출처: http://earlz.net/view/2017/08/13/0451/the-faults-and-shortcomings-of-the-evm

EVM의 결함과 단점

먼저 저의 소개를 먼저 해봅니다. 저는 Qtum 공동 창립자입니다. Qtum 프로젝트는 기본적으로 Ethereum Virtual Machine(EVM)을 가져와서 블록체인에 이더리움(다른 무언가와 함께) 이상의 것을 배치 하는 프로젝트 입니다. 저는 이일을 하는 도중에 EVM에 대해서 알고자 하는것보다 더많은 것을 배워야 했습니다. 이러한 학습이 저에게 발전 되냐고 묻는다면 저는 신경쓰지 않습니다. 저는 개인적으로 EVM은 실용적이지 않은 디자인이며 구현하기 어렵다고 봅니다. 고지 사항으로, 저는 이러한 문제의 대부분을 해결 할 수 있는 Qtum 에 다른 VM을 추가한다는 사실을 알고자 노력할 것 입니다.

어쨋든 이제 본론으로 돌아가 봅시다. EVM의 요점은 무엇이며, 왜 처음에 만들어 졌을까요? 합리적 설계론(Design Rational)에 따르면 다음 용도로 설계 되었습니다:

1. 간결성
2. 결정론
3. 소형 bytecode 크기
4. 블록체인을 위한 전문성
5. 간결성(어?)
6. 최적화 가능성

그리고 그 문서(Design Rational)를 보시면 EVM에 대한 추론이 제법 잘 설명되어 있습니다. 그럼 어디에서 잘못 됐을까요? 음, 오늘날의 기술과 패러다임에서는 잘 맞지 않습니다. 그것은 현재 존재하지 않는 세계를 위해 만들어진 아주 훌륭한 디자인입니다. 이 문제에 대해 다시 돌아보겠습니다. 그러나 제가 싫어하는 점 부터 시작하겠습니다.

256비트 정수

최신 프로세서의 경우 빠른 계산을 위해 4가지 선택 사항이 있습니다:

1. 8비트 정수
2. 16비트 정수
3. 32비트 정수
4. 64비트 정수

물론 어떤 경우에는 32비트가 16비트보다 빠르고, x86에서 8비트 연산은 완전히 지원하지 않습니다(다시말해, 직접적인 나눗셈이나 곱셈이 없다). 그러나 대부분의 경우 이 크기중 하나를 사용하면 수학 연산에 사용되는 시간을 보장 받을수 있으며 캐쉬 손실 및 메모리 지연을 포함하지 않으면 수나노초 내로 측정이 빨라집니다. 어쨌든, 이것들은 현대 프로세서가 외부 조작을 필요로 하는 어떠한 번역이나 다른 것들이 필요 없이 "본래부터" 사용하는 정수의 크기라고 말 할 수 있습니다.

물론 EVM은 속도와 효율성을 최적화하기 위한 것이기 때문에 정수 크기의 선택은 다음과 같습니다:

1. 256비트 정수

참고로 x86 어셈블리에 2개의 32비트 정수를 추가하는 방법은 다음과 같습니다(예를 들면 여러분이 가지고 있는 PC의 프로세서)

```
mov eax, dword [number1]
mov eax, dword [number2]
```

다음은 프로세서가 64비트를 사용할 수 있다고 가정 할 때 x86 어셈블리에 2개의 64비트 정수를 추가하는 방법입니다:

```
mov rax, qword [number1]
mov rax, qword [number2]
```

32비트 x86 컴퓨터에 2개의 256비트 정수를 추가하는 방법은 다음과 같습니다

```
mov eax, dword [number]
add dword [number2], eax
mov eax, dword [number1+4]
adc dword [number2+4], eax
mov eax, dword [number1+8]
adc dword [number2+8], eax
mov eax, dword [number1+12]
adc dword [number2+12], eax
mov eax, dword [number1+16]
adc dword [number2+16], eax
mov eax, dword [number1+20]
adc dword [number2+20], eax
mov eax, dword [number1+24]
adc dword [number2+24], eax
mov eax, dword [number1+28]
```

```
adc dword [number2+28], eax
```

ㅍ64비트 x86 컴퓨터에 이러한 256비트 정수를 추가하는 것은 약간 좋습니다

```
mov rax, qword [number]
add qword [number2], rax
mov rax, qword [number1+8]
adc qword [number2+8], rax
mov rax, qword [number1+16]
adc qword [number2+16], rax
mov rax, qword [number1+24]
adc qword [number2+24], rax
```

어쨌든, 256비트 정수로 작업하는 것은 프로세서가 기본적으로 지원하는 정수 길이로 작업하는 것보다 훨씬 복잡하고 느리다라고 말하기에 충분합니다.

EVM의 경우 다른 정수 크기로 작업하기 위한 opcodes를 추가하는 것 보다는 256비트 정수만 지원하는 것이 훨씬 간단하기 때문에 이 디자인을 채택합니다. 256비트가 아닌 연산은 메모리에서 1-32 바이트의 데이터를 가져오는 일련의 푸시 명령어와 8비트 정수로 작동하는 몇 가지 명령어가 있습니다.

그렇다면 모든 연산에서 이렇게 비효율적인 정수 크기를 사용하는 합리적 설계는 무엇일까요?

> "4바이트 또는 8바이트 단어는 암호 계산을 위한 주소 및 큰 값을 저장하기에 너무 제한적이며 무제한 값은 안전한 가스 모델을 만들기가 너무 어렵습니다."

저는 2개의 주소를 하나의 작업과 비교할 수 있다는 점을 인지하고 있습니다. 그러나 x86에서 32비트 모드로 동일한 작업을 수행하는 방법은 다음과 같습니다(SSE및 기타 최적화 없이):

```
mov esi, [address1]
mov edi, [address2]
mov ecx, 32 / 4
repe cmpsd
jne not_equal
; if reach here, then they're equal
```

address1 과 address2 가 하드코드된 주소라고 가정하면, 이는 약 6+5+5=16바이트의 opcode이거나, 주소가 스택에 있는 경우 6+3+3=12바이트의 opcode와 같을 수 있습니다.

큰 정수 크기에 대한 또 다른 명분은 "암호 계산을 위한 큰 가치" 입니다. 그러나 몇 달전에 무서를 읽은 이후로 주소나 해시가 동일한지 비교하는 것을 포함하지 않는 256비트 정수에 대해 단일 사용 예를 알아내는데 문제가 있었습니다. 사용자 맞춤 암호화는 공공 블록체인에서 실행하기에는 비용이 많이 듭니다. 저는 1시간 이상 github에서 암호화로 정의 할 수있는 Solidity 계약을

찾으려고 노력했지만 아무것도 얻지 못했습니다. 거의 모든 형태의 암호화는 최신 컴퓨터에서 느려지고 복잡해지기 때문에 가스 비용으로 인한 공개 이더리움 블록체인에서 실행하는 것은 경제적이지 않습니다(실제 알고리즘을 Solidity에 변환하려는 노력은 말 할 필요도 없습니다). 그러나 가스 비용이 중요하지 않은 사설 블록체인은 여전히 존재합니다. 또한 자신의 블록체인을 소유하고 있다면 느린 EVM 계약의 한부분으로 수행하기를 원하지 않을 것입니다. 네이티브 코드의 암호화를 pre-compiled 스마트 계약으로 구현하려면 C++, Go 또는 수많은 프로그래밍 언어를 사용합니다. 그래서 이것은 256비트 정수만 지원하는 이상한 상황에 대한 명분을 불어 넣습니다. 이 점은 제가 느끼는 EVM문제에 대한 진정한 토대이지만 보다 명확하지 않은 영역에 더 많은 문제점이 숨어있습니다.

EVM의 메모리 모델

EVM에는 데이터를 저장할 수 있는 3개의 주요 장소가 있습니다.

1. 스택
2. 임시 메모리
3. 영구 메모리

스택에는 일정한 제한이 있기 때문에 매우 비싼 영구 메모리 대신 임시 메모리를 사용해야 하는 경우가 있습니다. EVM에는 `allocate` 명령어나 이와 비슷한 것이 없습니다. 이와 같은 이유로 메모리에 쓰기를 하면서 선언을 할 수 있습니다. 이것은 꽤 똑똑해 보일지 모르겠지만 또한 매우 해로운 작업입니다. 예를 들어서 주소 0x10000에 쓰기를 하게 되면 여러분의 계약서는 64K(256비트 단어중 64K) 단어 메모리가 할당 되고 모든 64K 단어의 메모리를 사용하는 것처럼 가스 비용이 지불됩니다. 쉬운 해결 방법은 그냥 마지막으로 사용하는 메모리 주소를 추적하고 더 필요할 때 증가 시킵니다. 이것은 여러분이 사용한 시점에서 많은 메모리를 필요하지 않는 다면 좋은 점이 없습니다. 또한 더 이상 그 메모리가 필요하지 않습니다. 100 단어의 메모리를 사용하는 아주 이상한 알고리즘을 수행한다고 가정해 보겠습니다. 그래서 여러분은 그것을 할당하고, 메모리를 사용하고, 무엇이되었던 100 단어의 메모리를 사용합니다... 그런 다음 그 기능을 종료합니다. 이제 다른 기능으로 돌아오고 나서 스크래치 공간 혹은 무언가를 위해 단지 1단어의 메모리가 필요합니다. 그래서 다른 단어를 할당합니다. 이제 우리는 101 단어의 메모리를 사용하고 있습니다. 메모리를 해제 할 수 있는 방법은 없습니다. 이론적으로 메모리의 마지막 공간을 추적하고 있는 특수 포인터는 줄일 수 있지만 전체 메모리 블록이 다시 참조 되지 않고 안전하게 재사용 될 수 있을 경우에만 작동합니다. 100 단어중 50 단어와 90 단어가 필요하면 다른 위치(예를 들면 스택)에 복사한 다음 그 메모리를 해제해야 합니다. 이 부분에 대해 EVM이 제공하는 도구는 없습니다. 이것은 기술적인 용어로 메모리 단편화 입니다. 각 기능이 할당되고 전역으로 접근 가능한 메모리를 사용하지 않는지 그리고 메모리를 재사용하고 무언가 심사 과정을 거치면 계약서에 치명적

인 문제가 생길 수 있는지 여부를 확인하는 것은 사용자의 몫입니다. 그래서 선택사항은 기본적으로 메모리 재사용 버그를 통해서 보다 더 큰 클래스영역까지 열어두거나 이미 필요한 것 이상 할당 했음에도 더많은 가스를 메모리에 지불하게 됩니다.
또한 메모리를 할당하는데에 비용이 선형적으로 증가하지 않습니다. 100 단어의 메모리를 할당하고 1개의 단어를 더 할당하려면 프로그램 시작할때 메모리의 첫 번째 단어를 할당하는 것보다 훨씬 비용이 많이 듭니다. 이런 측면은 가스 비용을 감소를 위해 더많은 계약 버그로 자신을 몸담는 것에 비해 안적적인 측면에서 경제적 비용을 크게 증대 시킵니다.
그래서 왜 메모리를 사용하는 것인가요? 왜 스택을 사용하지 않습니까? 음, 스택은 엄청난 제약사항이 있습니다.

EVM의 스택

EVM은 스택 기반 시스템입니다. 다시 말하면 레지스터 집합이 아닌 대부분에 연산을 위해 스택을 사용한다는 의미입니다. 스택 기반 컴퓨터는 일반적으로 최적화는 간단하지만 레지스터 기반 컴퓨터와 비교할 때 대부분의 작업에서 보다 많은 opcode를 필요로 합니다.

어쨌든, EVM에서 많은 종류의 서로 다른 작업이 있으며 대부분은 스택위에서만 작동합니다. SWAP(값의 교환) 및 DUP(스택 상단에 복사) 시리즈 지시사항에 유의해야 합니다. 이 작업은 16단계 까지 올라갑니다. 이제 아래의 계약서를 컴파일 해봅시다:

```
pragma solidity ^0.4.13;

contract Something{
    function foo(address a1, address a2, address a3, address a4,
address a5, address a6){
        address a7;
        address a8;
        address a9;
        address a10;
        address a11;
        address a12;
        address a13;
        address a14;
        address a15;
        address a16;
        address a17;
    }
}
```

당신은 아래와 같은 오류를 맞나게 될 것입니다.

```
CompilerError: Stack too deep, try removing local variables.
```

항목이 스택에서 16 단계 깊이가 되면 스택에서 항목을 팝(Pop) 하지 않으면 더 이상 접근 할 수 없기 때문에 오류가 발생합니다. 이 문제의 공식적인 "해결책"은 변수를 적게 사용하고 함수를 더 작게 만드는 것 입니다. 다양한 해결 방법은 구조체 또는 배열에 변수를 채우는 것과 memory 키워드를 사용(이것에 대한 이유는 일반 변수에 적용 할 수 없는 것일까?) 하는 것입니다. 그래서 메모리 기반 구조체를 사용해서 계약서를 수정해봅니다.

```
pragma solidity ^0.4.13;

contract Something{
    struct meh{
        address x;
    }
    function foo(address a1, address a2, address a3, address a4,
address a5, address a6){
        address a7;
        address a8;
        address a9;
        address a10;
        address a11;
        address a12;
        address a13;
        meh memory a14;
        meh memory a15;
        meh memory a16;
        meh memory a17;
    }
}
```

이것에 대한 결과는...

```
CompilerError: Stack too deep, try removing local variables.
```

그러나 우리는 이 변수를 메모리로 대체 했나요? 아니면 그것을 고치지 않았나요? 음, 아닙니다. 이제 스택에 17개의 256비트 정수를 저장하는 대신 256비트 메모리 슬롯에 13개의 정수와 4개의 256비트 메모리 주소(메모리 참조)를 저장합니다. 이 문제의 일부분은 Solidity의 문제이지만 근본적인 문제는 EVM 스택의 임의 항목에 접근할 방법이 없다는 것 입니다. 제가 아는 모든 VM 구현은 이런 근본적인 문제를 해결 합니다.

1. 작은 스택 크기를 권장하고 스택 항목을 메모리 또는 대체 저장 장치(예를 을면 .NET 의 지역 변수)로 쉽게 바꿀 수 있습니다.
2. 임의의 스택 슬롯에 대한 접근을 허용하는 pick 명령어를 사용 하고나 이와 유사한 구현을 합니다.

그러나 EVM에서 스택은 데이터 및 계산을 위한 유일한 메모리 공간이며 다른 모든 장소는 가스 형태와 같은 직접적인 비용이 발생합니다. 따라서 다른 곳의 공간은 비용이 비싸기 때문에 작은

스택 크기 방식을 저해합니다. 그래서 우리는 이와 같은 기본 언어 구현 문제에 도달하게 됩니다.

Bytecode 크기

합리적 디자인 문서에서 EVM 바이트 코드에 대한 목표는 단순하고 간결함 입니다. 그러나 이것은 서술적이고 간결한 코드를 작성하는 것을 선호하는 것과 같습니다. 그것들은 근본적으로 다른 방식을 사용해서 근본적으로 다른 목표를 성취합니다. 간단한 명령어 세트는 연산 횟수를 제한하고 연산을 간결하고 단순하게 유지하면서 수행됩니다. 한편, 작은 프로그램을 생성하는 소형 bytecode는 가능한 많은 연산을 수행하는 명령 세트를 가능한 작은 크기의 코드로 작성함으로써 수행됩니다.

궁극적으로 "소형 bytecode 크기"가 이론적 근거기반의 목표이기는 하지만 EVM의 실제 구현은 다른 의미에서 그 목표를 달성하지 못합니다. 대신에 가스 모델을 쉽게 생성 할 수 있는 단순한 명령어 세트에 중점을 둡니다. 그리고 저는 잘못되었거나 나쁘다는 것을 말하는 것이 아닙니다. EVM의 주요 목표중 하나는 근본적으로 EVM의 다른 목표와 함께 끝납니다. 또한 이 문서에서 주어진 하나의 숫자는 C 프로그램이 "hello world"를 출력 하기 위해 4000 바이트 이상을 차지하는는 사실입니다. 이것은 C 프로그램에서 발생하는 다양한 환경과 최적화에 대한 사례는 아닙니다. 그들이 측정한 C 프로그램에서 ELF 데이터, 재배치 데이터 및 정렬 최적화가 있을 것으로 기대합니다 – 32 바이트 또는 4kb와 같은 특정 경계에서 코드와 데이터를 정렬하면 물리적 프로세서에서 프로그램의 성능에 상당한 영향을 줄 수 있습니다. 저는 개인적으로 46 바이트 x86 머신 코드로 커파일되는 간단한 베어 본 C 프로그램과 약700바이트로 컴파일되는 간단한 greeter 형식 프로그램을 만들었습니다. 반면에 Solidity의 예제는 1000바이트 이상의 EVM 바이트 코드로 컴파일됩니다.

보안상의 이유로 단순한 명령어 세트의 필요성을 이해하고 있지만, 블록체인에서 큰 공간 부풀림을 초래합니다. 마치 EVM 스마트 계약 bytecode가 가능한 작게 처리되는 것처럼 넘기는 것은 해롭습니다. 표준 라이브러리를 포함하고 이런 작업을 위해 여러가지 연산 코드를 실행하지 않고도 일반적인 연산을 수행하는 opcode를 지원함으로써 훨씬 더 작게 만들수 있습니다.

256비트 정수(또 다신 한번)

하지만 256비트 정수는 정말로 끔찍합니다. 그리고 가장 이상한 부분은 합리적으로 사용되지 않는 곳에서 사용되고 있다는 것 입니다. 4Byte(32 비트) 단위 이상의 가스를 사용하는 것은 사실상 불가능 합니다. 따라서 가스를 지정하고 점검하는데 어떤 정수 크기가 사용되나요? 물론 256비트 정수입니다. 메모리는 상당히 비쌉니다. EVM의 메모리 주소에 대한 주소 크기는 얼마일까요? 물론 당신의 계약이 우주의 원자보다 많은 메모리 공간이 필요할때 256비트 정수입니다. 영구 저장소에서 주소와 값 모두에 대해 256비트 정수를 사용하는 것에 대해 불만이 있지만, 실제로 일부

데이터에 대해 해시를 사용하고 주소 공간에서 충돌에 대해 걱정할 필요가 없는 몇 가지 흥미로운 기능을 제공합니다. 그래서 나는 이런 점이 통행권을 얻는다고 생각합니다. 모든 정수 크기를 사용할 수 있는 모든 인스턴스에서 EVM은 256비트를 호출합니다. JUMP 명령어 조차도 256비트를 사용하지만 방어를 위해 점프 목적지를 `0x7FFFFFFFFFFFFFFF`로 제한하고 점프 목적지를 부호있는 64비트 정수로 효과적으로 제한합니다. 그리고 이것은 통화 가치 그자체가 됩니다. ETH의 최소 단위는 wei 이므로 총 coin 공급량(wei 단위) `100000000000000000 * 200000000`(200M은 예상치, 현재 공급량은 약 90M) 입니다. 그래서 2의 256제곱수(256 비트 정수로 저장할 수 있는 최대값)를 빼면 1.157920892373162e+77이 됩니다. 우주에 있는 원자의 수보다 더 큰 크기를 더한 것보다 더 많은 wei를 보낼수 있는 충분한 공간입니다. 기본적으로 256비트 정수는 EVM이 설계된 모든 어플리케이션에서 믿기지 않을 정도로 비효율적이며 불필요합니다.

표준 라이브러리 부족

Solidity 스마트 계약을 개발한 적이 있다면 표준 라이브러리 부족은 문제들 중 첫번째 문제일 것입니다. 표준 라이브러리가 전혀 존재 하지 않기 때문에 두 문자열이 같은지 확인하고 싶다면 `strcmp`나 `memcmp` 같은 것이 없으므로 코드를 직접 작성하거나 인터넷에서 코드를 복사해서 사용해야 합니다. Zepplin Project에서 계약에서 사용할 수 있는 표준 라이브러리를 제공함으로써 이런 상황을 견딜 수 있게 합니다(계약 자체에 포함 시키거나 외부 계약을 호출하여 사용). 그러나 이 방법의 한계는 두 개의 SHA3 연산을 사용하고 결과 해시를 비교하는 것이 문자열(한번에 32바이트 씩)이 같은지 확인하는 것보다 저렴하다는 검을 고려할때 명백해 집니다. 합리적인 가스 가격으로 기본 코드를 사용하는 pre-compiled 계약서의 표준 라이브러리가 있으면 전체 스마트 계약 생태계에 큰 도움이 됩니다. 그러나 만약 표준 라이브러리가 없으면 사람들은 오픈 소스 코드의 일부 코드를 복사여 붙여 넣어야 하는데, 이때 알려지지 않은 보안 관련 문제가 발생할 수도 있습니다. 이 외에도 사람들은 계약의 보안 프로파일을 잠재적으로 손상시킬수 있는 위험이 있더라 하더라도 코드를 최적화하고 가스 사용량을 줄여서 비용을 줄이려고 합니다.

가스의 경제학과 게임 이론

이 주제에 대한 전체 게시물을 작성할 계획이지만 EVM은 좋은 예시일 뿐만 아니라 비용도 많이 들지 않습니다. 예를 들어 블록체인에 데이터를 저장하는 것은 상당히 큰 비용이 듭니다. 다시 말하면 스마트 계약 내에서 모든 양의 데이터를 캐시하는데 엄청난 비용이 발생합니다. 그래서 대신 각 계약 실행으로 계산 됩니다. 시간이 지남에 따라 더 많은 가스가 소모되고 블록체인 노드는 동일한 코드를 실행하여 동일한 데이터를 계산하는데 더 많은 시간을 낭비합니다. 또한 블록체인에 저장된 데이터에는 실제 비용이 거의 들지 않습니다. 이더리움 또는 Qtum중 하나에서 블록체인의 크기를 직접 증가시키지 않습니다. 실제 비용은 블록체인의 크기를 직접 증가시키기 때문에 계약서로 전송되는 데이터 형태로 블록체인에 입력되는 데이터입니다. 이더리움에서 계약(20,000)의 32 바이트를 저장하는데 드는 비용보다 거래내역(23176 가스) 형태로 블록체인에 32

바이트의 데이터를 입력하는 것이 훨씬 저렴합니다. 그리고 64바이트의 데이터(저장용 80,000 가스에 비해 tx의 경우 29704 가스)를 확장 할때 비용도 저렴해집니다. 계약서에 저장된 "가상"비용이 있지만 대부분의 사람들이 생각하는 것보다는 훨씬 적게 발생합니다. 기본적으로 전체 블록체인에 대한 데이터를 저장하는 데이터베이스를 반복하는데 드는 비용입니다. Qtum과 이더리움 모드에서 사용되는 RLP와 LevelDB 데이터베이스 시스템은 이를 효율적으로 처리하지만, 지속적인 비용은 선형적이 않습니다.

비효율적인 코드를 권장하는 EVM의 또 다른 부분은 스마트 계약에서 특정 기능을 호출 할 수 없다는 것입니다. 보안을 위해서 ERC20 계약에서 `withdraw()`와 같은 함수를 직접 호출 할 수 있는 것은 좋지 않습니다. 그러나 효율적인 방향을 위해서 표준 라이브러리가 필요합니다. 단순히 외부 계약에서 코드의 특정 부분을 불러오는 것이 아니라 모든 것이든 아니든 간에 실행은 항상 코드의 첫 번째 바이트에서 시작되므로 모든 Solidity ABI 부트 스트랩 코드를 건너 뛰거나 지나칠 수 있는 방법은 없습니다. 따라서 결국에는 작은 함수가 복제되도록(외부 호출은 비용이 많이 들기 때문) 가능한 많은 함수를 계약에 배포해야 합니다. 100바이트 계약 또는 10,000바이트 계약을 호출 할때 모든 코드를 메모리에 불러와야 할 필요도 없고 비용 차이도 없습니다.

그리고 마지막으로 계약의 저장공간에 직접 접근하는 것은 불가능합니다. 계약 코드는 디스크에서 완전히 불러와야 하고 실행 되어야 하며 코드는 요청한 저장소에서 데이터를 불러와야 하며 마지막으로 가변 크기 배열을 사용하지 않도록 하면서 호출 계약에 반환해야 합니다. 아, 그리고 필요한 정확한 데이터를 모르기 때문에 약간의 전후 시간이 필요하다면 적어도 캐시에 있기때문에 노드에 비해 저렴합니다. 그러나 외부 계약을 또 호출 하는 것에 대해서 가스 가격의 할인은 없습니다. 코드를 완전히 불러드릴 필요 없이 외부 계약의 저장공간에 접근할 수 있습니다. 실제로 계산상 현재 계약의 저장공간에 접근하는 것만큼이나 저렴한데 왜 그렇게 비싸게 만들고 효율성이 떨어 질까요?

디버깅 및 테스트 가능성 부족

이 문제는 EVM설계의 결함뿐만 아니라 구현상에 문제도 있습니다. 물론 일부 프로젝트는 Truffle과 같이 가능한 쉽게 만들려고 노력하고 있습니다. 그러나 EVM의 디자인으로써 이 모든 것을 쉽게 만들 수 없습니다. 현재 사용할 수 있는 유일한 예외 상황은 "OutOfGas"입니다. 로깅 기능이 없고 외부 네이티브 코드(테스트용 소프트웨어 및 모의 데이터와 같은)를 호출하기 위한 쉬운 방법이 없으며 이더리움 블록체인 자체도 사설 테스트 네트워크를 만드는 것이 어렵습니다. 사설 블럭체인은 다른 배개 변수와 서로 다른 동작성을 가지고 있습니다. Qtum은 "regtest"모드 덕분에 적어도 이 부분에 대해서 도움이 되지만 모의 데이터등으로 EVM을 테스트 하는 것은 실제로 독립형이 아니기 때문에 구현이 여전히 어렵습니다. Solidity 레벨에서 이 작업에 대해 알려진 디버거가 없습니다. 적어도 하나의 EVM 어셈블리 디버거가 있지만 전혀 사용자 친화적인 것이 아닙니다. EVM 과/혹은 Solidity에 대해 설정된 심볼 형식이나 디버그 데이터 형식이 없으며 DWARF와 같은 표준화 된 디버그 형식으로 작업하기 위한 EIP 또는 기타 활동을 찾지 못했습니다.

부동 소수점 숫자

보둥 소수점 지원이 부족할 때 사람들이 말하는 공통점은 "아무도 부동 소수점 숫자를 사용해서 통화값 처리 하지 않습니다" 입니다. 이 말은 매우 시야가 좁은 표현입니다. 위험 모델링, 과학 계산 및 실제 값보다 범위와 근사값을 더중요하게 여기는 분야와 같이 부동 소수점 숫자에 대해 많은 실제 사용 사례들이 있습니다. 스마트 계약의 잠재적인 응용프로그램을 통화 가치만 추가 한다라고 말하는 것은 매우 비현실적이며 불필요한 제약사항을 동반합니다.

변경 불가 코드

계약서의 주요 변경 사항중 하나는 업그레이드 가능성입니다. 왜냐하면 계약이 바뀌어야 할 필요가 없기 때문입니다. EVM 코드는 완전히 변경되지 않으며, Harvard Architecture of computing을 사용하기 때문에 코드를 메모리에 불러드린 이후 실행 할 수 없습니다. 코드와 데이터는 완전히 다르게 다루어 집니다. 따라서 계약 업그레이드를 위한 유일한 선택사항은 완전히 새로운 계약을 작성해서 모든 코드를 복제하고 이전 계약을 해당 계약으로 방향 전환 하는 것입니다. 계약을 부분적(또는 전체적으로)으로 보수하고 대체하는 것은 불가능합니다.

결론

저는 중대한 일을 끝냈고 저의 호언 장담이 끝나고 있다 생각합니다. 이 시점에서 EVM은 필요악입니다. 그것은 이 영역에서 처음겪는 일이고, 자바 스크립트와 같이 가장 먼저 나오는 것들 처럼 문제가 많이 있습니다. 그리고 설계가 상당히 독특하고, EVM에 이식된 프로그래밍 언어라고 볼 수 없습니다. 이 디자인은 지난 50년 이상에 걸쳐 설립된 많은 공통 언어 패러다임에 상당히 적대적입니다. 여기에는 `JUMPDEST`와 같은 점프 데이블 최적화, tail-recursion 미지원, 이상하고 유연하지 않은 메모리 모델, 이해하기 어려운 외부 코드에 대한 `DELEGATECALL` 모델, 비트 시프트와 같은 일반적으로 사용되는 opcode 부족, 유연한 스택 크기 제한 등이 포함 됩니다. 물론 256비트 정수도 같이 포함됩니다. 이러한 측면은 EVM에 대한 전통적인 언어 이식을 매우 비효율적이며 최악의 경우 불가능하게 만듭니다. 이것은 제가 EVM 언어가 현재 EVM을 위해 특별히 제작된 것이며 모든 것이 독특한 모델을 염두해두고 있는 이유입니다. 지금은 정말로 슬픈 상태입니다.

저는 이 게시물 전체가 EVM에 대한 공격이나 EVM설계자에게 아무런 의미가 없다는 것을 의미하는 바가 아닙니다. 그것은 단지 어떻게 작동하는지에 대한 이야기입니다. 소 잃고 외양간 고치기 이지만 EVM 디자인의 측면에 대해 많은 후회를 하고 있습니다. 저는 그들을 공격하고 싶지 않습니다. 그러나 오히려 이런 결함들은 더 큰 블록체인 개발자 커뮤니티에 가져와서 소모적인 논쟁이 되지 않기 바랍니다. 또한 동시에 "Solidity를 할 수 없는 이유는 무엇 입니까?" 라는 질문에 영감을 받습니다. EVM은 여전히 우리가 이익과 손실을 배우고 있는 놀라운 디자인을 가지고

있으며 스마트 계약이 효율적이고 강력해질수 있기 전에 오랜 시간이 필요하다는 것은 명백합니다. EVM은 이분야에서 첫번째 경쟁자였으며 궁극적으로 스마트 계약은 모든 사용 예제를 학습하고 어떤 디자인이 가장 큰 이점이 되는지를 발견했습니다. 우리는 먼 길을 왔지만 아직 갈길이 멉니다.