.

a) Keystore interface:

A cryptographic key is represented as follows:
• key type: A byte representing the kind of the cryptographic key (Paillier public key, RSA public key,...).
• key usage: A byte representing the usage of the key. Optional.

• key value: The value stored as a raw byte-array. Formatting of the key may vary according to the cryptosystem. TLV format is recommended in most of the cases.

Some verifications are performed before the execution of the HHVM code concerning the proper usage of the keystore. For example, the virtual machine check the key type and try to decode the value according to the associated format.
Let define a concrete example with Paillier cryptosystem.

b) Paillier public key generation:
• Choose two large prime numbers p and q randomly and independently of each other such that gcd(pq,(p−1)(q− 1)) = 1
• Compute n = pq and λ = lcm(p − 1, q − 1). lcm means Least Common Multiple.
• Select random integer g where g ∈ Z ∗ n2
• Ensure n divides the order of g • The public (encryption) key is (n, g) c)

Paillier public key formatting: code snipped to format a Paillier public key:

In the previous code section, we can see that a Paillier public key is composed of the following elements:
• N
• G
The resulting key value stored in the key stored is a TLV encoded byte-array, each element is identified by a specific tag value.

d) Load a cryptographic key in the keystore:
Load keystore instruction.
LOAD KEYSTORE OP1, OP2, OP3, OP4, OP5 ;
This is the definition of the instruction that load a given cryptographic key into a slot of the keystore.
The operands are defined as follows:
• **OP1**: The slot number. Must comply with intrinsic validity rules. See Intrinsic Validity section for more details.
• **OP2**: The key type. Defined as one byte.
• **OP3**: The key usage. Defined as one byte.
• **OP4**: The key size, the length of the key raw byte-array.
• **OP5**: The value represented as a raw byte-array. No restriction about the format of the value.

3) Instruction Dispatcher:
To process an instruction, the VM first figure out what kind of instruction the heap pointer refers to. The first byte of any instruction is the opcode. Given a numeric opcode, the dispatcher need to get to the right code that implements that instruction’s semantics. This process is called “decoding” or “dispatching” the instruction. The HHVM use indirect Call-threading dispatching technique. Each opcode is associated to a function pointer to access directly to the instruction code. The data structure used is a map where the key is the byte-opcode and the value is a function pointer with a specific signature.

a) Instruction function signature:

While the loaded representation used by the switch interpreter represents the opcode of each virtual instruction as a token, indirect call threading represents each virtual opcode as the address of the function that implements it. Thus, by treating the heap pointer as a function pointer key, callthreaded interpreter can execute each instruction in turn.

4) Heap:
The heap is created at HHVM start-up and shared by all concurrent processes on the machine. The HHVM code is loaded in the heap before the execution.

5) Stack:
The call stack is a stack data structure that stores information about the active subroutines of the HHVM code. The primary purpose of a call stack is to store the return addresses. When a subroutine is called, the location (address) of the instruction at which the calling routine can later resume needs to be saved somewhere. Using a stack to save the return address has important advantages over alternative calling conventions. One is that each task can have its own stack, and thus the subroutine can be thread-safe, that is, can be active simultaneously for different tasks doing different things. Another benefit is that by providing reentrancy, recursion is automatically supported. When a function calls itself recursively, a return address needs to be stored for each activation of the function so that it can later be used to return from the function activation.

B. HHVM Code

The bytecode that the HHVM can natively execute. HHVM Assembly is the human readable version of HHVM Code. The HHVM code is encoded in base64 and included in the HHComputation payload to be transported across the Hellhound network. The HHVM code can be generated by any developer in any language. Some templates of computations will be provided for many business use cases. We truly believe that several areas-industries will be conducive to the creation of generic computing modules. This kind of computation are candidates, if well crafted, to become general purpose products for business areas such as medical industry, electronic voting. It is worth mentioning that some computations require interactions with multiple Hellhound nodes depending on the cryptosystem selected for the computation. In particular, Secure Multi Party Computation cryptosystems involve the participation of multiple nodes.

1) Structure:
The HHVM byte code is encapsulated in a JSON string called HHComputation.
Here is the structure of this HHComputation payload :
• header: metadata related to the generated byte code
• code: base64 encoded HHVM byte code.
• footer: data to ensure integrity of the message.

2) Header:
The fields contained in the header are as follows:
• requestorId: The public address of the entity requesting the computation.
• cerberusId: The unique identifier of the Cerberus client that interact with the user.
• authorId: The public address of the entity that generated the byte code (can be same as requestorId).
• delegationReceipt: A proof that the requestor accepted the delegation of the code generation to another entity. Only present if authorId is different from requestorId.
• codeGenerationTimestamp: This is a record of Unix’s time at this code generation.
• version: HHVM version string to identify the minimal compatible version to run the code.
• registerSetSize: This is the size of the register set.
• keystoreSize: This is the size of the keystore.
• sla: The Service Level Agreement which is a commitment between a service provider (Hellhound node) and a client (user).

a) Service Level Agreement:
The SLA encapsulate particular aspects of the service – quality, availability, responsibilities. The Hellhound node selected for the computation must respect the terms of the agreement. During the Purgatory, mandatory phase to become a HH node, the HH node define the SLA it will follow. The rewards are proportional to the SLA level of quality. Some penalties are applied in case of non-respect of the agreement. In this case both the reputation and the stake of the HH nodes can be affected.

3) Footer:
The fields contained in the footer are as follows:
• integrityHash: This is the Keccak-256 hash of the HHVM code.
• signature: This is the signature of the integrityHash using authorId key.

The main objective of the footer is to ensure the computation code has been generated by the right author and not being altered.

4) Example:

C. Execution Overview
The fundamental task of a compute machine is to carry out an operation. An operation is performed with the execution of a set of instructions.
A basic instruction goes through the following cycle of events:
• Instruction fetch, where an instruction is fetched from memory.
• Decoding, which determines the type of the instruction — the opcode, or the operation it is supposed to perform. Additionally, decoding may also involve fetching operand(s) from memory.
• Execute. The decoded instruction is then executed; the operation is performed on the operands.
• Storing The result in the mentioned register(s).

Prior to the execution of instructions, there two additional steps:
• Initialization: Prepare the VM instance for the computation.
• Intrinsic Validity: Check code through different analysis.

1) Initialization:
The initialization phase if the first step of a Hellhound computation on the HHVM. First, the VM instance is created inside the host machine. The creation of the VM instantiate the core components of the HHVM such as the register set, the keystore, the instruction dispatcher. Some arguments present in the header define characteristics of the VM such as the number of keys allowed to be stored in the keystore and the number of registers. There are no restrictions on those arguments because they are part of the agreement that has been accepted or not by the selected node(s). The rules are known in advance, a user could request a million registers for a computation. Only nodes that accept this number of registers for the VM creation are candidates to be selected for this specific computation.

2) Intrinsic Validity:
This step ensure that the computation is well-formed and comply with Hellhound rules.
The criteria for intrinsic validity are as follows:
• The computation payload is well-formed.
• The integrity of the message was successfully verified.
• The signature is valid.
• The HHVM code is valid.

• The HHVM code match the SLA requirements.

3) Instruction fetch:
The instruction fetch step extract from heap the next instruction to execute.
4) Decoding:
The decoding step consist to decode the opcode, determine if it is a valid opcode and then dispatch to the corresponding piece of code.
5) Execute:
The execution of the instruction modify the VM state from S to S’.
The state S is defined by the hash of the following elements:
• register set hash: This is the Keccak-256 hash of all values of the register set.
• keystore hash: This is the Keccak-256 hash of all keys in the keystore.
• intermediary steps hash: This is the Keccak-256 hash of the intermediary steps of the computation.
6) Storing:
Once executed, the output(s) of each instructions are stored in the specified registers. Each instruction associated code represent an intermediary step for the computation. Therefore we must ensure the right execution of the code. We propose to use a dedicated smart contract to store the result of the successive state transitions of the VM during the computation. This can be defined as the sum of all states. Let n be the number of steps of the computation, the output to store on the blockchain is defined as follows:

Xni=1 S i

The nodes responsible of computations perform smart contract calls during the execution. However, this interaction with the blockchain network is realized asynchronously in order to continue the computation. One key problem of the Ethereum ecosystem is the scalability. Therefore Hellhound off-chain computations should not suffer from blockchain inherent latency. As we want to provide a user centric solution, we deliver the service outcome first even if the proof of computations are not ready. The user can then wait until all proofs are ready and start the verification process that is defined in the next section.

D. Verification

III. PURGATORY

Only nodes who have successfully completed Purgatory (vetting process) can register to become a HellHound node . Nodes which want to join the network as dedicated computing resources will apply on-chain to the STYX contract. This application mechanism allows Hellhound to have a permanent transparent record of the participating nodes.
The steps of the purgatory process are defined as follows:
• Requirements: Check about computing power, resources, legal commitment, SLA.
• Private key storage: Ensure that the private key of each node must be protected and stay safe.
• Identity verification: The STYX contract will verify the identity of each node on the public blockchain.
• Staking: Put an amount of HellHound tokens at stake.

IV. INCENTIVIZATION
A. Token Mechanism
We introduce with Hellhound a token. The symbol of this token is HH.

B. Rewards

In order to build an efficient distributed system, the Hellhound network must be composed of an important number of working nodes. The participating nodes are rewarded according to the computation power and the SLA they provide.

C. Staking

It is important that the nodes respect the Hellhound rules and deliver correct results to users. To ensure the integrity of the participating nodes, we introduce a staking mechanism. The system can take a portion of HH tokens if some nodes are delivering wrong results.

D. Reputation

Additionally to the two previous incentivization mechanisms, there is also a reputation system. The reputation system is represented by a global reputation matrix where each node is associated to its own reputation score. Another layer of reputation is used and is denoted as the proximity reputation.

V. NODE SELECTION

A. Process

The node selection is divided in 3 steps :

- Eligibility selection
- Load rate selection
- Random selection

Eligibility selection

This step aim to select the nodes that are eligible to perform a specific computation based on criteria such as the kind of computation, the SLA.

Load rate selection

This step aim to select the nodes that are not too busy to perform a computation.

Random selection

To avoid any collusion and to build a faire system, the last phasis is a random selection.
We also want to let a chance to new participating nodes to be selected.

VI. STYX SMART CONTRACT SUITE

APPENDIX

OPCODES

0x10’s: Bitwise and logical operations

Data	Opcode	Input	Output	Cost	Description
0x00	STOP	0	0		

0x20’s: General purpose

Data	Opcode	Input	Output	Cost	Description
0x20	LOADKEY	5	0		Load a key in the keystore.
0x21	LOADREG	3	1		Load a value in the register set.

0x30’s : Paillier

Data	Opcode	Input	Output	Cost	Description
0x30	PAILLIERADDCIPHERS	3	1		Add 2 ciphers using Paillier cryptosystem.
0x31	PAILLIERADDCONSTAN T	3	1		Add a cipher and a constant using Paillier cryptosystem.
0x32	PAILLIERMULCONSTAN T	3	1		Multiply a cipher by a constant using Paillier cryptosystem.

REFERENCES

[1] Ruijie Fang and Siqi Liu, A Performance Survey on Stack-based and Register-based Virtual Machines, 2016
[2] DR. GAVIN WOOD, Ethereum Yellow Paper: a formal specification of Ethereum, a programmable blockchain, 2018