Cap9 Specification

Jacob Payne Jake O'Shannessy Alexey Troitskiy

June 12, 2019

Abstract

We describe a secure and extensible operating system for smart contracts. Using a capability-based exokernel protocol, we can facilitate secure isolation, perform upgrades in a secure and robust manner, and prevent privilege escalation at any point in the development process. The protocol is intended to serve as an open standard and building block upon which more advanced and tailored security models can be built.

Contents

1 Definitions

1.1 Kernel Instance

A kernel instance is an Ethereum contract which using the Cap9 kernel. It is this contract that holds the storage data etc. of the system.

2 Kernel Storage

This is a critical feature. *All* data stored by the kernel is stored in a region of *storage* called *kernel storage*. *No* other data can be stored in *kernel storage*.

This kernel storage is defined as all storage locations within a certain storage range. All values in kernel storage share the same storage key prefix. This prefix is currently `0xff ff ff ff`, and so currently includes all values in the interval [`0xffffffff00` ··· `00`, `0xffffffffff` ··· `ff`].

Table 1: Overview of kernel storage layout.

0x	00 - 03	04	05 - 1f	1e - 20	Description
0x	ff ff ff ff	00	Proc Key	···	Procedure Heap
0x	ff ff ff ff	01	00 ··· 00	00 00 00	# of Procedures
0x	ff ff ff ff	01	Key Index	00 00 00	Procedure List
0x	ff ff ff ff	02	00 ··· 00	00 00 00	Kernel Address
0x	ff ff ff ff	03	00 ··· 00	00 00 00	Current Procedure
0x	ff ff ff ff	04	00 ··· 00	00 00 00	Entry Procedure

3 Procedures

Data about procedure is store in 2 locations: the procedure table, and the procedure heap. The procedure table is simply a managed, enumerable, and iterable list of procedure keys (procedure key being something that identifies a particular procedure). The data associated with a procedure, such as its Ethereum address and its capabilities are stored on the procedure heap. Given a procedure key, the data associated with a procedure can be located using that key.

3.1 Procedure Key

Each procedure is defined by a procedure key. This key is a sequence of 24 8-bit bytes. This is treated as a sequence of 192 bits in all cases.

The kernel itself treats the procedure keys as an opaque 192-bit value, however, the capability system and some functions apply prefixes to these key value. For example: capabilities that rely on procedure key values will often use a prefix value to define a range of keys. This allows one to defined a hierarchy of procedures, much like a directory structrue. See those capabilities for more detail.

3.2 Procedure Index

Each procedure key included in the kernel is given an index, which identifies the procedure in the procedure list. It is 1-based, that is the first value is 1, and the value 0 is a null value.

As shown below, the maximum number of procedures is $2^{24} - 1 = 16,777,215$, therefore the maxium value of the procedure index is $16,777,215$, therefore the procedure index lies in the range $[1, 1677215]$.

For this reason the procedure index is specified as 24 bytes.

3.3 Procedure List

The procedure list is simply an array of storage values. The first value is the length of this list, and the subsequent values are the procedure keys of the list.

When a procedure is added to the kernel:

1. The procedure data is added to the procedure heap (see Section 3.4).
2. The procedure key is appended to the end of the array.
3. If the length value is equal to the maximum procedure index value, abort.
4. The length value (the first value) is incremented by one.
5. The procedure index value (in procedure metadata) is set to the new length value.

When a procedure is deleted from the kernel:

1. If the procedure key is the same as the Entry Procedure Key, abort and throw an error.
2. If the procedure key does not exist in the list (i.e. when looking on the procedure heap no procedure index is associated with it), abort and throw an error.
3. The length value is decremented by one.
4. If the procedure being deleted is not the last in the list (i.e. it's procedure index does not equal the length of the procedure list), the last in the list is copied to overwrite the key being deleted. This also accounts for the case of an empty list.

It is important to note that none of these steps consider deletion (zeroing) of data. This is optional for efficiency and can be performed any time after the length value has been decremented.

The procedure table is stored under the prefix `0xff ff ff ff 01`. The maximum number of keys held under this configuration is $2^{24} - 1 = 16,777,215$ (1 is subtracted to account for the length value occupying a single space). This is one less than the total number of procedures that can be held by the kernel (which is $2^{24} = 16,777,216$).

```
0xff ff ff ff 01 + 24 more bytes + 3 zero bytes
```

Table 2: Procedure table.

Storage Location	Description
`0xff ff ff ff 01`	length/number of procedures
`0xff ff ff ff 01` + keyIndex	procedure key/name: 24 bytes
⋮	
`0xff ff ff ff 01` + n	procedure key/name for procedure *n*

3.4 Procedure Heap

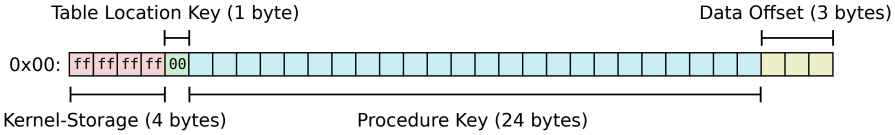
Procedure data is stored using the key `0xff ff ff ff 00`. This is combined with the procedure key to produce the following:

```
0xff ff ff ff 00 + 24 bytes procedure key + 3 bytes data offset
```

This leaves 3 bytes at the end of the storage location key. If the first of these bytes is 0x00, the storage key refers to various metadata of the procedure, including the location of its contract. If the first byte is any other value, then the storage key corresponds to the list of capabilities of that type. For example, the type of the write procedure is 7, so if the first of the last 3 bytes is 0x07, then that storage key refers to a capability of the write type.

The second value is the index into the capability list. If the second value is 0x00, this indicates the number of capabilities in this list. Therefore the capabilities start at 0x01. For example, if the third last and second last values are 0x0703, this refers to the third write capability (capability at index 0x02) held by this procedure. This implicitly limits the number of capabilities *per type* to 255 (256 minus 1 for the length value).

The very last value is an offset into the capability data. The meaning of this is different for each capability as they each have different formats. The format of each capability is specified in Section 4.



The following table outlines what data is found at each of the different locations specified by the *Data Offset*.

Table 3: Procedure data.

Data Offset				Description
0x	00	00	00	Address: 20 bytes, aligned right in the 32 bytes
0x	00	00	01	Procedure Index: 24 bytes, aligned right in the 32 bytes
0x	ty	00	00	The number of capabilities of type 0xty.
0x	ty	in	of	Capability of type 0xty, with index 0x$in - 1$, and offset 0xof into that capability.

4 System Calls and Capabilities

All system calls are transactions to the kernel. The transaction data is defined as follows:

Table 4: System call structure.

Byte Offset	Description
0x00	1 byte, system call value
0x01	1 bytes, the capability index, [0-254]
0x02	The system call data as defined by each system call
⋮	

If the transaction data is longer that specified by the system call format, the additional data is simply ignored. **TODO:** We should consider throwing an error on this.

System call numbers match capability numbers.

Table 5: System call and capability numbers.

Type Value	Description
0x0	Null capability / noop
0x3	Call Procedure
0x4	Register Procedure
0x5	Delete Procedure
0x6	Set Entry Procedure
0x7	Write
0x8	Log
0x9	Gas Send

4.1 Executing a System Call

The running code will issue a transaction to the kernel by perfoming a DELEGATECALL to the kernel with the transactions message data as both above and in the definitons of each capability. The system call will be processed as follows:

1. The kernel will receive the transaction as defined by the EVM.
2. The kernel will read the first byte of the transaction data as an unsigned 8-bit integer. This value is s.
3. If s is not one of the values listed in Table 5 then the kernel will revert the current transaction (i.e. the DELEGATECALL) and return the error code `SYSCALL_NOEXIST` (error codes are defined in Section 4.2).
4. If s is 0, it will do nothing and return successfully.
5. Otherwise the control will be passed to the code in the kernel corresponding to the s. For example if $s = 0x4$, then the code for executing a Register Procedure syscall will be called.
6. This code will execute and return whatever value it deems appropriate.

4.2 Return and Error Codes

This section defines return and error codes for interacting with the kernel. These values are returned by system calls *only* in the event of a failure of the kernel, in which case the DELEGATECALL will have been reverted. As a result, these error codes are provided only when the kernel reverts, if the DELEGATECALL returns normally, whatever is returned is determined by the procedure being called. That is, the DELEGATECALL should leave a 0 on the stack.

The error code is always 8-bits, and may followed by additional data.

Therefore:

- If 1 is left on the stack after the DELEGATECALL of a system call, then the system call succeeded, and whatever data was returned was returned by system call.
- If 0 is left on the stack after the DELEGATECALL of a system call, then the system call failed, and whatever data was returned is defined in the table below.

Table 6: Return and error codes.

Type	Value	Additional Data	Description
SYSCALL_BADCAP	0x33	None	Capability insufficient.
SYSCALL_NOGAS	0x44	None	Procedure execution ran out of gas.
SYSCALL_REVERT	0x55	Returned data from procedure	The called procedure reverted.
SYSCALL_FAIL	0x66	System call specific error data	The system call failed for reasons specific to the system call.
SYSCALL_NOEXIST	0xaa	None	The syscall integer values is not a valid syscall as defined in Table 5.

4.3 Capability Subsets

Capability B is a subset[1] of Capability A, if everything that can be done using Capability B can be done using Capabilty A. The following rules apply:

- A is always a subset of itself. $A \subseteq A$
- Transitivity: $A \subseteq B \wedge B \subseteq C \implies A \subseteq C$

As the nature of each capability is different, the way subsets are defined is different for each one. The following sections will define how subsets are defined for each capability.

4.4 Procedure Call System Call

4.4.1 System Call Format

Type Value: 3

[1] *A* subset, not a *strict* or *proper* subset.

Data: Note that here the offset is from after system call type.

Note that here we could pack the data much more densely, but we don't in order to err on the side of simplicity and stick as close as we can to 32-byte values.

Table 7: Call procedure system call format.

Byte Offset	Description
`0x00`	Procedure key, 24 bytes aligned right in 32
`0x20`	Payload for the procedure
⋮	

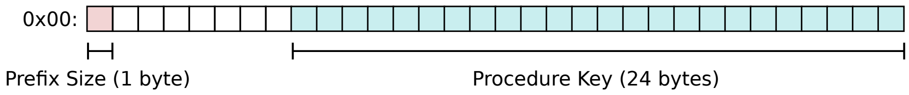
4.4.2 Capability Format

The capability format for the Call Procedure system call defines the a range of procedure keys what the capability allows one to call. This is defined as a base procedure key b and a prefix s. Given this capability, a procedure may call any procedure where the first s bits of the key of that procedure key are the same as the first s bits of procedure key b.

The values of this capability are packed into a single 32-byte value.

Table 8: Call procedure capability format.

Key Offset	Byte Offset	Description
`0x00`	`0x00`	The prefix, which is in the interval [0,24].
`0x00`	`0x01 - 0x07`	Unused and undefined.
`0x00`	`0x08 - 0x1f`	The 24 bytes of the base procedure key.
End		



4.4.3 Capability Subsets

As defined in Section 4.3, each capability has subsets and supersets. For the call capability a subset is capability which possesses the ability to call no more procedures than the original.

The prefix size of a capability C is given as C_s and the base procedure key C_b. Given a call procedure capability A and a proposed subset B:

- If the prefix size of B is equal to or greater than A (that is: $B_s \geq A_s$), and
- If the first A_s bits of B_b is equal to the first A_s bits of A_b.

4.4.4 Error Codes

If the Call Procedure system call fails for any reason other than those general system call failure conditions, the system call will return `SYSCALL_FAIL` (as definted in Section 4.2) followed by one of the following codes.

Table 9: Return and error codes.

Type	Value	Additional Data	Description
CALL_NOPROC	33	None	The procedure key specified does not exist.

Non-Normative

For example, if the Call Procedure system call was executed with a procedure key that does not exist then the DELEGATECALL will leave a 0 on the stack and return two bytes (`0x6633`).

4.5 Register Procedure System Call

4.5.1 System Call Format

Type Value: 4

Data: min(96 bytes) max(96 bytes)

Table 10: Register procedure system call structure.

Byte Offset	Description
`0x00`	Procedure key; 24 bytes, aligned right in 32 bytes
`0x20`	Procedure address; 20 bytes, aligned right in 32 bytes
`0x40`	Capabilities; A series of capabilities. Each capability is in the format specified in Table 11. The length of this list is not provided.
⋮	
End	

Table 11: Capability data as sent in the Register Procedure system call.

Byte Offset	Description
0x00	CapSize: 1 bytes, aligned right in the 32 bytes. The number of 32-byte values associated with this capability.
0x20	CapType: 1 byte, aligned right in the 32 bytes. The type of this capability.
0x40	CapIndex: 1 byte, aligned right in the 32 bytes. The index into the C-List of this type for the current procedure from which to derive a subset.
0x60	key value #1: 32 bytes
0x80	key value #2: 32 bytes
0x40 $+n \times$ 0x20	key value #n: 32 bytes
(CapSize − 1) × 0x20	final key value

NB: CapSize is the number of 32-byte values for this capability, including the CapSize, CapType, and CapIndex values. Therefore by adding CapSize × 32 to the current byte offset will give you the offset of the CapSize value of the next capability.

For each of these capabilities, CapSize is either 3, or the length of a capability of this type plus 3. If the CapSize is 3, then there is no data used to determine the correct subset, and the kernel will therefore create an exact copy of the capability to be included in the newly registered procedure.

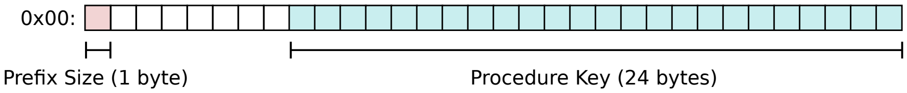
4.5.2 Capability Format

The capability format for the Register Procedure system call defines the a range of procedure keys what the capability allows one to call. This is defined as a base procedure key b and a prefix s. Given this capability, a procedure may call any procedure where the first s bits of the key of that procedure key are the same as the first s bits of procedure key b.

NB: Register procedure also relies on all of the other capabilities possessed by a procedure to determine what capabilities the new procedure will have.

Table 12: Register procedure capability format.

Key Offset	Byte Offset	Description
`0x00`	`0x00`	The prefix, which is in the interval [0,24].
`0x00`	`0x01 - 0x07`	Unused and undefined.
`0x00`	`0x08 - 0x1f`	The 24 bytes of the base procedure key.
End		



4.5.3 Capability Subsets

As defined in Section 4.3, each capability has subsets and supersets. For the register procedure capability a subset is capability which possesses the ability to register no more procedures than the original.

The prefix size of a capability C is given as C_s and the base procedure key C_b. Given a register procedure capability A and a proposed subset B:

- If the prefix size of B is equal to or greater than A (that is: $B_s \geq A_s$), and
- If the first B_s bits of B_b is equal to the first A_s bits of A_b.

4.5.4 Function

This registers a new procedure which already exists as a contract at a certain address, giving it a name an a list of capabilities. It does a number of things:

- Validate the code at the given address to show that it complies with the requirements of procedure code as described in Section 5.
- Add the procedure name to the procedure list.
- Store the specified capabilities with the procedure, but only if each of the capabilities is found to be a subset of one of the capabilities of the procedure performing this system call.

There are two highly critical functions here upon which the safety of the kernel relies. The first is the validation of the procedure bytecode, which is covered in Section 5, but it is also necessary to ensure that the capabilities that are being asked to be given to the new procedure can be provided by the current procedure. In order to satisfy this constraint it must be shown that for every requested procedure, there exists a capability in the capability list of the current procedure. It is currently not able to “combine” capabilities. That is, if the procedure has a capability of Write(0x80,5) and Write(0x85,5) it cannot provide a capability Write(0x80,10) even though it theoretically still able to perform the same writes.

The algorithm is as follows:

1. For each of the capabilities in the list of requested capabilities, search though the list of the current procedures capabilities until a superset of the requested capability is found.
2. If for any of the requested capabilities a superset is not found, abort the entire process.
3. If for all of the requested capabilities a superset is found, register the capability.

4.5.5 Error Codes

If the Register Procedure system call fails for any reason other than those general system call failure conditions, the system call will return `SYSCALL_FAIL` (as definted in Section 4.2) followed by one of the following codes.

Table 13: Return and error codes.

Type	Value	Additional Data	Description
REG_TOOMANYCAPS	77	None	Too many caps were provided.

Non-Normative

For example, if the Register Procedure system call was executed but provides too many capabilities then the DELEGATECALL will leave a 0 on the stack and return two bytes (`0x6677`).

4.6 Delete Procedure System Call

4.6.1 System Call Format

Type Value: 5

Data: min(64 bytes) max(64 bytes)

Table 14: Delete procedure system call.

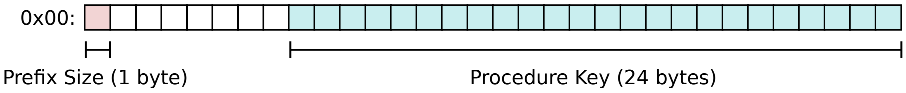
Byte Offset	Description
`0x00`	Procedure key; 24 bytes aligned right in 32 bytes
End	

4.6.2 Capability Format

The capability format for the Delete Procedure system call defines the a range of procedure keys what the capability allows one to delete. This is defined as a base procedure key b and a prefix s. Given this capability, a procedure may delete any procedure where the first s bits of the key of that procedure key are the same as the first s bits of procedure key b.

Table 15: Delete procedure capability format.

Key Offset	Byte Offset	Description
`0x00`	`0x00`	The prefix, which is in the interval [0,24].
`0x00`	`0x01 - 0x07`	Unsed and undefined.
`0x00`	`0x08 - 0x1f`	The 24 bytes of the base procedure key.
End		



4.6.3 Capability Subsets

As defined in Section 4.3, each capability has subsets and supersets. For the delete procedure capability a subset is capability which possesses the ability to delete no more procedures than the original.

The prefix size of a capability C is given as C_s and the base procedure key C_b. Given a delete procedure capability A and a proposed subset B:

- If the prefix size of B is equal to or greater than A (that is: $B_s \geq A_s$), and
- If the first B_s bits of B_b is equal to the first A_s bits of A_b.

4.7 Set Entry Procedure System Call

The value of the Entry Procedure is 24-byte procedure key which identified which procedure should be first called upon receiving a transaction. This is currently store at the storage location:

`0x00 00 00 00 04 00 ··· 00 00 00 00`

4.7.1 Error Codes

If the Delete Procedure system call fails for any reason other than those general system call failure conditions, the system call will return `SYSCALL_FAIL` (as definted in Section 4.2) followed by one of the following codes.

Table 16: Return and error codes.

Type	Value	Additional Data	Description
DEL_NOPROC	33	None	The procedure key specified does not exit.

Non-Normative

For example, if the Delete Procedure system call was executed with a procedure key that does not exist then the DELEGATECALL will leave a 0 on the stack and return two bytes (`0x6633`).

4.7.2 System Call Format

Type Value: 6

Data: min(64 bytes) max(64 bytes)

Table 17: Set entry procedure system call format.

Byte Offset	Description
`0x00`	Procedure key; 24 bytes aligned right in 32 bytes
End	

4.7.3 Capability Format

If this capability is present, the procedure is permitted to set the entry procedure, and if it is not present, the procedure is not permitted to set the entry procedure. Therefore this capability has no data associated with it and no format.

4.7.4 Capability Subsets

As defined in Section 4.3, each capability has subsets and supersets. As the Set Entry Procedure capability is a trivial value, there is only one value for Set Entry Procedure capabilities and they are all equal.

Given a Set Entry Procedure capability A and any other Set Entry Procedure capability B, not only is $B \subseteq A$ but also $B = A$.

4.7.5 Function

The only possible state change is the "entryProcedure" value. No other state changes should occur.

4.7.6 Error Codes

If the Set Entry Procedure system call fails for any reason other than those general system call failure conditions, the system call will return `SYSCALL_FAIL` (as definted in Section 4.2) followed by one of the following codes.

Table 18: Return and error codes.

Type	Value	Additional Data	Description
SETENT_NOPROC	33	None	The procedure key specified does not exit.

Non-Normative

For example, if the Set Entry Procedure system call was executed with a procedure key that does not exist then the DELEGATECALL will leave a 0 on the stack and return two bytes (`0x6633`).

4.8 Write System Call

This system call will write a single 32-byte value under a single 32-byte key in the storage of the kernel instance.

4.8.1 System Call Format

Type Value: 7

Data:

Table 19: System call structure.

Byte Offset	Description
`0x00`	Write address; 32 bytes
`0x20`	Write values; 32 bytes
End	

4.8.2 Capability Format

The write capability includes 2 values: the first is the base address where we can write to storage. The second is the number of *additional* addresses we can write to. For example, if the first value is 0x8000, and the second value is 0, we can write only to location 0x8000. If the second value was 5, we could write to 0x8000, 0x8000+1, 0x8000+2, 0x8000+3, 0x8000+4, and 0x8000+5.

Table 20: Write capability format.

Key Offset	Value	Description
`0x00`	a	The base storage address
`0x01`	n	The number of additional keys we can write to
End		

4.8.3 Capability Subsets

As defined in Section 4.3, each capability has subsets and supersets.

Given a Write capability A and a proposed subset B, where C_a is the base address of capability C and C_n is the number of additional keys writable with capability C, $B \subseteq A \iff (B_a \geq A_a) \land (B_a + B_n \leq A_a + A_n)$.

Or more verbosely, B is only a subset of A if and only if:

- The lowest writable address (which is the base address) of B is greater than or equal to the lowest writable address of A, and
- The highest writable address (which is base address plus the number of additional keys) of B is less than or equal to the highest writable address of A.

4.8.4 Error Codes

There are no specific error conditions for SSTORE, therefore only general system call errors are applicable.

4.9 Log System Call

4.9.1 System Call Format

Type Value: 8

Data: min(96 bytes) max(224 bytes)

Table 21: System call structure.

Byte Offset	Description
0x00	Number of topics (nTopics); 32 bytes
...	Potentially topic value #1; 32 bytes
...	Potentially topic value #2; 32 bytes
...	Potentially topic value #3; 32 bytes
...	Potentially topic value #4; 32 bytes
0x40+(nTopics × 0x20) ⋮	Log value any length
End	

4.9.2 Capability Format

The Write capability includes between 0 and 4 values. Each value forces the use of a particular value. For example: if the capability has 1 value, then that means that the first topic in the log call must equal that. If it has 2 values then the first log topic must be the first of those values and the second topic must be the second of those values and so on. If there are no topics listed then there are no restrictions on what can be logged.

This capability is 5 values. If the number of enforced topics is less than 4, then the unused values are undefined.

Table 22: Log capability format.

Key Offset	Value	Description
0x00	[0x00, 0x4]	Number of enforced topics; 32 bytes
0x01	t_1	Potentially an enforced value for the first topic
0x02	t_2	Potentially an enforced value for the second topic
0x03	t_3	Potentially an enforced value for the third topic
0x04	t_4	Potentially an enforced value for the fourth topic
End		

4.9.3 Capability Subsets

As defined in Section 4.3, each capability has subsets and supersets.

Given a Log capability A and a proposed subset B, $B \subseteq A$ if and only if for each log topic of A the topic is either undefined or equal to that of B.

If C_{t_i}, where i is 1, 2, 3, or 4, is the log topic i of capability C, and C_n is the number of defined topics:

$$\forall i \in \{1, 2, \ldots, A_n\} . B_{t_i} = A_{t_i}$$

4.9.4 Error Codes

There are no specific error conditions for LOG, therefore only general system call errors are applicable.

4.10 External Call System Call

This system call uses the `CALL` functionality to call and/or send Ether to another address.

4.10.1 System Call Format

Type Value: 9

Data:

Table 23: System call structure.

Byte Offset	Description
`0x00`	Account address; 20-bytes aligned right in 32 bytes
`0x20`	Value amount; 32 bytes
`0x40`	Payload for the contract
⋮	
End	

4.10.2 Capability Format

There are three distinct values in the External Call capability:

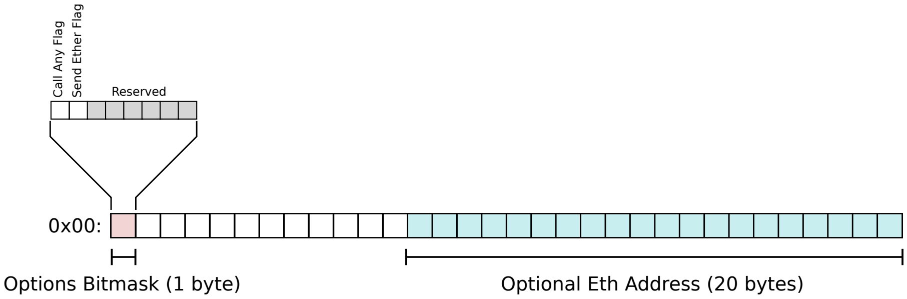
- The **CallAny** Flag: If this flag is true (the bit is set to 1) then any Ethereum address can be called. If this flag is false (the bit is set to 0) then only the address specified by EthAddress can be called.
- The **SendValue** Flag: If this flag is true (the bit is set to 1) then any quantity of Ether/Value can be sent as part of the call. If this flag is false (the bit is set to 0) then no Ether/Value can be sent, and the system call will only succeed if the value parameter is set to zero.

- **EthAddress**: If CallAny is true, this value is undefined, otherwise it is the single address that this capability permits to be called.

These three values are contained in a single storage key, formatted as per the table and diagram below. The CallAny flag is the first bit of the storage value, the SendValue flag is the second bit of the storage value, and the EthAddress is the last 20 bytes of the storage value.

Table 24: External call capability format.

Key Offset	Byte Offset	Bit Offset	Description
`0x00`	`0x00`	0x0	CallAny Flag: If 1, any address can be called, else only specified EthAddress.
`0x00`	`0x00`	0x1	SendValue Flag: If 1, Ether can be sent, else not.
`0x00`	`0x01 - 0x0b`	-	Unused and undefined.
`0x00`	`0x0c - 0x1f`	-	EthAddress: The 20 bytes of the Eth address.
End			



4.10.3 Capability Subsets

As defined in Section 4.3, each capability has subsets and supersets.

For a capability C:

- C_c is the CallAny flag.
- C_s is the SendValue flag.
- C_a is the optional EthAddress.

Given an External Call capability A and any other External Call capability B, B is a sunset of A iff:

- If B_c is true, then A_c must be true.
- If B_c is false, then A_c must be true or B_a must equal A_a.
- If B_s is true, then A_s must be true.

If these conditions hold then B is a subset of A. That is: $B \subseteq A \iff (B_c \implies A_c) \wedge ((\neg B_c \implies A_c) \vee (B_a = A_a)) \wedge (B_a \implies A_s)$.

5 Procedure Bytecode Validation

In order to be added to the procedure list of a kernel, the bytecode of a contract must be validated by the kernel and must conform to a strict set of requirements. These requirements are as follows:

- All the opcodes must be on the whitelist as defined in Section 5.1, unless part of a system call.
- The first opcodes of the contract must form a valid execution guard as defined in Section 5.2.

5.1 Whitelist

Generally the whitelist consists of all of the non-state-changing opcodes (REVERT being a notable exception). This *must* be implemented as a whitelist and not a blacklist, as this should continue to work in the case that new state-changing opcodes are added to the VM.

The table below contains the ranges of opcodes that are allowed.

Table 25: Opcode whitelist.

Range (Inclusive)	Description
`0x00 - 0x0b`	Stop and Arithmetic
`0x10 - 0x1a`	Comparison & Bitwise Logic Operations
`0x20`	SHA3
`0x30 - 0x3e`	Environmental Information
`0x40 - 0x45`	Block Information
`0x50 - 0x53`	Stack, Memory, Storage and Flow Operation - Part 1
`0x54`	SLOAD
`0x56 - 0x5b`	Stack, Memory, Storage and Flow Operation - Part 2
`0x80 - 0x8f`	Duplication Operations
`0x90 - 0x9f`	Exchange Operations
`0xf3`	Return
`0x60 - 0x7f`	Push
`0xfa`	STATICCALL
`0xfd`	REVERT
`0xfe`	INVALID

5.2 Execution Guard

An execution guard *must* start at position `0x00` in the bytecode. The purpose of the exeuction guard is to ensure that the procedure is being called by a kernel. The risk is that somebody might perform a CALL to a bare procedure contract. The procedure contract will then DELEGATECALL back to whoever called it, giving them the power to execute a SELFDESTRUCT instruction. In order to prevent this, we want to make sure that the procedure is only being called via CALLCODE, DELEGATECALL, or STATICCALL. On simple way to check this is to check that a kernel address is stored in storage. This means the contract is being called either by a kernel, or something pretending to be a kernel. Because this value is undefined in the procedures own storage, we will abort if it is undefined.

It is defined as follows:

Listing 1: Sequence of steps which constitutes an execution guard.

```
0x00: PUSH 0xffffffff0200...00 // Push kernel addr. location
0x21: SLOAD     // Load value to the stack
0x22: PUSH 0x2a // Load value to the stack
0x24: JUMPI     // Jump if kernel address non-zero
0x25: PUSH1 0x00 // Revert data size
0x27: PUSH1 0x00 // Revert data location
0x29: REVERT    // Revert because we are not a kernel
0x2a: JUMPDEST  // Destination to jump over
```

5.3 System Call

A system call is in the form:

Listing 2: Sequence of steps to perform a system call.

```
CALLER       // Get Caller
GAS          // Put all the available gas on the stack
DELEGATECALL // Delegate Call to Caller
```

6 Initialization

In order to initialize a Cap9 kernel instance, we need to provide the user an initialization routine. The user must provide an entry procedure to act as the initial interface as well as the root capabilities available to the system. Initialization is done as follows:

- For the entry procedure, either an already existing procedure is chosen or deployed within the network, including the id.
- A list of capabilities is chosen that will be minted for the entry procedure.
- On deployment of the cap9 kernel code, the address of the entry procedure and the capability list are given as inputs 26, however the parent capability index in each field are ignored.
- Following this the entry procedure id is set as the entry procedure.

Table 26: Initialization input structure.

Byte Offset	Description
`0x00`	Procedure key; 24 bytes, aligned right in 32 bytes
`0x20`	Procedure address; 20 bytes, aligned right in 32 bytes
`0x40`	Capabilities; A series of capabilities. Each capability is in the format specified in Table 11. The length of this list is not provided.
⋮	
End	