Class 12: Script

Schedule

Read: *Chapter 3: Mechanics of Bitcoin*, from Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies*. Sections 3.2 and 3.3 are about bitcoin scripts, and should be read carefully. (You should read the whole chapter, but those sections are most relevant to today's class.)

Friday, October 9: Problem Set 2 (8:29pm).

Monday, October 19: Midterm Exam

Hash Collisions!

Computing a bitcoin address: (bitcoinwiki)

Private Key: pick a random number, *k*.
Public Key: compute (*Ux*, *Uy*) = *Gk* (elliptic curve multiplication, *G* is specified generator point)
Ux and *Uy* are 32 bytes each.

The bitcoin address is (|| is bitstring concatenation):
raw = 1 || RIPEMD160(SHA256(*Ux* || *Uy*)) *RIPEMD output is 160 bits (20 bytes) + one byte for 1*
checksum = first 4 bytes of SHA256(SHA256(raw)) *Compute a checksum using SHA256 double hash*
address = Base58Check(raw || checksum) *convert to printable, unambiguous characters*

How important is pre-image resistance for the security of bitcoin addresses?

How important is collision resistance for the security of bitcoin addresses?

How important is pre-image resistance for the integrity of bitcoin's proof-of-work? What about collision resistance?

Xiaoyun Wang and Hongbo Yu, *How to Break MD5 and Other Hash Functions*, EuroCrypt 2005.

Bitcoin Script

Transaction outputs in bitcoin are protected by *locking scripts*, and must be unlocked by *unlocking scripts*. The scripts are written in a simple (compared to, say, the Java Virtual Machine language, but quite complex and poorly specified for what one might expect would be needed for bitcoin transactions) stack-based language. A transaction output is not unlocked unless an unlocking script is provided such that the result of executing the unlocking script, followed by executing the locking script, is a stack with value True on top (and no invalid transaction results during the execution).

Opcode	Input	Output	Description
`OP_1`	-	**1**	Pushes a **1** (True) on the stack
`OP_DUP`	*a*	*a a*	Duplicates the top element of the stack
`OP_ADD`	*a b*	(*a*+*b*)	Pushes the sum of the top two elements.
`OP_EQUAL`	*a b*	**0** or **1**	Pushes **1** if the top two elements are exactly equal, otherwise **0**.
`OP_VERIFY`	*a*	-	If *a* is not **True** (**1**), terminates as Invalid.
`OP_RETURN`	-	-	Terminates as Invalid.
`OP_EQUALVERIFY`	*a b*	-	If *a* and *b* are not equal, terminates as Invalid.
`OP_HASH160`	*a*	H(*a*)	Pushes bitcoin address, RIPEMD160(SHA256(*a*)).

Some more complex instructions:

`OP_IF` *statements* `OP_ENDIF` - If the top of the stack is **1**, executes *statements*. Otherwise does nothing.

`OP_CHECKSIG` - Pops two items from the stack, *publickey* and *sig*. Verifies the entire transaction (known from node state, not the stack) using the *publickey* and *sig*. If the signature is valid, push **1**; otherwise, **0**.

The most common locking script (send to public address):
`OP_DUP` `OP_HASH160` `OP_DATA20` *(bitcoin address)* `OP_EQUALVERIFY` `OP_CHECKSIG`

What must be on the stack to unlock this locking script (end with **1** on top of stack)?

What must be on the stack to unlock `OP_HASH160` *20-byte hash* `OP_EQUAL` ("Pay-to-Script-Hash")?

Is the bitcoin scripting language Turing-complete? What would be risky about a crytocurrency scripting language being Turing-complete?

See web notes for links to BTCD and Bitcoin Core Code for interpreting script.