Cryptographic Implementations In Digital Design

Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

Cryptography and Digital Implementations

- Cryptography has long been a "typical" application for digital design
  - A large repetitive calculation repeated over all data
  - Some significant parallelism opportunities (with limits)
  - Situations for dedicated hardware when cost is essential
- Where Hardware is Different
  - High throughput operation & multiple streams of encryption
  - ***True*** Random Number Generators
  - Shielding secrets
- And Why You Should Almost ***Never*** build this stuff

Large Number Arithmetic: Diffie/Hellman

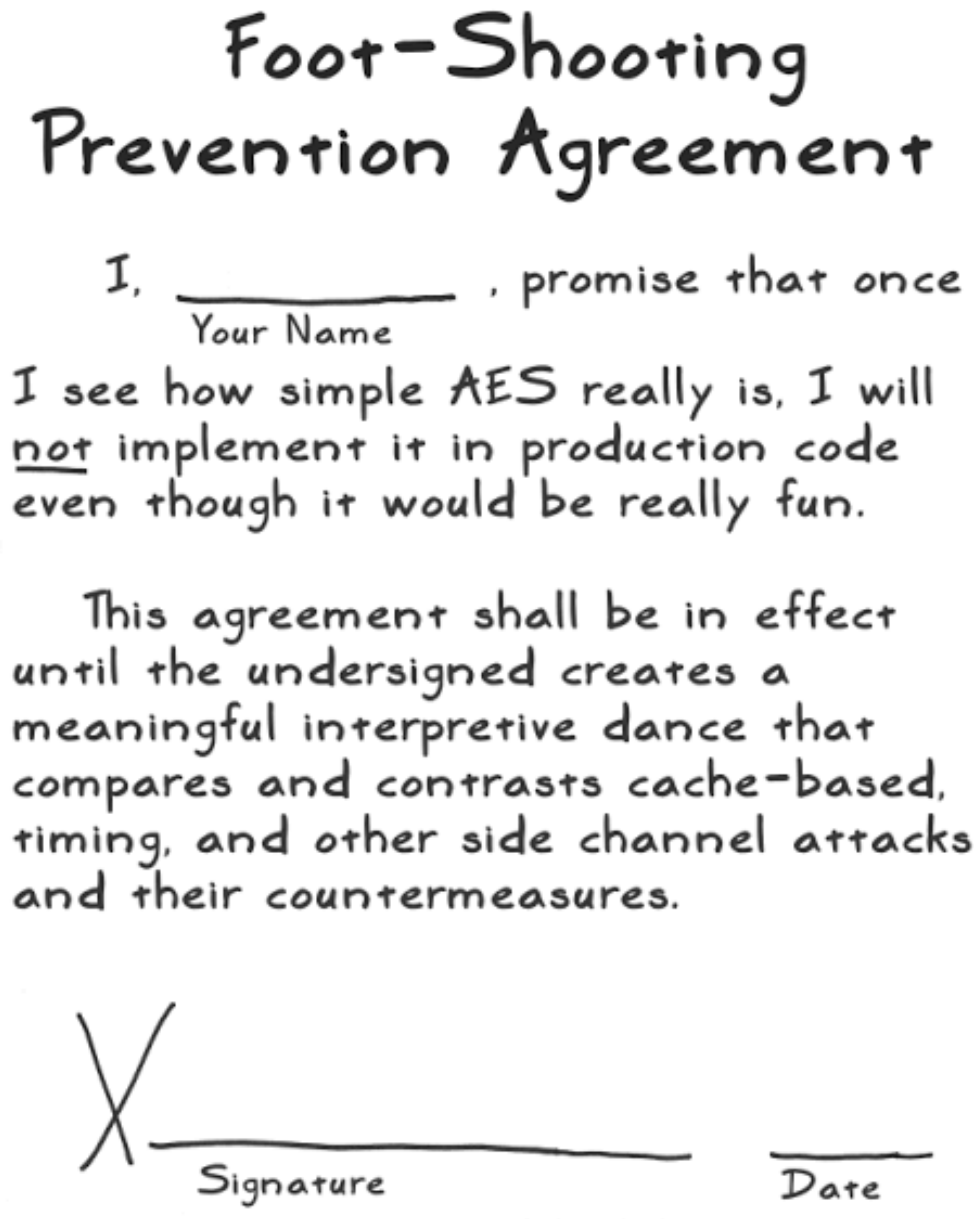
- Diffie/Hellman Key Exchange
  - Public prime ***p***, public generator ***g***
  - Private random variables ***a*** and ***b*** belonging to "Alice" and "Bob"
- Goal is to create a shared random value
  - Alice computes g^a mod p and sends it to Bob
  - Bob computers g^b mod p and sends it to Alice
  - Alice then computes g^{ba} mod p
  - Bob computers g^{ab} mod p
    - Both values are the same
- Similar math for RSA and other public key systems

How Big A Number Are We Talking About Here?

- Not very secure, p, a, b are 1024b
- OK security they are 2048b
- Properly paranoid security: 3072b
- Result is some pretty significant math:
  - Exponentiation by a 3072b exponent modulo a 3072b value
- But these days, software is almost always fast enough
  - Vector/SIMD instructions can be used to greatly speed up the multiplication
  - Mostly only used for key exchange or signatures: ***not per data elements***

Real Use:
Bulk Encryption

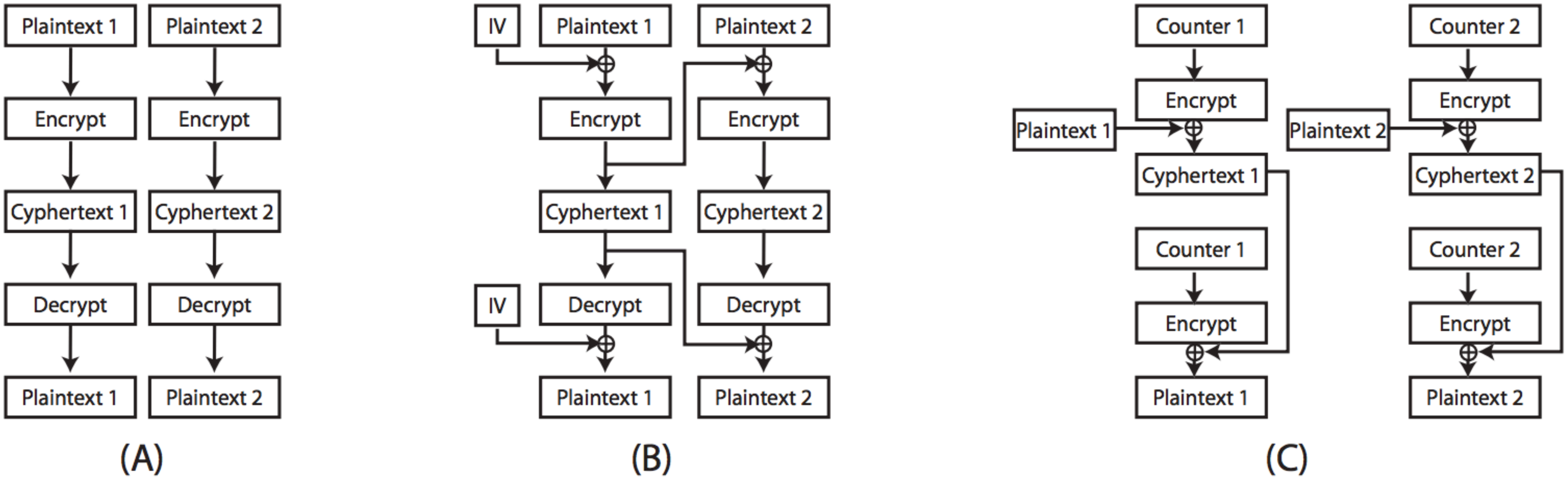
- Block ciphers. Most block ciphers consist of:
  - Small/medium table lookups
  - XORs
  - Shifts and rotates
  - Same for hash functions as well
- Most common algorithm is AES
  - A more detailed description here: http://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html



Basic Concept

- Block cipher accepts a fixed amount of data (block) & key
  - for AES == 128b block, 128b, 192b, or 256b key
- It acts as a ***keyed permutation***, creating a block sized output
  - There is also an inversion function which can accept this block of data and the key and recreate the original input
- Used in an encryption mode
  - Lots more details in CS161, but...
  - The best encryption modes take the output of the previous block encryption when encrypting the next block

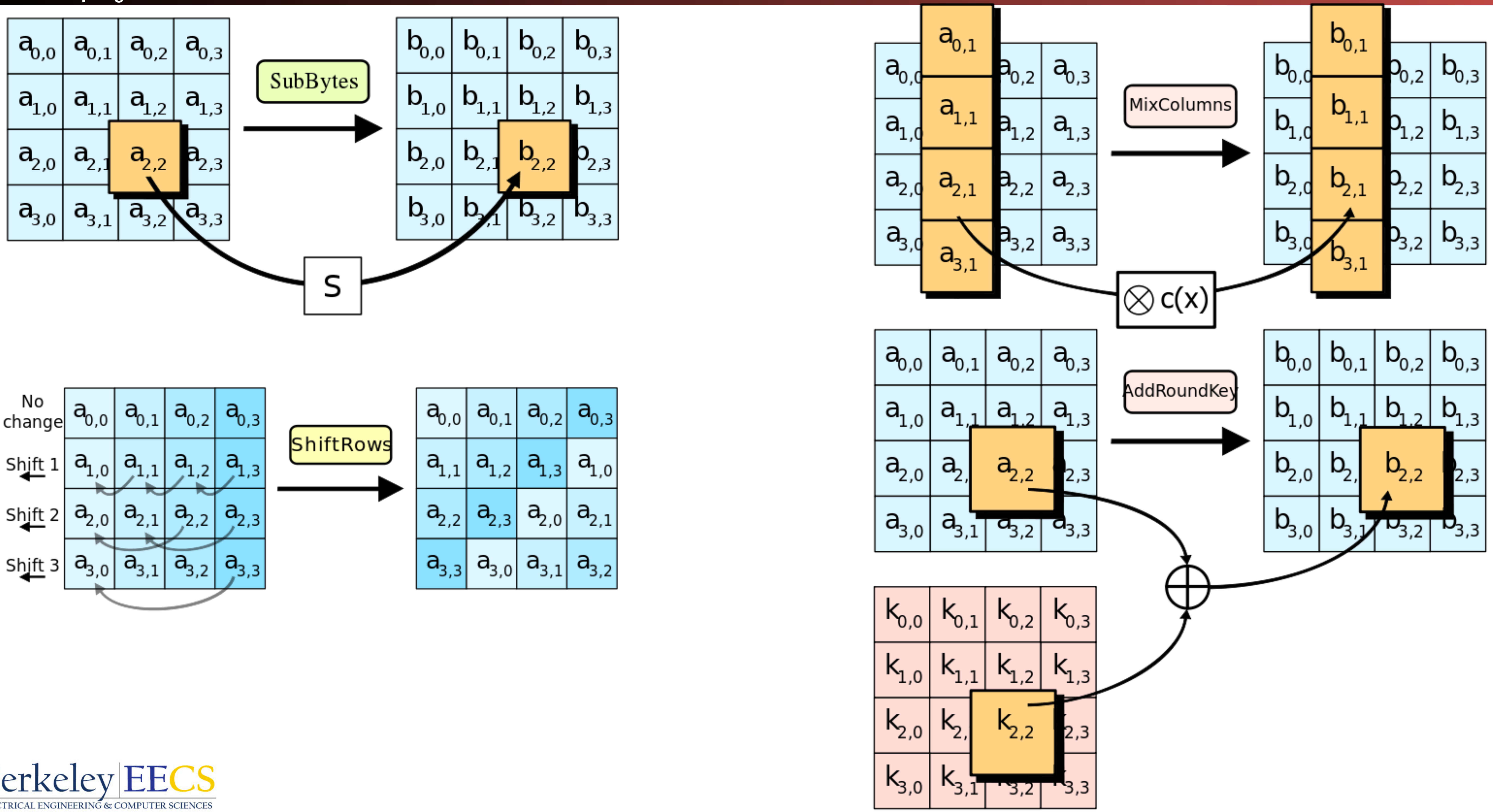
Some Common Encryption Modes



How AES works...

- Treats data as a 4x4 array of 8b quantities
- Key expansion
  - Take the initial key and create a different "subkey" for each round
- At the start: AddRoundKey
  - Just xor the data with the key
- Then 10 rounds (for 128b key)
  - SubBytes: an 8b->8b S-Box operation for each word
  - ShiftRows: a rotation within the array {last round omits this step}
  - MixColumns: a bit-oriented mixing of all the input in a column
    - Some funky-galios math stuff, but can generally be implemented as 4-LUTs and XORs
  - AddRountKey: again an xor

Visually:


SubBytes
S
No change
Shift 1
Shift 2
Shift 3
ShiftRows
MixColumns
⊗ c(x)
AddRoundKey

Comments:

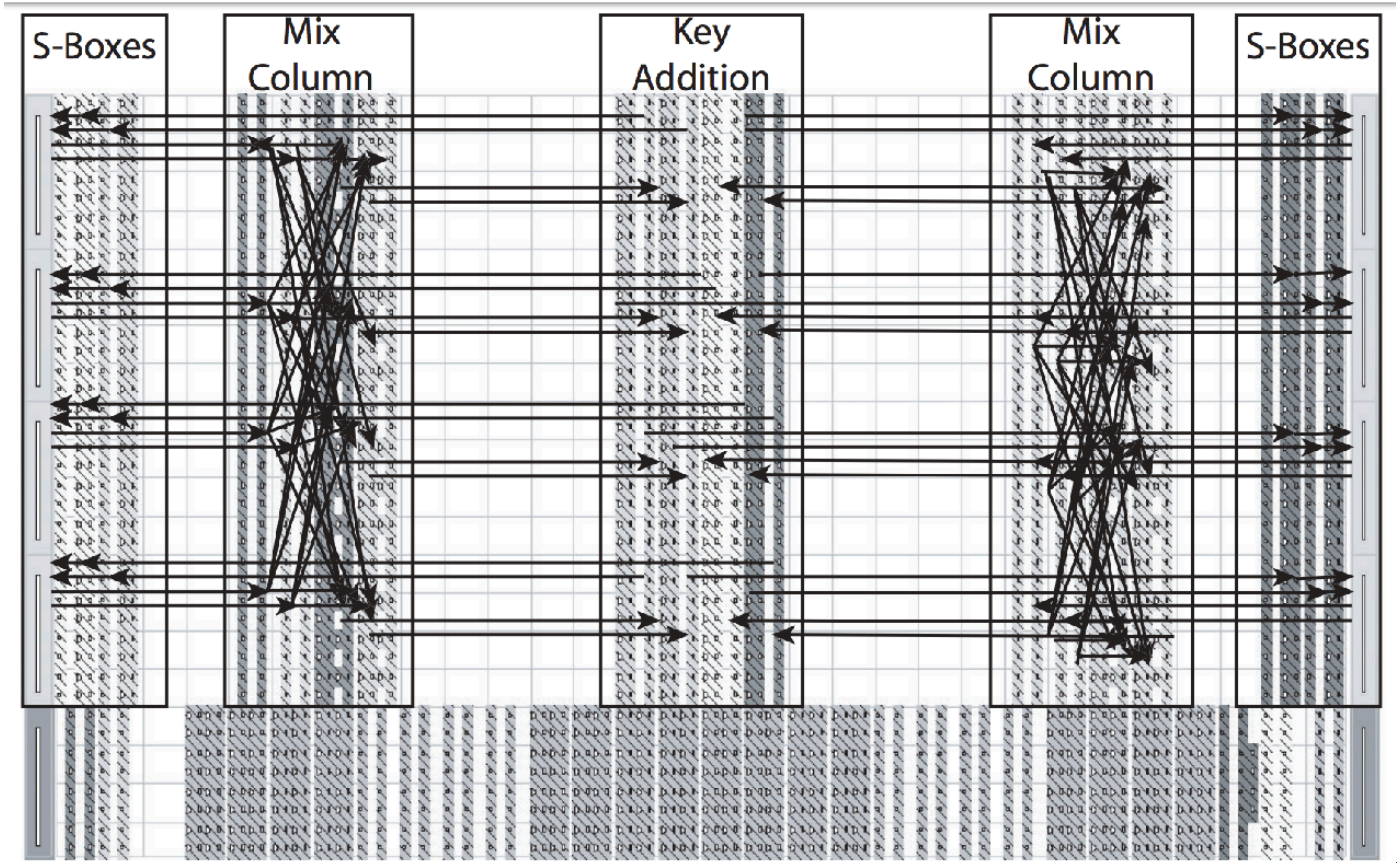
- This is ***really really*** good for hardware
  - XORs are great
  - 8b table lookups map to pretty small ROMs
    - 8 BlockRAMs for a round + 2 for the key expansion
  - MixColumn is designed to map well to small logic gates
    - Since it is a couple of easy galios multiplies (which are bit-twiddling) and then summing things up (which are XORs)
- Great target for C-slow designs
  - Build one round, pipeline it aggressively:
    Now can be working on C separate blocks at the same time
- Beyond that, for more throughput, just replicate...
- But often latency limited by feedback loops, so just build 1 round w/o pipelining

How To Implement...

- If concerned about latency
  - EG, because you are running in a feedback mode
- Implement a single round logic
  - Every clock cycle it computes exactly 1 round
- This is effectively optimal
  - You could "unroll" and do multiple rounds, but you'd only save the setup & hold-time of the flip-flops for a huge cost in area
- If concerned about throughput
  - EG, counter mode (don't do it!), or encrypting multiple streams
  - Just pipeline the hell out of the single round in a C-slow manner...
  - And beyond that, just ***replicate*** the entire unit

Best FPGA AES Implementation circa 2003: Spartan-II 100 based

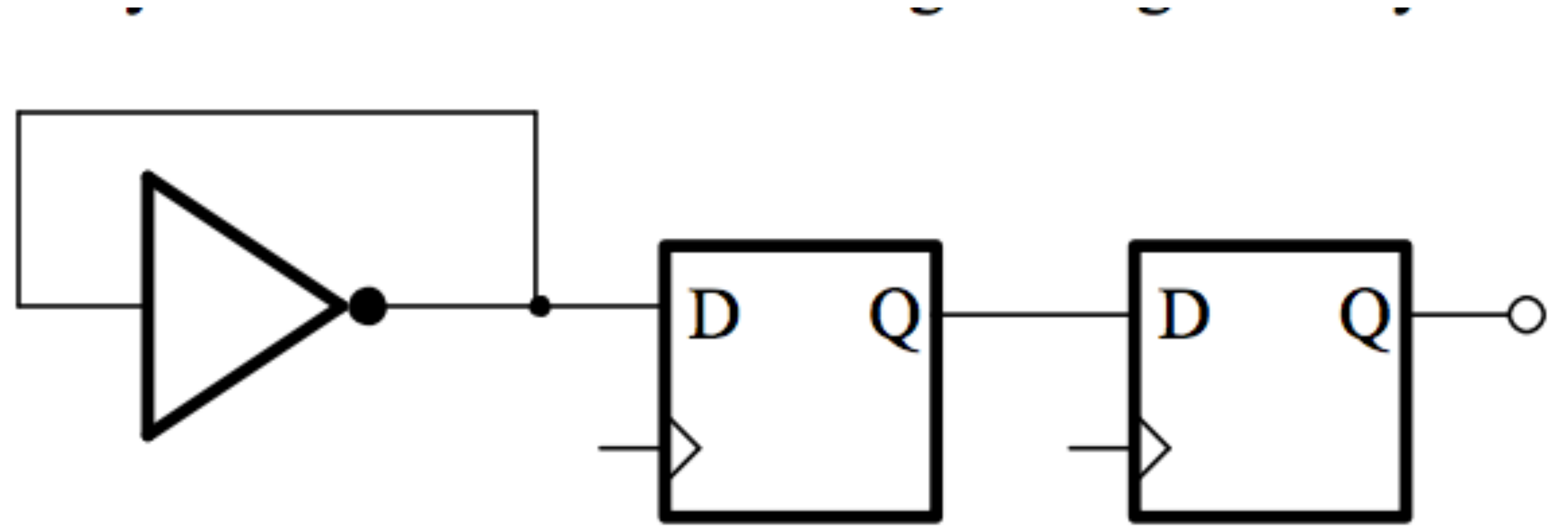
- "Key Agile":
  - Accept key and data, calculate the key generation
- 5-stage C-slow
  - 5 ***independent*** encryptions
- 10 BlockRAMs, 780 slices (2 LUTs in each slice)
  - 1.3 Gbps, 115 MHz
  - Unpipelined still 500 Mbps



But the REAL special in hardware: Random Numbers & Keeping Secrets...

- Cryptography uses random numbers ***all the time***
  - And if they can ***ever*** be predicted by an adversary, you lose!
- Software ***sucks*** for generating random numbers...
  - You need true physical randomness to "seed" the random number generator
- But pseudo-random-number-generators are good, ***if seeded properly***
  - Can flip a heavily biased coin (90% heads) a lot, feed that into a pRNG, and get good random numbers out

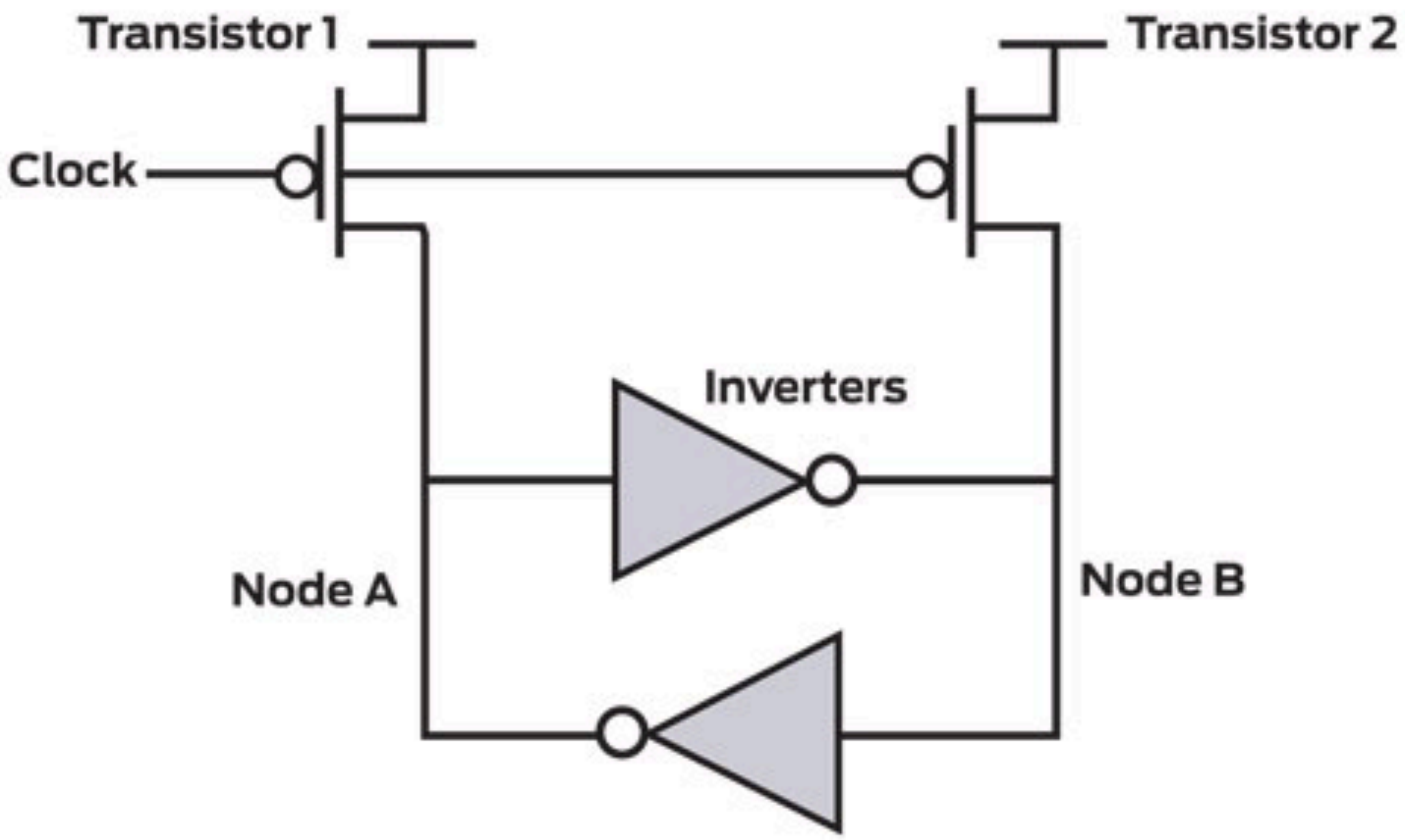
Possibility #1: Ring Occilators...

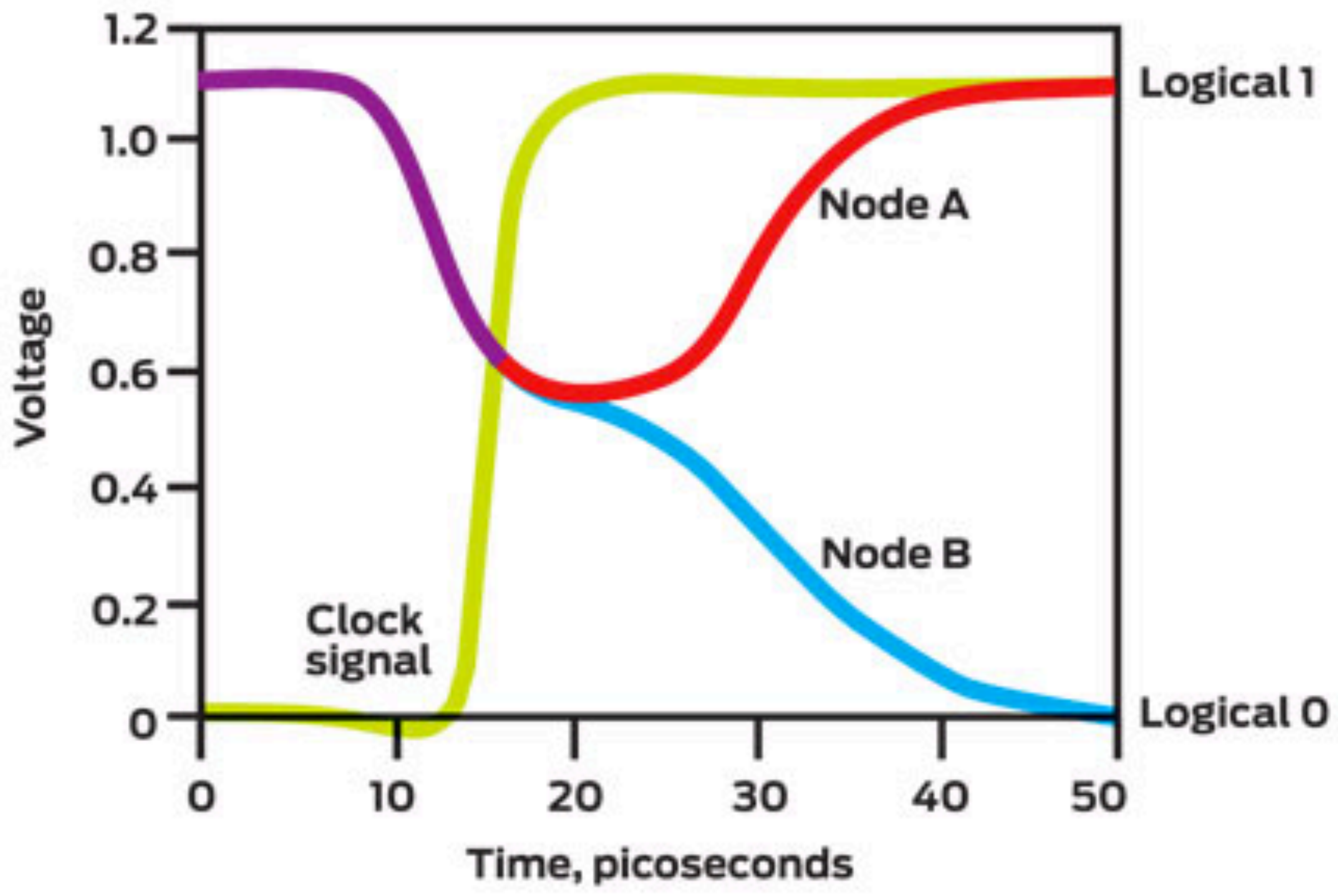


- An inverter tied to itself is an occilator...
  - But not that stable, it has jitter that is affected by temperature and a whole bunch of other things...
- So have a fast & noisy oscillator
  - And sample it with a slow clock
- Result is a good but ***biased*** random number generator
  - Its based on physical noise, but not all the bits are truly independent.
- Can be built in FPGA logic!

Possibility #2 (Intel): Use metastability and watch it fall

- Idea: nudge a latch into a metastable state
  - Then let it fall to a 0 or 1





Intel's Tweaks...

- They don't want the coin to be too biased
  - (IMO, somewhat overkill, even .1b of actual entropy works when continually mixed into a secure pRNG)
- So they add a balancing circuit underneath
  - Adjusts the available capacitance on the two sides of the nodes
  - Keep track of several flips, use that to shift the bias function

And from there...

- Feed into a cryptographically secure psudo-random-number generator (also called a DRBG)
  - Intel uses AES encryption for counter mode DRBG: Mix in the new entropy into the key...
  - Output of the DRBG fed into the instruction
- And that is just "ordinary" software for CS161 type stuff...

The Other Big Use: Holding Secrets...

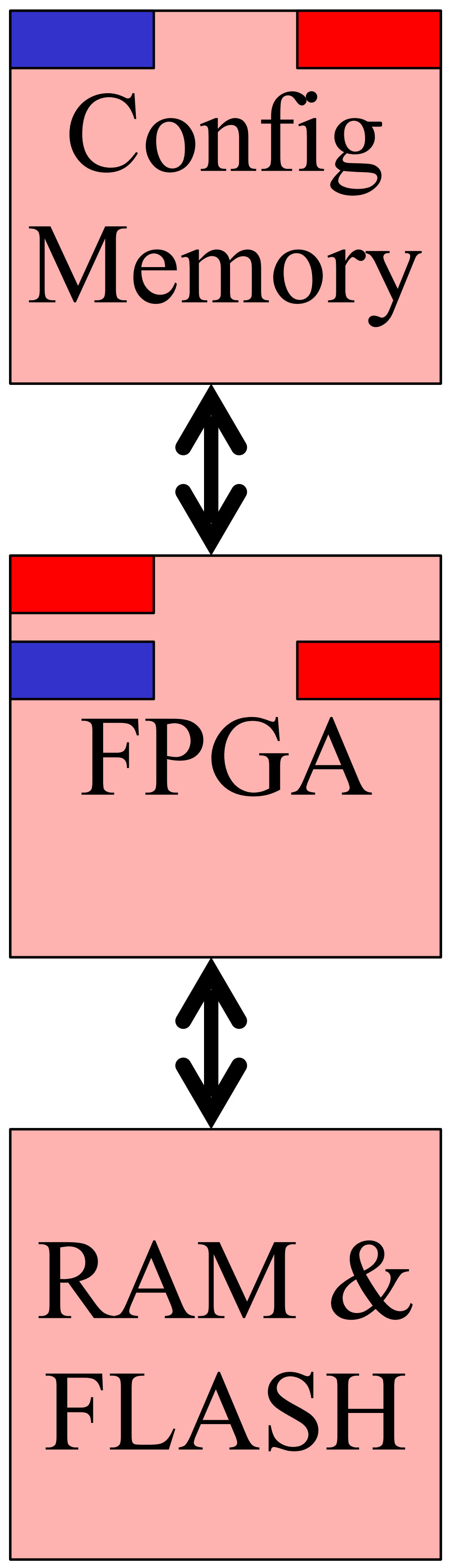
- Have a small amount of data that ***never*** leaves the chip
  - Either battery-backed SRAM cells
  - Or programmable memory that is programmed during the manufacturer
- This data can be a random cryptographic key for everything else
  - So you can protect the entire system: Unless someone can get the secret
- How Apple Does it (on Whiteboard)
- How I'd Do it w Xilinx:
  A paper design to protect design secrecy & integrity

The key: Bitfile Encryption

- Current FPGAs support bitfile encryption
  - A secret key is stored in the FPGA
    - In static memory in the Altera Stratix series
    - In SRAM in the Xilinx series, with a separate V_{batt} input
      - Will assume the Xilinx technique for now, its more powerful
- The bitfile is stored off chip in an encrypted form
  - When the FPGA first loads, it decrypts the bitfile using the encryption key as it is read into the configuration
    - 256b AES in current designs
  - The configuration is used to set the circuit function inside the FPGA
- The keys and decrypted configuration only exist within the FPGA
  - To determine the configuration, need to break the FPGA encryption
    - Easiest is probably to extract the key stored in the FPGA
- Designed to prevent piracy by providing circuit secrecy
  - Without circuit secrecy, FPGA piracy is trivial
  - With circuit secrecy, it is impossible

Leveraging Bitfile Encryption

- On *first* boot, in a ***controlled environment***
  - FPGA is given initial unencrypted configuration
    - Configuration includes an ***Authorizer key***
      - Could be just a public key, or a secret key
  - FPGA generates an internal random secret ***Device Key***
  - FPGA loads the ***Device Key*** into the bitfile decryptor's storage
  - FPGA rewrites the configuration
    - Inserting the ***Device Key***
    - Encrypting it with the ***Device Key***
  - Can also create additional key material at this time
    - Such as a public key for device authentication
- All subsequent loads are protected by the device key
  - Device key is also used to encrypt optional off-chip memory
    - Secure persistent storage
  - Device key can also present a unique public key



Circuit Secrecy

- The design in the FPGA is now protected by the bitfile encryptor
  - Outside of the FPGA, the design is ***always*** encrypted with a key ***unique to the specific FPGA***
  - The cleartext key ***NEVER*** leaves the FPGA once programmed
    - And is stored in volatile memory
  - Within the running FPGA, the design is decrypted internally and stored as distributed SRAM cells
  - ***All*** off-chip memory is encrypted
    - Provides encrypted storage
- Protection equivalent to the anti-piracy mechanism
  - Anti-piracy is all about maintaining circuit secrecy
  - Need to either extract the bitfile from SRAM from the running FPGA
  - Or extract the bitfile key from the FPGA's key storage
  - Or perform a side channel attack on the bitfile loader
  - Or bribe an engineer to give you the design…

Tamper Resistance

- Attacker ***CAN*** run his own design in a stolen device
  - As she can always just overwrite/erase the configuration key stored in the FPGA and load the design of her choice
- But if the attacker can't modify the original bitfile (break the circuit secrecy), then the entire system can be ***tamper evident***
  - The configuration can also contains a unique public/private key pair for the device as well as the ***Device Key***
    - Device can now authenticate that it is running a valid bitfile to everyone else
  - Attacker's design can't access storage (its encrypted with the ***Device Key***) or ***any*** external resources which require authentication
- Only slightly less powerful than tamper resistance
  - But not by much, as the attacker ***still*** has to do her own design from scratch, so we can still probably call it fully Tamper Resistant

Activation and Updates

- Present a new bitfile, signed (or encrypted by) the ***Authorizer key***
  - Device authenticates that the new bitfile is valid
    - Pick your authorization/delegation scheme
  - Device decrypts the new bitfile internally, and reencrypts the bitfile using the ***Device Key***
    - At this time, the new design is modified to include a copy of the ***Device Key***
    - Unencrypted design never leaves the FPGA
  - New bitfile is written out to configuration storage
- New design still contains the basic primitive blocks
  - Needed so further activation and updating can occur
  - So requires a persistent IP core across all designs
    - Engineering effort to design: best solution is probably to store all keys in a fixed BlockRAM on the FPGA
- Thus ***ONLY*** authorized updates are allowed, and are semantically equivalent to activation
  - No limit on the number of upgrades or activations

Revocation

- If in communication with the device, or after a specified time, we wish to remove some functionality…
- Simply have the device overwrite/destroy the configuration state for the revoked design
  - Need to overwrite the whole data, to prevent a key compromise from recovering the revoked design
  - Need to include the notion of time in activation, to prevent reactivation of a revoked design
    - Perhaps also include a check in the persistent storage, so design could never be reactivated
- Revoke the device completely
  - Overwrite the key storage and all designs stop working
    - But overwrite the configuration storage ***anyway***
  - "Bricks" the system completely until it can be reprogrammed again in the secure environment

But Why This Is Don't Try This At Home: Side Channels

- There are lots of ways to attack a cryptosystem...
  - And almost ***none*** of them involve breaking the cryptography!
- Power consumption
  - Directly indicates what bits are being encrypted
- Timing
  - How long operations take. You ***can not*** optimize crypto systems in some ways
- Fault injection...
  - Deliberately cause a hardware device in hand to screw up!

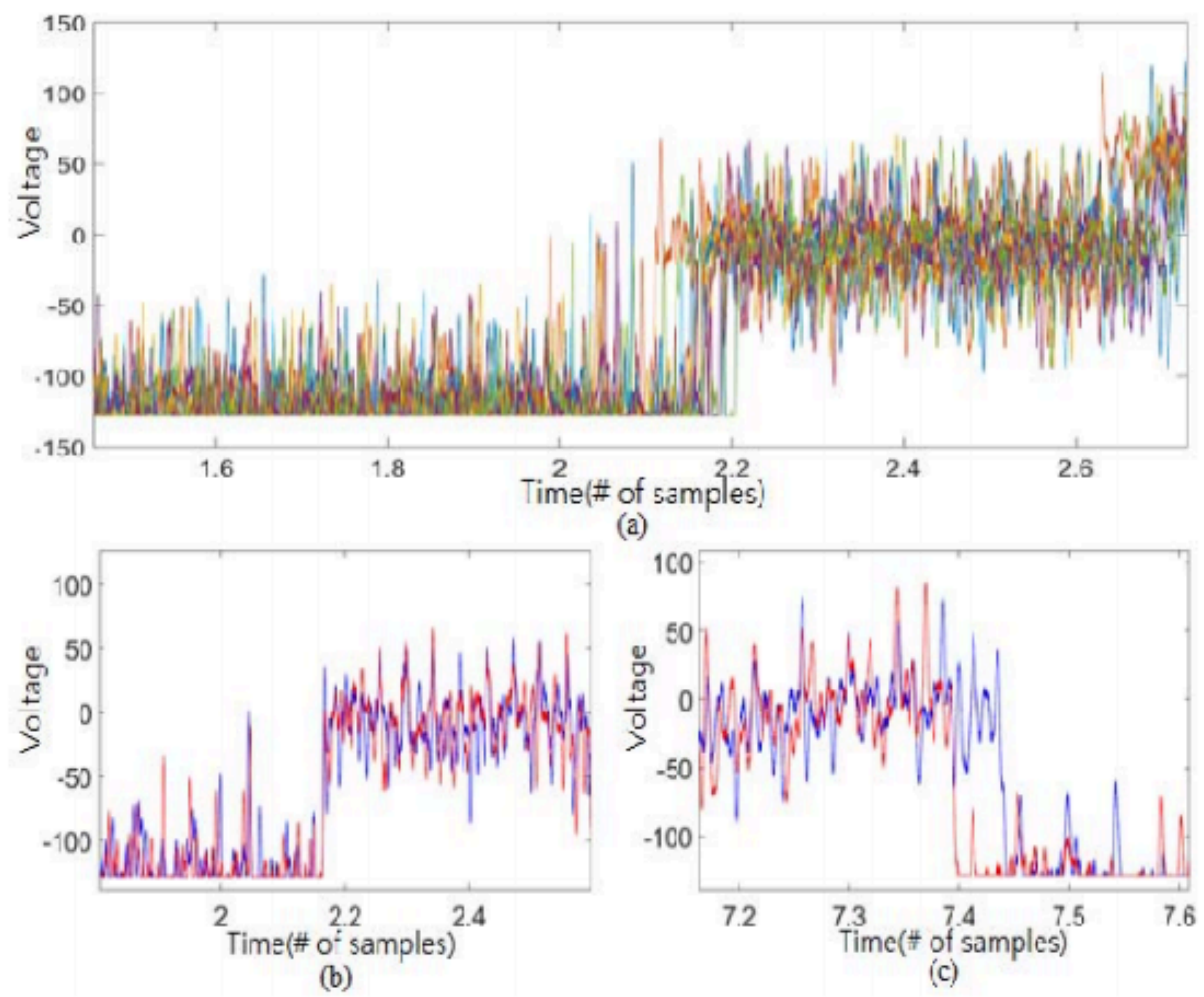


Figure 9. Countermeasure effects in the measurements

https://www.blackhat.com/docs/asia-17/materials/asia-17-Kim-Breaking-Korea-Transit-Card-With-Side-Channel-Attack-Unauthorized-Recharging-wp.pdf