Patterns to filter out native SHA3 Operations

KECCAK256 is natively used in Solidity to maintain the mapping and dynamic array data types and compute the topics for on-chain events. Since we only focus on *user-initiated* cryptographic operations in smart contracts, we filtered *native* KECCAK256 operations based on the following patterns.

Mapping and Dynamic Array

Storage Layout in Solidity

We refer to Solidity documentation for full description of Mapping and Dynamic Array storage layout rules. https://docs.soliditylang.org/en/v0.8.20/internals/layout_in_storage.html

> Due to their unpredictable size, mappings and dynamically-sized array types cannot be stored “in between” the state variables preceding and following them. Instead, they are considered to

> occupy only 32 bytes with regards to the the storage layout rules and the elements they contain are stored starting at a different storage slot that is computed using a Keccak-256 hash.
>
> Assume the storage location of the mapping or array ends up being a slot `p` after applying the storage layout rules. For dynamic arrays, this slot stores the number of elements in the array (byte arrays and strings are an exception, see below). For mappings, the slot stays empty, but it is still needed to ensure that even if there are two mappings next to each other, their content ends up at different storage locations.
>
> Array data is located starting at `keccak256(p)` and it is laid out in the same way as statically-sized array data would: One element after the other, potentially sharing storage slots if the elements are not longer than 16 bytes. Dynamic arrays of dynamic arrays apply this rule recursively. The location of element `x[i][j]`, where the type of `x` is `uint24[][]`, is computed as follows (again, assuming `x` itself is stored at slot `p`): The slot is `keccak256(keccak256(p) + i) + floor(j / floor(256 / 24))` and the element can be

obtained from the slot data `v` using `(v >> ((j % floor(256 / 24)) * 24)) & type(uint24).max`.

The value corresponding to a mapping key `k` is located at `keccak256(h(k) . p)` where `.` is concatenation and `h` is a function that is applied to the key depending on its type:

- for value types, `h` pads the value to 32 bytes in the same way as when storing the value in memory.
- for strings and byte arrays, `h(k)` is just the unpadded data.

If the mapping value is a non-value type, the computed slot marks the start of the data. If the value is of struct type, for example, you have to add an offset corresponding to the struct member to reach the member.

As an example, consider the following contract:

open in Remix

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;


contract C {
    struct S { uint16 a; uint16 b; uint256
c; }
    uint x;
    mapping(uint => mapping(uint => S))
data;
}
```

Let us compute the storage location of `data[4][9].c`. The position of the mapping itself is `1` (the variable `x` with 32 bytes precedes it). This means `data[4]` is stored at `keccak256(uint256(4) . uint256(1))`. The type of `data[4]` is again a mapping and the data for `data[4][9]` starts at slot `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1)))`. The slot offset of the member `c` inside the struct `S` is `1` because `a` and `b` are packed in a single slot. This means the slot for `data[4][9].c` is `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1))) + 1`. The type of the value is `uint256`, so it uses a single slot.

Patterns

Except for mapping and dynamic array, Solidity arranges the storage location of all variables in sequential order. Therefore, if the result of the KECCAK opcode is directly used for Storage access (SLOAD/SSTORE), it is a *native* operation used for reading/writing mapping and dynamic array variables.

Pattern 1: KECCAK-SLOAD/SSTORE

Pattern 1 can filter out mapping/dynamic access operations such as data[x]. However, as the elements in a mapping/array may be structs, the actual storage location accessed might incur an offset based on the KECCAK result: `SLOT(data[x].c)=SLOT(data[x])+ some constant offset`. We introduce Pattern 2 to recognize such situations: if the result of the KECCAK opcode, along with an added offset, is used for Storage access (SLOAD/SSTORE), it is a *native* operation used to read/write mapping and dynamic array variables.

Pattern 2: KECCAK-ADD-SLOAD/SSTORE

Finally, we introduce Pattern 3 to handle multi-layer mappings/arrays. For example, the access to data[a][b] will be interpreted as: `keccak256(b|keccak256(a|p))`, where p is the placeholder of 'data' and | is bytes concatenation. Therefore, if the result of KECCAK is stored in memory (MSTORE) and later used as input for the next KECCAK operation, where the resulting value is utilized for SLOAD/SSTORE, it is a *native* operation used to read/write multi-layer mappings and dynamic array variables.

Pattern 3: KECCAK-(ADD)-[MSTORE-(ADD)-KECCAK]*-SLOAD/SSTORE

In pattern 3, (ADD) means the ADD opcode is optional (only needed when there is a struct), and * means match 0 or more repetitions (Each layer of mapping/dynamic array adds a repetition).

Event

Event Topics in Solidity

We refer to Solidity documentation for the full description of Events in Solidity. https://docs.soliditylang.org/en/v0.8.20/abi-spec.html#events.

> Events are an abstraction of the Ethereum logging/event-watching protocol. Log entries provide the contract's address, a series of up to four topics and some arbitrary length binary data. Events leverage the existing function ABI in order to interpret this (together with an interface spec) as a properly typed structure.
>
> Given an event name and series of event parameters, we split them into two sub-series: those which are indexed and those which are not. Those which are indexed, which may number up to 3 (for non-anonymous events) or 4 (for anonymous ones), are used alongside the Keccak hash of the event signature to form the topics of the log entry. Those which are not indexed form the byte array of the event.
>
> In effect, a log entry using this ABI is described as:
>
> - `address`: the address of the contract (intrinsically provided by Ethereum);
> - `topics[0]`: `keccak(EVENT_NAME+"("+EVENT_ARGS.map(canonical_type_of).join(",")+")")` (`canonical_type_of` is a function that simply returns the canonical type of a given

> argument, e.g. for `uint indexed foo`, it would return `uint256`). This value is only present in `topics[0]` if the event is not declared as `anonymous`;
>
> - `topics[n]`: `abi_encode(EVENT_INDEXED_ARGS[n - 1])` if the event is not declared as `anonymous` or `abi_encode(EVENT_INDEXED_ARGS[n])` if it is (`EVENT_INDEXED_ARGS` is the series of `EVENT_ARGS` that are indexed);
> - `data`: ABI encoding of `EVENT_NON_INDEXED_ARGS` (`EVENT_NON_INDEXED_ARGS` is the series of `EVENT_ARGS` that are not indexed, `abi_encode` is the ABI encoding function used for returning a series of typed values from a function, as described above).

Patterns

Pattern 4: KECCAK+LOG[1-4]

KECCAK is natively used to calculate topics[0] for events. Therefore, we use the following pattern to recognize such operations: if the result of KECCAK is later used as the topic[0] of LOG1, LOG2, LOG3, LOG4

opcodes, it is a native operation used for calculating on-chain topics.