Hacash Virtual Machine Technical Design

Hacash.com
info@hacash.com

May 4, 2025

Abstract. The Hacash whitepaper [1] outlines a tri-currency system comprising HAC, HACD, and one-way transferred BTC, coupled with the Layer 2 technology of the Channel Chain Payment Network, forming the core foundation of a new crypto financial system. Hacash.com believes that Hacash's ultimate vision is to establish an open, trustless financial network centered around crypto sound money. In 2022, the Hacash.com team pioneered Layer 3 multi-chain scaling technology [2] based on Hacash to further refine the technical architecture and accelerate ecosystem development.

Following phased infrastructure development and scaling technology validation, and inspired by HIP-16 [3] and Hacash community discussions on Hacash's programmability, the Hacash.com team proposes upgrading Hacash Layer 1 with Turing-complete programmable technology, the Hacash Virtual Machine (HVM). This upgrade aims to fully realize Hacash's vision of sound money and open, trustless finance.

In proposing HVM, the Hacash.com team also specified its design priorities and principles. HVM will prioritize security, space optimization, and financial adaptation, and will address the challenges inherent in smart contract blockchains, such as protocol complexity, code security risks, infrastructure development difficulty, and the state explosion problem. The Hacash.com team intends to pave the way for a robust and scalable crypto financial ecosystem by adhering to a set of core technical design principles.

1 Introduction

The HVM upgrade will materialize several key features outlined in the Hacash whitepaper and deliver substantial benefits including but not limited to:

1. The L2 payment network [4] implementation will transcend mainnet hardcoding limitations of state channels, accommodating diverse third-party technical solutions to significantly enhance application flexibility and functionality, such as enabling programmable PayFi services within payment channels.
2. Direct implementation of versatile asset management models (multi-signature address, equity hierarchical control account, etc.) through contract code, achieving full "account abstraction": accounts as contracts, contracts as accounts.
3. Robust data and interface support for L3 and AppChain's multi-layer, multi-chain scaling ecosystem, establishing the foundation for exploring advanced application-layer scaling solutions and accelerating Hacash DeFi ecosystem growth.
4. Enabling sophisticated asset contracts and DeFi services (Swap, DEX, controlled payment) on L1, while enhancing programmability of the core tri-currency system, particularly for powerful BTCFi scenarios.
5. Technical infrastructure for highly customizable economic models of HIP-20 [5] compliant mainnet primary assets, with parallel support for fully customizable secondary asset issuance within contracts.
6. Facilitating development and experimentation with various stablecoin payment technologies and economic models.

Despite these advantages, HVM implementation may introduce inherent challenges of smart contract blockchains: increased full-node protocol complexity, contract code security risks, elevated infrastructure development difficulty, and state explosion problem weakening decentralization. To mitigate these impacts, HVM's technical design adheres to these core principles:

1. Prohibit all unsafe opcode behaviors (arithmetic overflow, memory segmentation faults, etc.) to minimize contract vulnerability risks.
2. Ensure each opcode operates simply, unambiguously and deterministically, avoiding complexity or uncertainty.

3. Enhance contract code accessibility and auditability through IR and other methods of support, thereby strengthening security.
4. Implement reasonable resource limits across all dimensions to support embedded device operation while maintaining upgrade flexibility.
5. Optimize bytecode compression for identical functions to reduce full-node storage overhead.
6. Enable efficient on-chain code reuse via library contract, inheritance contract, etc. to minimize redundant deployments.
7. Calls between contracts should be static and hierarchical to reduce code vulnerabilities and execution overreach risks.
8. Only fully executed and successful transactions are recorded on-chain to ensure HVM upgrade forward compatibility.
9. Adopt "time-bound leasing" for state storage to prevent permanent space occupation by transient data, effectively solving the state explosion problem.

The following sections detail HVM's implementation (note: certain technical parameters or mechanisms may be adjusted in the future release).

2 Contract Address

Currently, the Hacash address is version number 0 [6], appearing with leading '1' after Base58Check encoding. The HVM contract address is assigned version number 1, formatted to begin with Q-Z, a-k, or m-o (e.g., VFE6Zu4Wwee1vjEkQLxgVbv3c6Ju9iTaa), maintaining consistent encoding methodology with all decoded addresses being 21 bytes.

To enable deterministic contract address generation supporting offline transaction construction and multi-chain deployment, the address generation process combines the primary transaction address with 4 random bytes, processes through SHA-256 and RIPMED-160 hashing to 20 bytes, and adds with the version number prefix before Base58Check encoding.

Notably, deterministic contract address technology introduces address poisoning attacks: malicious actors could perform sustained hash collision attacks to generate deceptive addresses partially matching target contracts. One solution is to combine block hash or chain ID to generate a completely different and longer new address through conversion calculation, with clients recovering original addresses via reverse conversion. This approach offers additional advantages:

since block hashes are unique across L3 chains, the secondary encryption enables explicit target chain specification through contract addresses.

3 Contract Codes

Unlike Ethereum's simple one-dimensional (byte list) storage of contract code, HVM's contract code uses structured (object) method for deployment and reading, with each contract consisting of four parts:

1. Contract attributes and tags
2. Inheritance and library list
3. System call table
4. User function table

This brings the following advantages:

1. Pre-parsing and high-performance calls
2. Multi-language and JIT support
3. Multi-layer programmability
4. Partial upgrades on demand
5. Account abstraction support
6. Code reuse through inheritance and libraries

Since contracts are parsed into function tables upon loading, function calls within contracts and between different contracts become extremely fast, eliminating the bytecode offset jump overhead in Ethereum contract calls within the virtual machine, especially when frequently accessed contracts can be cached to save overhead each time.

Each function's source code is stored separately, allowing different functions to use different bytecode types: AST, IR, or Bytecode, even supporting dedicated bytecode from other virtual machines, and calling different target virtual machines for execution. When repeatedly called as hot code, the VM can use JIT compilation to convert AST or IR to Bytecode for efficient execution and caching. This also provides the foundation for L3 extension chains' multi-language, multi-platform hybrid programming and multi-VM architecture.

Each different function in contract code can choose different code types like AST, IR or Bytecode for deployment, allowing users to emphasize or balance between security auditability, clarity/simplicity, and execution efficiency/flexibility. This multi-level programmability solves

issues like high barriers and insufficient security in traditional single contract mechanisms.

HVM's design allows for modular updates of contracts. By structuring contracts as a collection of functions, HVM enables granular upgrades. Only the necessary functions are replaced, reducing deployment costs and storage overhead. This function-level upgradability is crucial for implementing account abstraction, which allows for the dynamic modification of access control logic without altering the core state of the account's contract.

Contracts can specify inherited contract lists and library contract lists, using on-chain inter-contract calls to avoid deploying duplicate functional code in each contract, adopting 1-byte contract pointers for table lookup calls to avoid the storage overhead of specifying 21-byte addresses each time. This "on-chain inheritance" contract technology also achieves a good programming paradigm of separating data and code, for example only needing to deploy one Swap contract, then all trading pair data contracts inherit from this contract, thus avoiding deploying hundreds of duplicate source codes, and enabling one-time unified upgrades during feature iterations.

Contracts can also directly return false in upgrade authorization system calls to permanently disable the upgrade functionality for the contract.

4 Fee Model

Hacash's transaction fees adopt a "space occupation" charging method, which is calculated according to the size of the transaction. After introducing HVM, the Turing-complete programmability("halting problem") makes this simple transaction fee charging method no longer feasible.

We abstract the Gas fee for contract execution into the concept of space occupation, where 1Gas = 1Byte, and each opcode Gas fee will be calculated as several bytes, thus the total fee required for transaction execution can be calculated and the halting problem can be solved.

We conceptualize contract execution Gas fees similarly as space occupancy (1Gas=1Byte), calculating total costs by converting each opcode execution into byte equivalents, thereby solving the halting problem.

Transactions now include an Extra Gas field representing executable Gas multiples (of original fees) as the Gas Limit, while the original fee-to-size ratio becomes Gas Price.

Since contract code will permanently occupy large space in the state storage, following the principle of HIP-11 [7], transactions that deploy or upgrade contracts will burn 90% of the fee, with additional space occupation fees charged.

Considering the final stability and perpetual nature of Hacash's block rewards, the space occupation fees for contract storage and Gas fees for calling code execution will be burned by the protocol. One important reason for this is that the space and execution costs will be borne by all full nodes together, not just miners, and miners already have stable block reward income.

5 Calling and Parameter

Contract calls can specify a particular function of the target contract, with the following call types:

1. Call External
2. Call Location
3. Call Lib
4. Call Lib Static
5. Call Code

Among these, External calls modify the state data of the target contract (the called contract), Location and Code calls modify the state of the calling contract, Lib calls can only read but not modify the state of the calling contract, while Lib Static calls cannot read or modify any state and can only be used as pure algorithm functions. Due to contract upgradability, the permission control of all call types is verified in real-time during runtime.

For local or inheritance calls, function lookup always starts from the calling contract, even when executing code from called contracts - meaning functions defined in the local contract will completely override functions with the same name in inherited contracts, and the two contracts will be treated as a single contract from the calling logic perspective.

The mechanism of code copy calls is completely different from function calls in traditional programming languages - it copies the function code of the target contract to the stack space of the calling contract for execution, and is logically treated entirely as source code of the calling contract. This calling mechanism is highly flexible, with typical application scenarios including decoupling code and data in permission control scripts for account abstraction, bringing significant security improvements and eliminating substantial deployment overhead.

Parameter passing and function return values between functions within the same contract or across different contracts all use a single buffer approach - meaning there is no unified parameter ABI, and parameter parsing is left to custom handling within functions. This provides several benefits:

1. Simplifies call logic implementation
2. Allows flexibility in custom parameter parsing logic
3. Supports VM precompiled bytecode
4. Unifies with Hacash's core Action calling logic
5. Supports contract function upgradability (variable parameters)

HVM will support a series of buffer-related opcodes, providing powerful capabilities for byte array splitting, merging, type conversion, and local variable access.

6 Virtual Machine

HVM is a strongly-typed, safely-convertible stack-based virtual machine that also serves as a multi-level runtime environment with JIT compilation capabilities. Designed specifically for decentralized money and open finance, it prioritizes the following core objectives:

1. Security: The primary requirement for large-scale currency and financial operations is security. One of the main reasons hindering massive capital from entering the EVM contract system is the security of contract virtual machines, which is also the key problem that public chains like Aptos and Sui using Move language attempt to solve. HVM performs boundary and overflow checks during every call stack, type conversion, and opcode execution for numerical operations, minimizing potential vulnerabilities in contract code, although this may slightly impact runtime efficiency. In other words, due to the strict security requirements of currency and finance, HVM will always prioritize security over code execution efficiency.
2. Progressive encapsulation: Vitalik once discussed the encapsulation issue of blockchain virtual machines in the article [8]. HVM's design philosophy agrees with the "progressive trade-off" logic mentioned in the article - starting from minimal viable encapsulation, gradually iterating and encapsulating more general opcodes and external functions to optimize Gas efficiency and code size. On the other hand, HVM also starts with smaller limits for runtime resource usage like stack depth, call hierarchy, and maximum Gas

consumption, gradually increasing these parameter limits according to ecosystem project needs during development. This progressive enhancement mechanism brings benefits in terms of security, efficiency optimization, and backward compatibility, but requires more full-node upgrades on the mainnet.

3. Space optimization: HVM does not prioritize performance as its primary goal. In fact, we believe that in the monetary finance logic pioneered by Bitcoin and Hacash, the most negative factor for decentralized ledgers is not execution speed but space occupation. Computation is immediate, short-term, and optimizable, while space occupation is comprehensive, long-term, and difficult to resolve. In designing opcodes, runtime resources, calling mechanisms, and functional encapsulation, HVM follows an iteration direction that maximizes savings in final bytecode size.
4. Financial adaptation: The Hacash blockchain is envisioned to support financial services such as currency circulation, derivatives settlement, and programmable asset issuance. HVM primarily serves these financial scenarios, which differ significantly from the technical structure needed for broadly defined "decentralized applications." The focus here is much more on asset value persistence, contract auditability, and underlying technical security, rather than simply transaction processing performance.

Unlike MoveVM-based public chains that rely on upper-layer contract language paradigms, HVM prioritizes the security of its underlying virtual machine. Rather than attempting to solve all challenges with a single technical concept, HVM makes selective trade-offs in the impossible triangle of smart contract security, simplicity, and efficiency. This is achieved through a layered programmable model, consistent with Hacash's approach to resolving scalability issues.

7 Runtime Resources

The resource space of HVM virtual machine operation consists of the following 6 parts:

1. Operand Stack: Consistent with the concept of other stack-based virtual machines, but each stack item has type tags, and the type and width of stack items can be explicitly or implicitly converted. Available within functions, destroyed after call returns.
2. Local Variable Stack: The structure is similar to the operand stack, but can be explicitly accessed by index, with depth and type control like the operand stack.

3. Contiguous Byte Heap: A contiguous byte space that can be applied for use as needed, supporting mutual access with the operand stack by type, available within functions.
4. Contract Memory Variables: Hash Map structure, values have types, readable and writable within the current contract, readable only by other contracts. Persists during the transaction cycle, meaning these variables still exist even when the contract is re-entered within the same transaction. Space is released after the transaction ends.
5. Global Variables: Hash Map structure, values have types, readable and writable by all contracts during the transaction, mainly used for standardized cross-contract communication.
6. Persistent Time-Limited Storage: Hash Map structure, values have types, accessed by contract. After the transaction ends, values are written to the blockchain state space for persistent storage, but require payment of rent (consuming Gas) based on storage time, expired data will be deleted.

By designing corresponding code runtime resource spaces in various dimensions such as function calls, contract space, transaction duration and persistent state storage, compared with traditional and other blockchain virtual machines, it can support more abundant, flexible and secure decentralized financial applications. For example, with the contract memory variable space, re-entrancy vulnerabilities in contracts can be efficiently solved, and authorization and atomicity for cross-contract multiple calls like flash loans can be supported.

8 Account Abstraction

As an important part of account abstraction technology, customizable system functions can bring significant paradigm shifts in programming models, no less revolutionary than the impact of object-oriented thinking on programming languages.

In HVM contracts, core control logic such as contract upgrades, asset spending and receiving, authorization, and cancellation typically governed by Elliptic Curve Digital Signature Algorithm (ECDSA) accounts can be customized through code deployed within the contracts. Users can interact with these contract accounts much like traditional public-private key accounts, with the key difference being that permission checks are replaced by custom code instead of fixed ECDSA signatures. This customization grants the HVM contract accounts enhanced programmability and functionality, making them even more powerful.

This design that unifies external accounts (EOA) and contract accounts (CA) will bring leapfrog improvements to the smart contract user experience. For example, when adding liquidity into a Swap contract, there's no need to explicitly call the contract - you just need to transfer both assets of the trading pair to the contract address like making a regular transfer to a normal address. Similarly, when withdrawing liquidity, you only need to initiate a transfer from the contract to the user's account. Moreover, the receiving address can also be another contract address while simultaneously initiating a completely different DeFi operation. Externally, these operations all appear as several consecutive transfer operations within the same transaction.

This security, flexibility and convenience greatly enhances the composability of DeFi legos and the atomicity of financial operations. All of this is achieved through the customizable system extension function architecture supported by HVM.

9 Storage Rental

The space occupation of state data is a major factor affecting the decentralization degree of blockchain. When the ledger size exceeds the disk capacity of ordinary consumer-grade devices, mainnet verifiability sharply declines. One-time payments for permanent storage are fundamentally flawed, especially when most verification data only requires temporary retention. HVM divides persistent state storage fees into two parts and establishes an expired state recovery mechanism:

1. Slot rental: Persistent storage uses Hash Map method. Creating a new key-value pair will pay rent corresponding to the retention period after expiration. The basic fee is 64 Gas, retained for one period (100 Hacash blocks) after expiration. Retaining for 10,000 blocks after expiration costs 6,400 Gas, and so on.
2. Data rental: According to the size and duration of the stored value, corresponding rent will be charged. Every 100 Hacash blocks (Average 5 minutes a block, about 8 hours) is set as a time unit. For example, writing 200 bytes of state data will charge 200 Gas and store for 100 blocks, which can be renewed in integer multiples of the time unit.
3. Data recovery: When a storage key-value expires, nodes will compress the data into a 16-byte hash and retain the period set when the slot was created. During this period, users can resubmit the original state data to restore usage. After the recovery period, this state data will be completely cleared and unrecoverable.

The above design will greatly optimize the usage efficiency of blockchain state space, encourage developers to reasonably plan data validity periods, and reclaim space occupation when necessary. More importantly, it ensures the mainnet only stores valid and essential data.

10 IR

HVM, as a multi-layer, multi-language virtual machine, will natively support an intermediate language that preserves program logic structure as deployable and executable contract source code. After loading IR code, the virtual machine will use JIT technology to compile it into Bytecode for execution. The compilation results can be cached, reducing the compilation overhead of hot contracts to negligible levels.

Specifically, a newly designed scripting language can be directly converted into corresponding IR code, supporting recovery of program logic structure from IR code:

````
Unset
```
use AnySwap = VFE6Zu4Wwee1vjEkQLxgVbv3c6Ju9iTaa
lib ERC20 = VFE6Zu4Wwee1vjEkQLxgVbv3c6Ju9iTaa(1)

callcode ERC20::do_transfer

let abc = $0
let num = $7

abc = 8 as u32
num = 2 as u64
num = block_height()

AnySwap.do_swap(abc, 50)

ERC20:do_func1(abc)
ERC20::do_static_func(abc)

self.do_some_trs(num + 10)


abc = sha3(0xABC123)
abc = sha3("\"hacash\" \\\nworld")
````
````

```
abc = sha3(0x0000111100001111)
num = ripemd160(abc)

num = check_signature(VFE6Zu4Wwee1vjEkQLxgVbv3c6Ju9iTaa)

memory_put(abc, 24)
num = memory_get(abc)

if num > 100 {
    throw "panic error!"
}

while abc > 0 {
    abc = abc - 1
    num += 2
    if abc < 3 {
        AnySwap.do_swap(abc, 50)
        return memory_get(abc + 1)
    }
}
if num < 10 {
    assert abc >= 1
    num += 1
} else if num < 5 {
    abc *= 2
}else{
    return num + check_signature(num, abc*5, 2)
}
return (abc && 1) / (num - 1)

```

The above example of script language can be converted into an IR code with structure after compilation:

Unset
```
IRBLOCK 20 :
```

```
ALLOC 8
CALLCODE 0x011c33cd5f
PUTX 0
  CU32 PU8 8
PUTX 7
  CU64 PU8 2
PUTX 7 EXTENV 1
CALL 0x0135d4470300daabea474d082733333c1b694d8065e2a51fc7
  CAT GETX 0 PU8 50
CALLLIB 0x017da3371b GETX 0
CALLSTATIC 0x0108f30a6e GETX 0
CALLLOC 0xdd6a6c76
  ADD GETX 7 PU8 10
PUTX 0
  NTCALL 2 PBUF 0x02abc123
PUTX 0
  NTCALL 2 PBUF 0x0f2268616361736822205c0a776f726c64
PUTX 0
  NTCALL 2 PBUF 0x070000111100001111
PUTX 7
  NTCALL 3 GETX 0
PUTX 7
  EXTFUNC 1 PBUF 0x140135d4470300daabea474d082733333c1b694d8065
MPUT GETX 0 PU8 24
PUTX 7
  MGET GETX 0
IRIF
  GT GETX 7 PU8 100
  ERR PBUF 0x0b70616e6963206572726f7221 NOP
IRWHILE
  GT GETX 0 P0
  IRBLOCK 3 :
  PUTX 0
       SUB GETX 0 P1
  XOP 7 PU8 2
  IRIF
       LT GETX 0 PU8 3
       IRBLOCK 2 :
       CALL 0x0135d4470300daabea474d082733333c1b694d8065e2a51fc7
       CAT GETX 0 PU8 50
       RET
       MGET
       ADD GETX 0 P1 NOP
IRIF
```

```
  LT GETX 7 PU8 10
  IRBLOCK 2 :
  AST
       GE GETX 0 P1
  XOP 7 P1
  IRIF
       LT GETX 7 PU8 5
       XOP 0 PU8 2
       RET
       ADD GETX 7
       EXTFUNC 1
       CAT GETX 7
              CAT
              MUL GETX 0 PU8 5 PU8 2
RET
  DIV AND GETX 0 P1 SUB GETX 7 P1

IRBLOCK 20 :
ALLOC 8
CALLCODE 0x011c33cd5f
PUTX 0
       CU32 PU8 8
PUTX 7
       CU64 PU8 2
PUTX 7 EXTENV 1
CALL 0x0135d4470300daabea474d082733333c1b694d8065e2a51fc7
       CAT GETX 0 PU8 50
CALLLIB 0x017da3371b GETX 0
CALLSTATIC 0x0108f30a6e GETX 0
CALLLOC 0xdd6a6c76
       ADD GETX 7 PU8 10
PUTX 0
       NTCALL 2 PBUF 0x02abc123
PUTX 0
       NTCALL 2 PBUF 0x0f2268616361736822205c0a776f726c64
PUTX 0
       NTCALL 2 PBUF 0x070000111100001111
PUTX 7
       NTCALL 3 GETX 0
PUTX 7
       EXTFUNC 1 PBUF 0x140135d4470300daabea474d082733333c1b694d8065
MPUT GETX 0 PU8 24
PUTX 7
       MGET GETX 0
```

```
IRIF
       GT GETX 7 PU8 100
       ERR PBUF 0x0b70616e6963206572726f7221 NOP
IRWHILE
       GT GETX 0 P0
       IRBLOCK 3 :
       PUTX 0
       SUB GETX 0 P1
       XOP 7 PU8 2
       IRIF
       LT GETX 0 PU8 3
       IRBLOCK 2 :
       CALL 0x0135d4470300daabea474d082733333c1b694d8065e2a51fc7
              CAT GETX 0 PU8 50
       RET
              MGET
              ADD GETX 0 P1 NOP
IRIF
       LT GETX 7 PU8 10
       IRBLOCK 2 :
       AST
       GE GETX 0 P1
       XOP 7 P1
       IRIF
       LT GETX 7 PU8 5
       XOP 0 PU8 2
       RET
              ADD GETX 7
              EXTFUNC 1
                     CAT GETX 7
                            CAT
                            MUL GETX 0 PU8 5 PU8 2
RET
       DIV
       AND GETX 0 P1
       SUB GETX 7 P1

```

This IR code can be deployed as a contract function in binary format:

Unset

```
bf0824011c33cd5fbc00424808bc07434802bc070701200135d4470300daabea474d082733333c1
b694d8065e2a51fc758bb00483222017da3371bbb00230108f30a6ebb0021dd6a6c7680bb07480a
bc0027024e02abc123bc0027024e0f2268616361736822205c0a776f726c64bc0027024e0700001
11100001111bc072703bb00bc0706014e140135d4470300daabea474d082733333c1b694d8065cd
bb004818bc07ccbb00f17dbb074864ee4e0b70616e6963206572726f7221fef27dbb004af00003b
c0081bb004bb9074802f17cbb004803f00002200135d4470300daabea474d082733333c1b694d80
65e2a51fc758bb004832ebcc80bb004bfef17cbb07480af00002ed7fbb004bb9074bf17cbb07480
5b9004802eb80bb07060158bb075882bb0048054802eb8378bb004b81bb074b
```

Users or contract auditors can directly restore the program structure of IR code into a readable format:

Unset

```

local_alloc(8)
callcode <1>::<1c33cd5f>
$0 = 8 as u32
$7 = 2 as u64
$7 = block_height()
VFE6Zu4Wwee1vjEkQLxgVbv3c6Ju9iTaa.<e2a51fc7>($0 ++ 50)
<1>:<7da3371b>($0)
<1>::<08f30a6e>($0)
self.<dd6a6c76>($7 + 10)
$0 = sha3(0xabc123)
$0 = sha3("\"hacash\" \\\nworld")
$0 = sha3(0x0000111100001111)
$7 = ripemd160($0)
$7 = check_signature(VFE6Zu4Wwee1vjEkQLxgVbv3c6Ju9iTaa)
memory_put($0, 24)
$7 = memory_get($0)
if $7 > 100 {
       throw "panic error!"
}
while $0 > 0 {
       $0 = $0 - 1
       $7 += 2
       if $0 < 3 {
```

```
            VFE6Zu4Wwee1vjEkQLxgVbv3c6Ju9iTaa.<e2a51fc7>($0 ++ 50)
            return memory_get($0 + 1)
        }
}
if $7 < 10 {
        assert $0 >= 1
        $7 += 1
} else {
        if $7 < 5 {
            $0 += 2
        } else {
            return $7 + check_signature($7 ++ $0 * 5 ++ 2)
        }
}
return ($0 && 1) / ($7 - 1)

```

The restored readable code format maintains the same logical structure as the uncompiled source code, only losing variable names. Combined with public Source Map data, the complete source code can be fully restored, significantly improving contract reusability and auditability.

When calling contracts, after the contract's IR code is loaded, HVM compiles it into executable Bytecode and sends it to the virtual machine for execution:

```
Unset
```


```
block: [183, 198, 268, 227, 265, 198, 311, 304, 308, 319]
entry:
        local_alloc[8]
        callcode[1,28,51,205,95]
        8 · push_u8[8]
        cast_u32
        put_x[0]
        2 · push_u8[2]
```

```
cast_u64
put_x[7]
call_extend_env[ block_height() ]
put_x[7]
50 · push_u8[50]
get_x[0]
concat
call[ VFE6Zu4Wwee1vjEkQLxgVbv3c6Ju9iTaa.<e2a51fc7> ]
pop_stack
get_x[0]
calllib[1,125,163,55,27]
pop_stack
get_x[0]
callstatic[1,8,243,10,110]
pop_stack
10 · push_u8[10]
get_x[7]
add
callloc[221,106,108,118]
pop_stack
push_bytes[2,0xabc123]
native_call[2]
put_x[0]
push_bytes[15,0x2268616361736822205c0a776f726c64]
native_call[2]
put_x[0]
push_bytes[7,0x0000111100001111]
native_call[2]
put_x[0]
get_x[0]
native_call[3]
put_x[7]
push_bytes[20,0x0135d4470300daabea474d082733333c1b694d8065]
call_extend_func[ check_signature(..) ]
```

```
        put_x[7]
        24 · push_u8[24]
        get_x[0]
        memory_put
        get_x[0]
        memory_get
        put_x[7]
        100 · push_u8[100]
        get_x[7]
        greater_than
        ?# branch_offset_long[ -#183- ]
        nop
        # jump_offset_long[ -#198- ]
#183:
        push_bytes[11,0x70616e69632065726726f7221]
        --throw
#198:
        0 · push_0
        get_x[0]
        greater_than
        ?# branch_offset_long_not[ -#268- ]
        1 · push_1
        get_x[0]
        sub
        put_x[0]
        2 · push_u8[2]
        local_operand[7, +=]
        3 · push_u8[3]
        get_x[0]
        less_than
        ?# branch_offset_long[ -#227- ]
        nop
        # jump_offset_long[ -#265- ]
#227:
```

```
        50 · push_u8[50]
        get_x[0]
        concat
        call[ VFE6Zu4Wwee1vjEkQLxgVbv3c6Ju9iTaa.<e2a51fc7> ]
        pop_stack
        1 · push_1
        get_x[0]
        add
        memory_get
        --return
#265:
        # jump_offset_long[ -#198- ]
#268:
        10 · push_u8[10]
        get_x[7]
        less_than
        ?# branch_offset_long[ -#311- ]
        5 · push_u8[5]
        get_x[7]
        less_than
        ?# branch_offset_long[ -#304- ]
        2 · push_u8[2]
        5 · push_u8[5]
        get_x[0]
        mul
        concat
        get_x[7]
        concat
        call_extend_func[ check_signature(..) ]
        get_x[7]
        add
        --return
        # jump_offset_long[ -#308- ]
#304:
```

```
       2 · push_u8[2]
       local_operand[0, +=]
#308:
       # jump_offset_long[ -#319- ]
#311:
       1 · push_1
       get_x[0]
       greater_equal
       assert
       1 · push_1
       local_operand[7, +=]
#319:
       1 · push_1
       get_x[7]
       sub
       1 · push_1
       get_x[0]
       and
       div
       --return


```

The abbreviated form is:

Unset

```
ALLOC 8 CALLCODE 1 28 51 205 95 PU8 8 CU32 PUTX 0 PU8 2 CU64 PUTX 7 EXTENV 1
PUTX 7 PU8 50 GETX 0 CAT CALL 1 53 212 71 3 0 218 171 234 71 77 8 39 51 51 60
27 105 77 128 101 226 165 31 199 POP GETX 0 CALLLIB 1 125 163 55 27 POP GETX 0
CALLSTATIC 1 8 243 10 110 POP PU8 10 GETX 7 ADD CALLLOC 221 106 108 118 POP
```

```
PBUF 2 0xabc123 NTCALL 2 PUTX 0 PBUF 15 0x2268616361736822205c0a776f726c64
NTCALL 2 PUTX 0 PBUF 7 0x0000111100001111 NTCALL 2 PUTX 0 GETX 0 NTCALL 3 PUTX
7 PBUF 20 0x0135d4470300daabea474d082733333c1b694d8065 EXTFUNC 1 PUTX 7 PU8 24
GETX 0 MPUT GETX 0 MGET PUTX 7 PU8 100 GETX 7 GT BRSL 0 4 NOP JMPSL 0 15 PBUF
11 0x70616e6963206572726f7221 ERR P0 GETX 0 GT BRSLN 0 63 P1 GETX 0 SUB PUTX 0
PU8 2 XOP 7 PU8 3 GETX 0 LT BRSL 0 4 NOP JMPSL 0 38 PU8 50 GETX 0 CAT CALL 1 53
212 71 3 0 218 171 234 71 77 8 39 51 51 60 27 105 77 128 101 226 165 31 199 POP
P1 GETX 0 ADD MGET RET JMPSL 255 186 PU8 10 GETX 7 LT BRSL 0 35 PU8 5 GETX 7 LT
BRSL 0 20 PU8 2 PU8 5 GETX 0 MUL CAT GETX 7 CAT EXTFUNC 1 GETX 7 ADD RET JMPSL
0 4 PU8 2 XOP 0 JMPSL 0 8 P1 GETX 0 GE AST P1 XOP 7 P1 GETX 7 SUB P1 GETX 0 AND
DIV RET
```

IR code can be very quickly converted into Bytecode format in a one-to-one correspondence, "flattening" the program logic structure into branch jump statements for efficient execution.

Additionally, contracts can also be deployed directly in Bytecode format. Moreover, if compilers from Rust, Solidity or other languages to HVM Bytecode are developed, users can choose their preferred language to develop HVM contracts.

11 Bytecode Instruction Set

HVM utilizes single-byte opcodes while reserving space for future upgrades. Below is a draft opcode design:

0x	0	1	2	3	4	5	6	7
0	1024 Core Action						256 Ext Func	256 Env Var
1								
2	CALL addr,****(a)+	CALL_LOC ****(a)+	CALL_LIB *,****(a)+	CALL_STATIC *,****(a)+	CALL_CODE *,****		256 VM Func *(a)+	256 VM Env Var *+
3								
4	CastU8 (v)+	CastU16 (v)+	CastU32 (v)+	CastU64 (v)+	CastU128 (v)+	CastU256	CastBuffer (v)+	
5	TrimLeftZero (v)+	TrimRightZero (v)+	LeftCut *(v)+	RightCut *(v)+	Cut (v,ost,len)+	Byte (s,n)+	Size (s)+	TypeID (v)+
6								
7	BitAnd (a,b)+	BitOr (a,b)+	BitXor (a,b)+	BitNot (v)+	BitLeft (v)+	BitRight (v)+	And (x,y)+	Or (x,y)+
8	Add (a,b)+	Sub (a,b)+	Mul (a,b)+	Div (a,b)+	Mod (a,b)+	Pow (a,b)+	Sqrt (v)+	
9	AddMod (x,y,z)+	MulMod (x,y,z)+						
A								
B	HGrow $ *	HRead (len,ost)+	HWrite (v,ost)	HReadU *+	HReadUL **+	HWriteX *(v)	HWriteXL **(v)	HLength +
C	STimeRent $ (t)	SLoad (k)t,v	SSave (v,k)					
D								
E	Block **(v,...)	IF (a,b,x)+	While (a,x)+					
F	JmpL **	JmpS *	JmpSL **	BrL **(t)	BrS *(t)	BrSL **(t)	BrNSL **(t)	

8	9	A	B	C	D	E	F
PushU8 *+	PushU16 **+	Push0 +	Push1 +	PushEmptyBuf +	PushBufL **~+	PushBuf *~+	
Dup (a)a,v	Pop (v)	PopN *(v...)	JoinN *(v...)+	Join (v...,n)+	Cat (a,b)+	Swap (a,b)b,a	Reverse *(v...)
Not (v)+	Eq (x,y)+	Neq (x,y)+	Lt (x,y)+	Gt (x,y)+	Le (x,y)+	Ge (x,y)+	Rg [a,b) (a,b,x)+
Inc (v)+	Dec (v)+	Max (a,b)+	Min (a,b)+	Shl (x,n)+	Shr (x,n)+		Chiose (a,b,x)+
			GetX (i)+	PutX (v,i)	Get *+	Put *(v)	Alloc $ *
Log (s)	LogN *(v...)			MGet (k)v	MPut (v,k)	GGet (k)v	GPut (v,k)
			BurnGas $	Nop	Return (v)	ErrAbort (e)	End
							NeverTouch

References

[1] [Anon.], *Hacash: A Cryptocurrency System for Large-Scale Payments and Real-Time Settlement*, 2018, https://hacash.org/whitepaper.pdf.

[2] Hacash.com Team, *Hacash Layer3: A Multi-Chain System That Brings Sound Money Into The Daily Economy*, 2022, https://hacash.com/doc/l3-whitepaper.pdf.

[3] Jojoin, *Equity account model and readable contract syntax tree abstraction*, 2023, https://github.com/hacash/paper/blob/master/HIP/protocol/account_and_syntax_tree_abstraction.md

[4] *Hacash Layer-2 Introduction*, https://hacash.org/layer-2-intro

[5] Vincent, *Proposal for the Issuance of External Assets on the Hacash Mainnet (Draft)*, HacashTalk Forum, 2024, https://hacashtalk.com/t/proposal-for-the-issuance-of-external-assets-on-the-hacash-mainnet-draft/338

[6] Bitcoin Wiki Contributors, *List of address prefixes*, https://en.bitcoin.it/wiki/List_of_address_prefixes

[7] *HAC Currency Issuance Rules Consensus Proposal*, 2023, https://github.com/hacash/paper/blob/master/HIP/currency/HAC_currency_issuance_rules_consensus_proposal.pdf

[8] V. Buterin. *Should Ethereum be okay with enshrining more things in the protocol*, 2023, https://vitalik.eth.limo/general/2023/09/30/enshrinement.html