

the attacker will be required to submit a value for STARTGAS limiting the number of computational steps that execution can take, so the miner will know ahead of time that the computation will take an excessively large number of steps.

- An attacker sees a contract with code of some form like `send(A,contract.storage[A]); contract.storage[A] = 0`, and sends a transaction with just enough gas to run the first step but not the second (ie. making a withdrawal but not letting the balance go down). The contract author does not need to worry about protecting against such attacks, because if execution stops halfway through the changes get reverted.
- A financial contract works by taking the median of nine proprietary data feeds in order to minimize risk. An attacker takes over one of the data feeds, which is designed to be modifiable via the variable-address-call mechanism described in the section on DAOs, and converts it to run an infinite loop, thereby attempting to force any attempts to claim funds from the financial contract to run out of gas. However, the financial contract can set a gas limit on the message to prevent this problem.

The alternative to Turing-completeness is Turing-incompleteness, where JUMP and JUMPI do not exist and only one copy of each contract is allowed to exist in the call stack at any given time. With this system, the fee system described and the uncertainties around the effectiveness of our solution might not be necessary, as the cost of executing a contract would be bounded above by its size. Additionally, Turing-incompleteness is not even that big a limitation; out of all the contract examples we have conceived internally, so far only one required a loop, and even that loop could be removed by making 26 repetitions of a one-line piece of code. Given the serious implications of Turing-completeness, and the limited benefit, why not simply have a Turing-incomplete language? In reality, however, Turing-incompleteness is far from a neat solution to the problem. To see why, consider the following contracts:

```
C0: call(C1); call(C1);  
C1: call(C2); call(C2);  
C2: call(C3); call(C3);  
...  
C49: call(C50); call(C50);  
C50: (run one step of a program and record the change in storage)
```

Now, send a transaction to A. Thus, in 51 transactions, we have a contract that takes up  $2^{50}$  computational steps. Miners could try to detect such logic bombs ahead of time by maintaining a value alongside each contract specifying the maximum number of computational steps that it can take, and calculating this for contracts calling other contracts recursively, but that would require miners to forbid contracts that create other contracts (since the creation and execution of all 50 contracts above could easily be rolled into a single contract). Another problematic point is that the address field of a message is a variable, so in general it may not even be possible to tell which other contracts a given contract will call ahead of time. Hence, all in all, we have a surprising conclusion: Turing-completeness is surprisingly easy to manage, and the lack of

