# Appendix A: Details on how we analyze the blockchain

Here is the bytecode that the Solidity compiler generates for a `send()` instruction whose return value is ignored:

“CALL SWAP4 POP POP POP POP POP”

When the `CALL` instruction completes, it places 5 elements on the stack, the first of which is the return code. All of these elements are then `POP`ped before they can be used at all. The `SWAP4` instruction is actually redundant here --- it swaps the 1st element with the 5th, but all of them are discarded anyway).

When the return value is stored/checked immediately after the function call, the opcode becomes:

```

CALL SWAP4 POP POP POP POP **SWAP1** POP

```

Because the `SWAP1` instruction pushes the return value even further into the value stack, this opcode pattern leaves the return value available to the subsequent code. Therefore, we can quickly determine whether the send function’s return value is ignored by counting the number of consecutive `POP`s after `CALL SWAP4`.

To identify the other defensive technique, which uses a test message to check whether the callstack is empty, we check whether a contract contains a second `CALL` from within the same top-level function as the `send` instruction.

This requires segmenting the contract code into its separate top-level functions. To do this, we look for the following pattern, which shows up as a “delimiter” between functions:

```

“CALL ISZERO PUSH [ERROR TAG] JUMPI”

```

We can also use this delimiter to guess whether a blockchain contract is compiled from Solidity or from Serpent/LLL.