Implementation of MITM Attack on HDCP-Secured Links

bunnie / 28c3

Twitter your comments @bunniestudios or #28c3

What is HDCP?

- High Definition Content Protection
  - Pixel-level encryption operating at the link layer
- Cipher structure
  - Stream cipher capable of generating 24 bits of pseudorandom data per clock cycle
    - Two parallel 84-bit block functions per round
    - LFSR-based “key scheduler” that whitens block functions at the beginning of each horizontal line of pixels
    - Block functions initialized with publicly exchanged 64-bit initial vector (A_n) that evolves once during each vertical blanking interval

What is HDCP?

- Key management
  - Distributed private keys with sort of key revocation
  - Public key is a “key selection vector” (KSV)
    - 40 bits (20 zeros and 20 ones)
  - Private key is a vector of 40 56-bit numbers
  - All private keys derived from a master key consisting of a 40x40 matrix of 56-bit numbers
- Master key can be directly computed from a collection of 40 unique private keys
  - The master key was revealed in September 2010

Why HDCP?

- Encrypt video transmissions
  - Complements AACS, BD+ to create studio-to-screen cryptographic chain
- Chain was broken long ago: AACS was the weakest link
  - HDCP master key leak is thus largely a “nop” from the content access standpoint
  - Strippers based on legitimate HDCP keys have long been available on the market; key revocation is largely ineffective

So Why Implement HDCP MITM?

- It's about control
  - Broadcasters and studios control your screen
  - DMCA and other legal tricks make it illegal for you to modify content – on *your own screen*

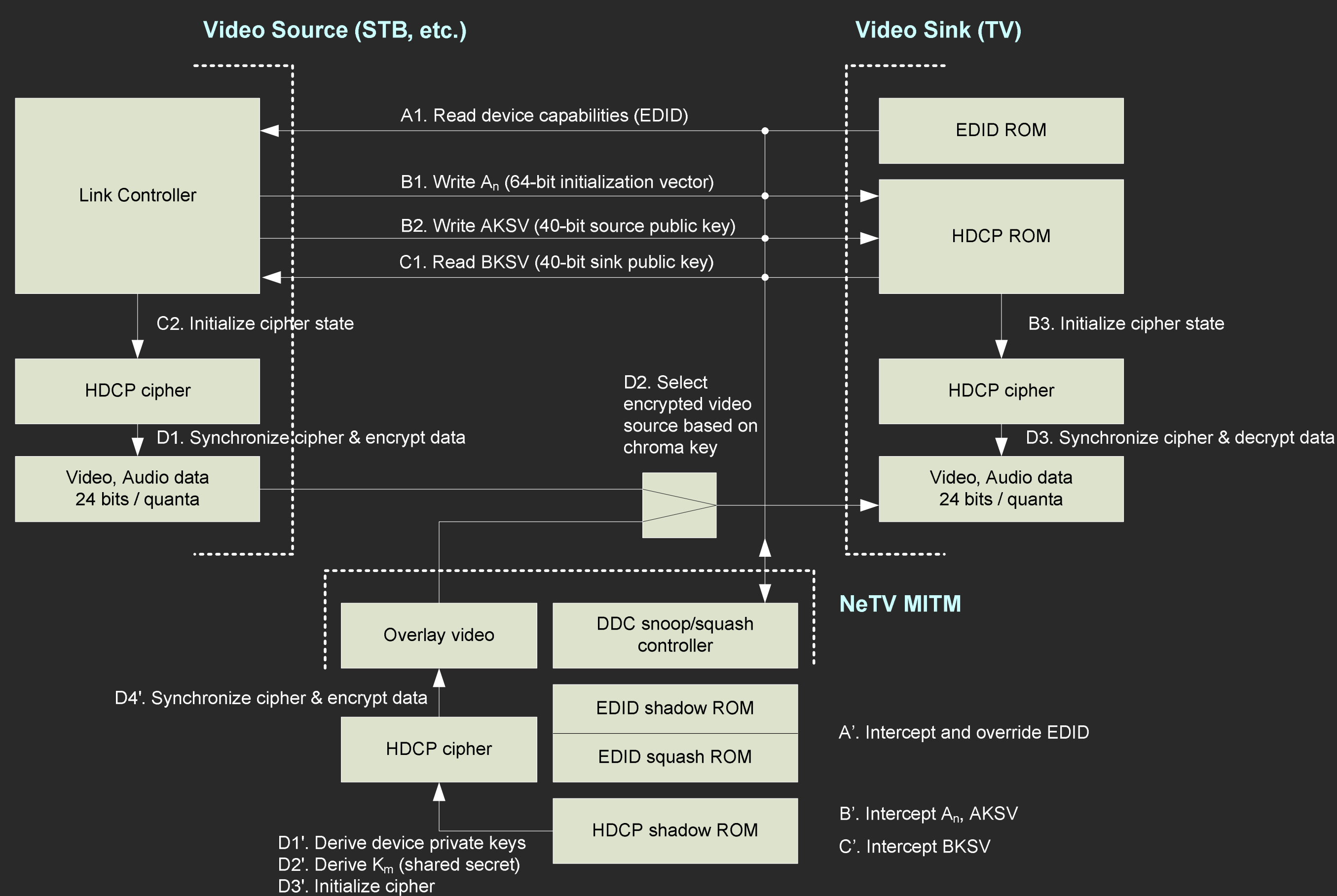
So Why Implement HDCP MITM?

- HDCP restricts the implementation of legitimate content manipulation
  - Picture in picture
  - Content overlays
  - 3^{rd} party filtering & image modification
- As a result, there are few HDMI video mixing solutions that can operate on broadcast/movie content

Goal

- Consumer-side content remixing
  - Add web content to existing TV
  - Live comment & chat
- “Over the top” advertising
  - Eliminate ads
  - Or replace ads with targeted ads
- Interactive TV
  - Add interactive elements to broadcast TV
- Compatibility with any TV

How Do We Do It?



Video Source (STB, etc.)
Video Sink (TV)
Link Controller
EDID ROM
HDCP ROM
A1. Read device capabilities (EDID)
B1. Write A_n (64-bit initialization vector)
B2. Write AKSV (40-bit source public key)
C1. Read BKSV (40-bit sink public key)
C2. Initialize cipher state
B3. Initialize cipher state
HDCP cipher
HDCP cipher
D1. Synchronize cipher & encrypt data
D2. Select encrypted video source based on chroma key
D3. Synchronize cipher & decrypt data
Video, Audio data 24 bits / quanta
Video, Audio data 24 bits / quanta
NeTV MITM
Overlay video
DDC snoop/squash controller
D4'. Synchronize cipher & encrypt data
HDCP cipher
EDID shadow ROM
EDID squash ROM
A'. Intercept and override EDID
HDCP shadow ROM
B'. Intercept A_n, AKSV
C'. Intercept BKSV
D1'. Derive device private keys
D2'. Derive K_m (shared secret)
D3'. Initialize cipher

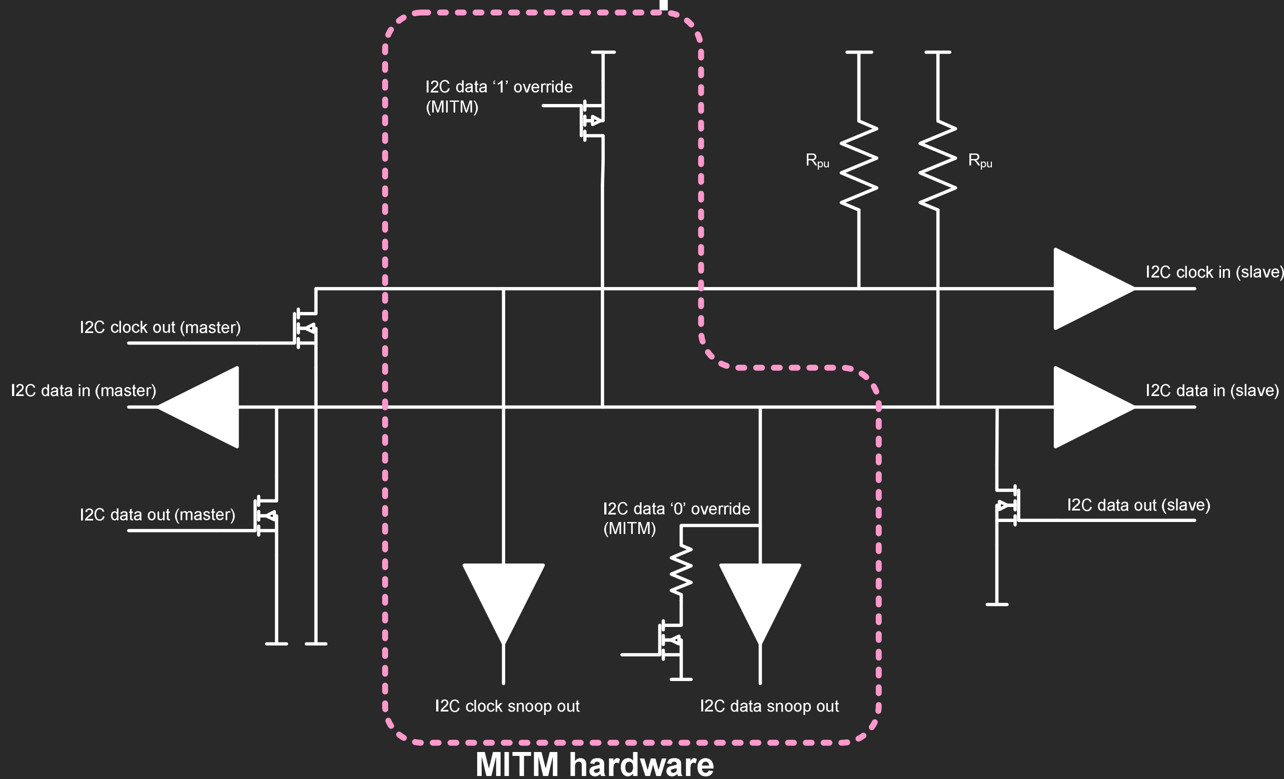
A’: Intercept and override EDID

- HDMI uses an I2C bus (referred to as DDC) to communicate between video source & sink
- Bus shared between two functions:
  - Monitor capability identification
  - HDCP key exchange

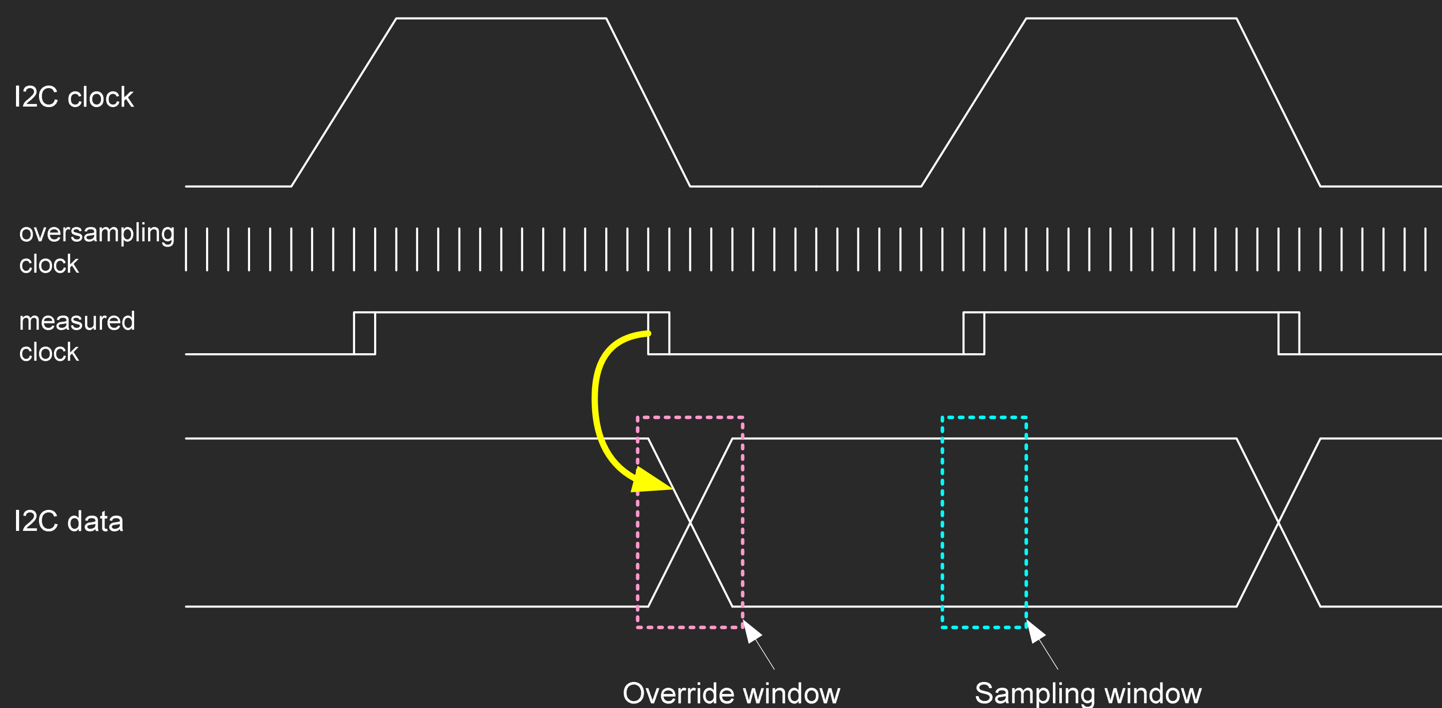
Snoop & squash

- Snooping: intercept key exchange
- Squashing: force TV characteristics
  - The implementation can't do all HDMI standards
  - Rewrite the EDID record on the fly to reflect only the standards NeTV supports, e.g. no 3D, etc.

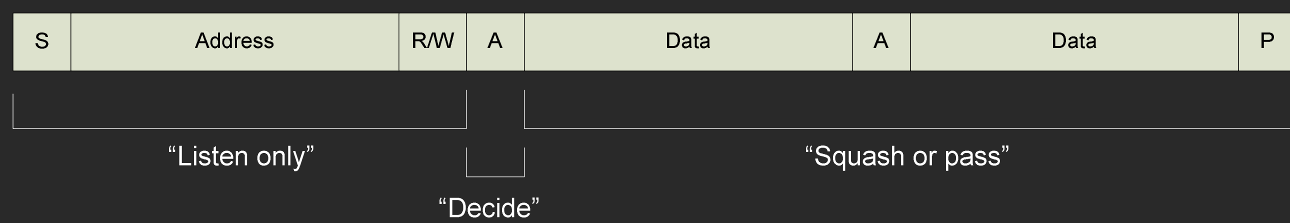
I2C snoop & override



I2C snoop & override



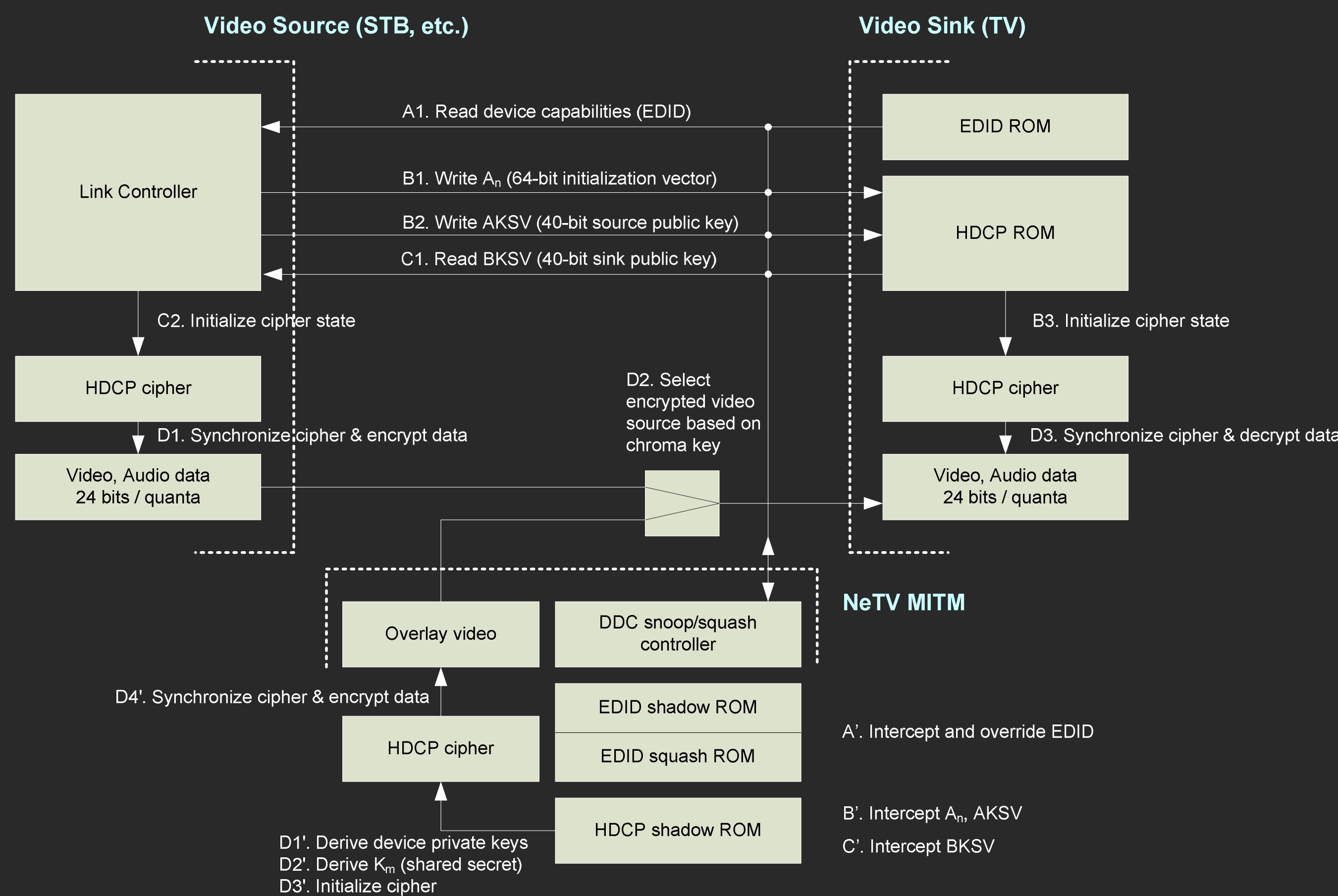
I2C snoop & override



- Oversampled squash can modify data on the fly
  - Snoop address, and change only bits that need changing

Hot Plug Override

- Hot plug bus has a FET on it to simulate a plug/unplug event
  - Hot plug is an open-drain bus, so this is a safe and easy thing to do
  - Used to resynchronize state when necessary
  - Used to manipulate EDID state


Video Source (STB, etc.)
Video Sink (TV)
Link Controller
A1. Read device capabilities (EDID)
B1. Write A_n (64-bit initialization vector)
B2. Write AKSV (40-bit source public key)
C1. Read BKSV (40-bit sink public key)
EDID ROM
HDCP ROM
C2. Initialize cipher state
B3. Initialize cipher state
HDCP cipher
HDCP cipher
D1. Synchronize cipher & encrypt data
D2. Select encrypted video source based on chroma key
D3. Synchronize cipher & decrypt data
Video, Audio data 24 bits / quanta
Video, Audio data 24 bits / quanta
NeTV MITM
Overlay video
DDC snoop/squash controller
D4'. Synchronize cipher & encrypt data
HDCP cipher
EDID shadow ROM
EDID squash ROM
A'. Intercept and override EDID
HDCP shadow ROM
B'. Intercept A_n, AKSV
C'. Intercept BKSV
D1'. Derive device private keys
D2'. Derive K_m (shared secret)
D3'. Initialize cipher

B’, C’, D’: Intercept keys & sync cipher

- Getting A_n, AKSV, BKSV accomplished with I2C snooper listening for specific addresses
- Once key exchange is captured, private key vector and shared secret must be derived
  - Final byte write of AKSV is “trigger” to start authentication
  - FPGA fires interrupt to host linux system
  - udev event starts a helper program that does the math

Computing Private Keys

$$K \cdot AKSV = APK$$

$$\begin{bmatrix} K_{00,00} & K_{00,01} & \dots & K_{00,39} \\ K_{01,00} & K_{01,01} & \dots & K_{01,39} \\ & \vdots & & \\ K_{38,00} & K_{38,01} & \dots & K_{38,39} \\ K_{39,00} & K_{39,01} & \dots & K_{39,39} \end{bmatrix} \bullet \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} APK_{00} \\ APK_{01} \\ \\ APK_{38} \\ APK_{39} \end{bmatrix}$$

$$K^T \cdot BKSV = BPK$$

$$\begin{bmatrix} K_{00,00} & K_{00,01} & \dots & K_{00,39} \\ K_{01,00} & K_{01,01} & \dots & K_{01,39} \\ & \vdots & & \\ K_{38,00} & K_{38,01} & \dots & K_{38,39} \\ K_{39,00} & K_{39,01} & \dots & K_{39,39} \end{bmatrix}^T \bullet \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} BPK_{00} \\ BPK_{01} \\ \\ BPK_{38} \\ BPK_{39} \end{bmatrix}$$

- Modular inner product of master key and public key vectors
  - HDCP master key K is 40x40 matrix of 56-bit numbers
  - AKSV, BKSV are 40-bit numbers consisting of 20 ones and 20 zeros
  - APK, BPK are 40-element vectors of 56-bit numbers

Computing Shared Secret

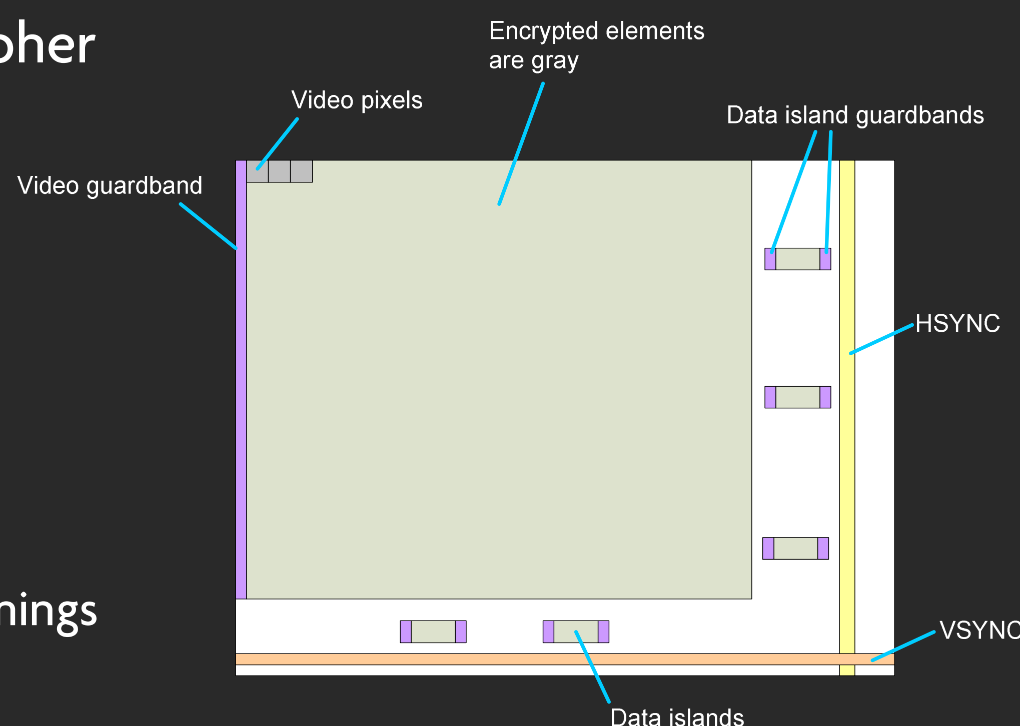
$$\mathrm{BKSV} \cdot \mathrm{APK} = K_m \qquad \mathrm{AKSV} \cdot \mathrm{BPK} = K_m$$

$$\begin{bmatrix} 0 \\ 1 \\ . \\ . \\ . \\ 1 \\ 0 \end{bmatrix} \bullet \begin{bmatrix} APK_{00} \\ APK_{01} \\ \\ \\ \\ APK_{38} \\ APK_{39} \end{bmatrix}^T = K_m \qquad \begin{bmatrix} 1 \\ 0 \\ . \\ . \\ . \\ 1 \\ 1 \end{bmatrix} \bullet \begin{bmatrix} BPK_{00} \\ BPK_{01} \\ \\ \\ \\ BPK_{38} \\ BPK_{39} \end{bmatrix}^T = K_m$$

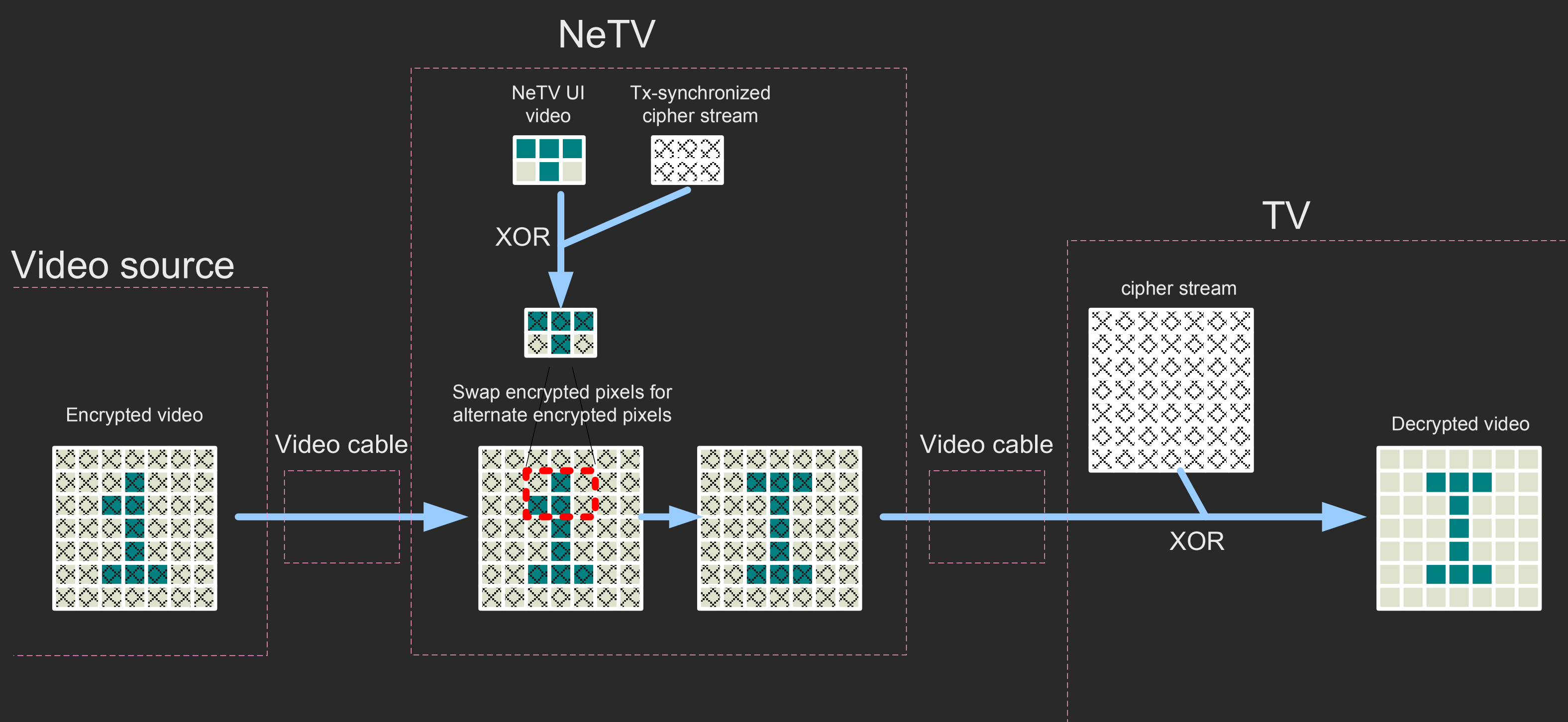
- Multiply KSVs by private keys to get 56-bit shared secret K_m

Synchronize Ciphers

- Plug An, Km into cipher hardware
- Init key schedules
- Evolve cipher state based on:
  - Pixclock
  - HSYNC
  - VSYNC
  - Data guardband timings
  - All in plaintext



Pixel-by-pixel synchronization



Synchronize Frame Buffers

- Overlay pixels must be exactly timed to video pixels
- Overlay comes from /dev/fb0 of attached linux computer
- Challenges
  - linux interrupt jitter is too high (10's to 100's of us, i.e. thousands of pixels)
  - Local crystal oscillators drift over time (100's of pixels per frame)
  - Ultimately, overlay "jitters" and "drifts" without tight synchronization

Synchronize Frame Buffers

- Technique #1: source graphics engine pixclock from video, not locally

Synchronize Frame Buffers

- Technique #1: source graphics engine pixclock from video, not locally
- Technique #2: derive timing dynamically from video stream and set /dev/fb0 properties to match

Synchronize Frame Buffers

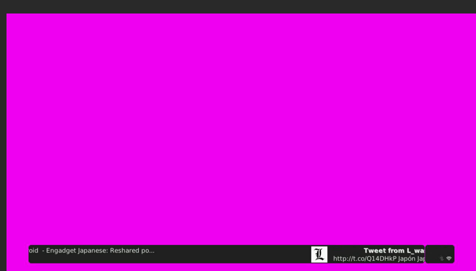
- Technique #1: source graphics engine pixclock from video, not locally
- Technique #2: derive timing dynamically from video stream and set /dev/fb0 properties to match
- Technique #3: start LCD DMA based on VSYNC start from video stream

Synchronize Frame Buffers

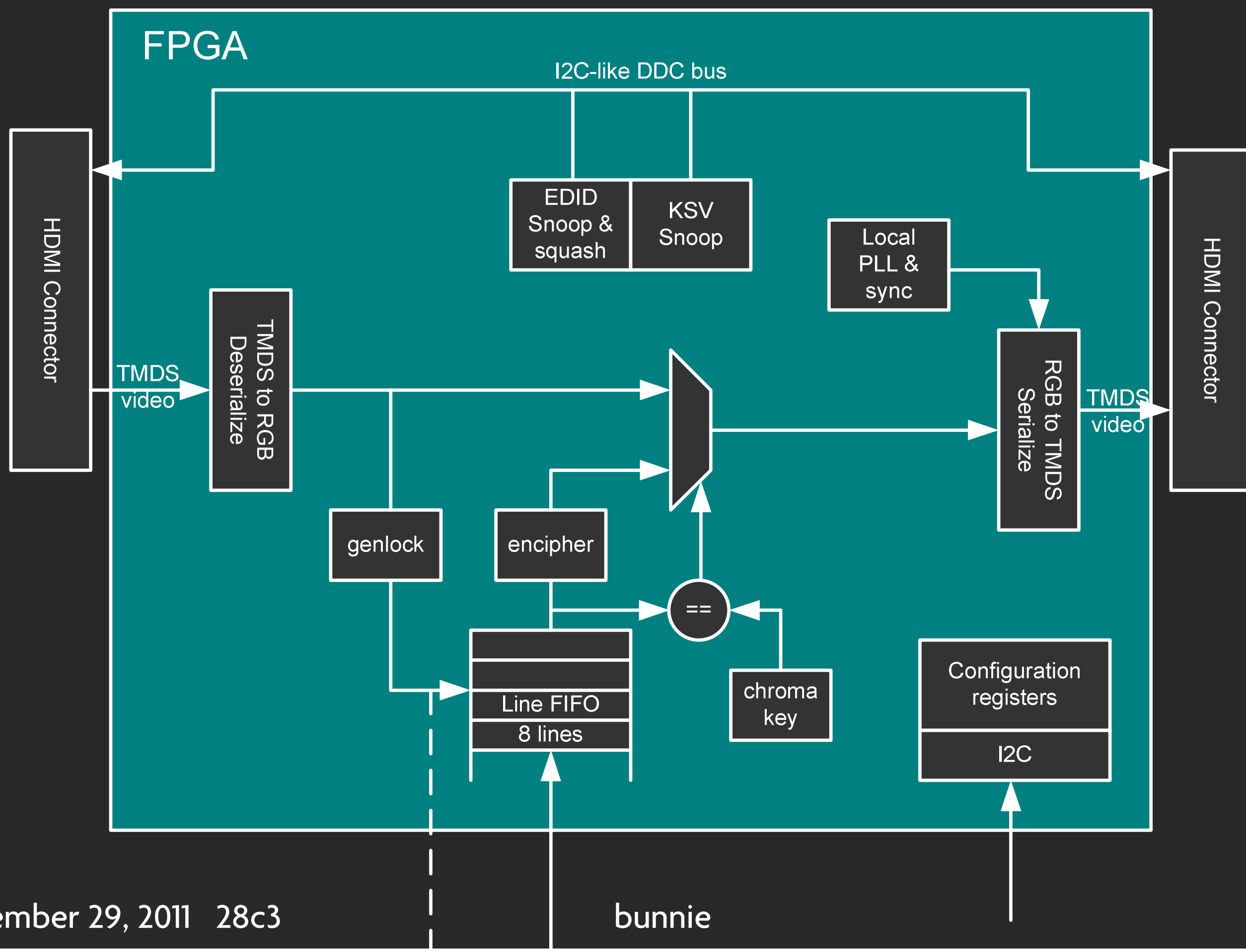
- Technique #1: source graphics engine pixclock from video, not locally
- Technique #2: derive timing dynamically from video stream and set /dev/fb0 properties to match
- Technique #3: start LCD DMA based on VSYNC start from video stream
- Technique #4: add a few video lines' elastic FIFO buffering to absorb VSYNC interrupt jitter

Chroma Key

- Chroma key reserves a specific color and substitutes its value for "transparent"
- In this implementation, F0,00,F0 (a shade of pink) is the magic color
  - A comparator within the FPGA inspects every pixel and switches a mux


+

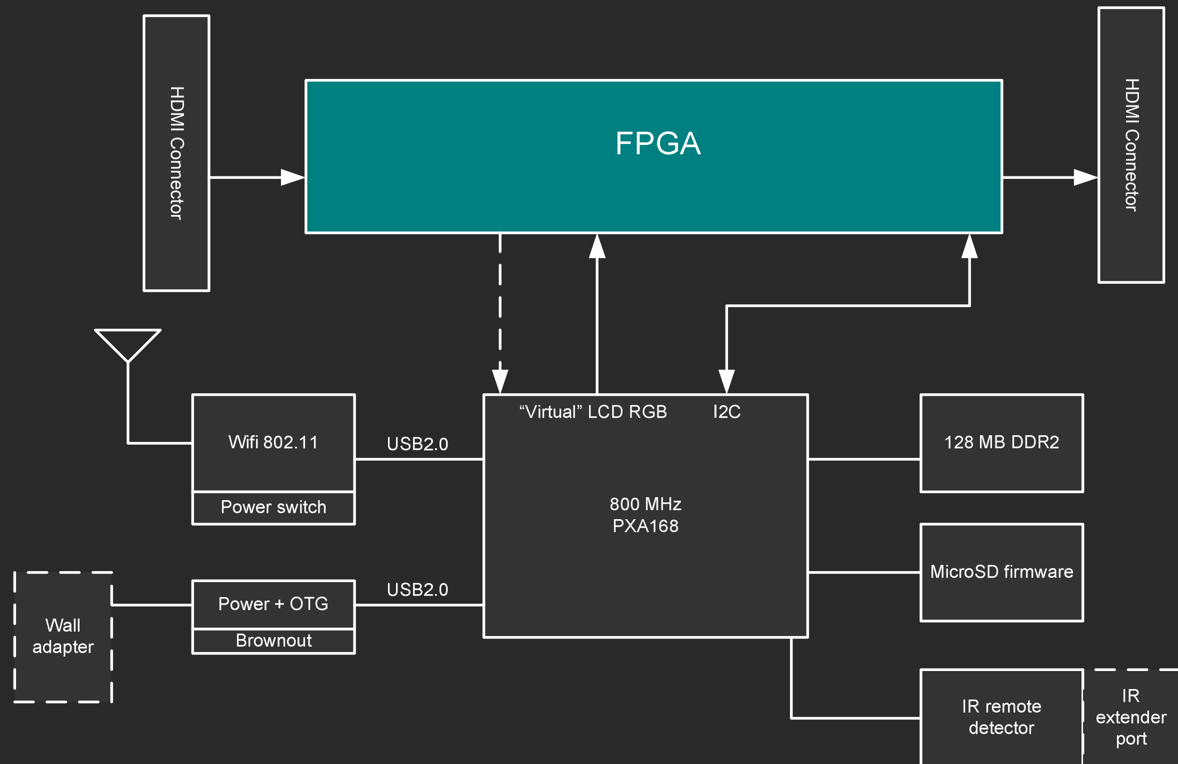
=



FPGA
I2C-like DDC bus
HDMI Connector
TMDS video
TMDS to RGB Deserialize
EDID Snoop & squash
KSV Snoop
Local PLL & sync
RGB to TMDS Serialize
TMDS video
HDMI Connector
genlock
encipher
==
Line FIFO 8 lines
chroma key
Configuration registers
I2C

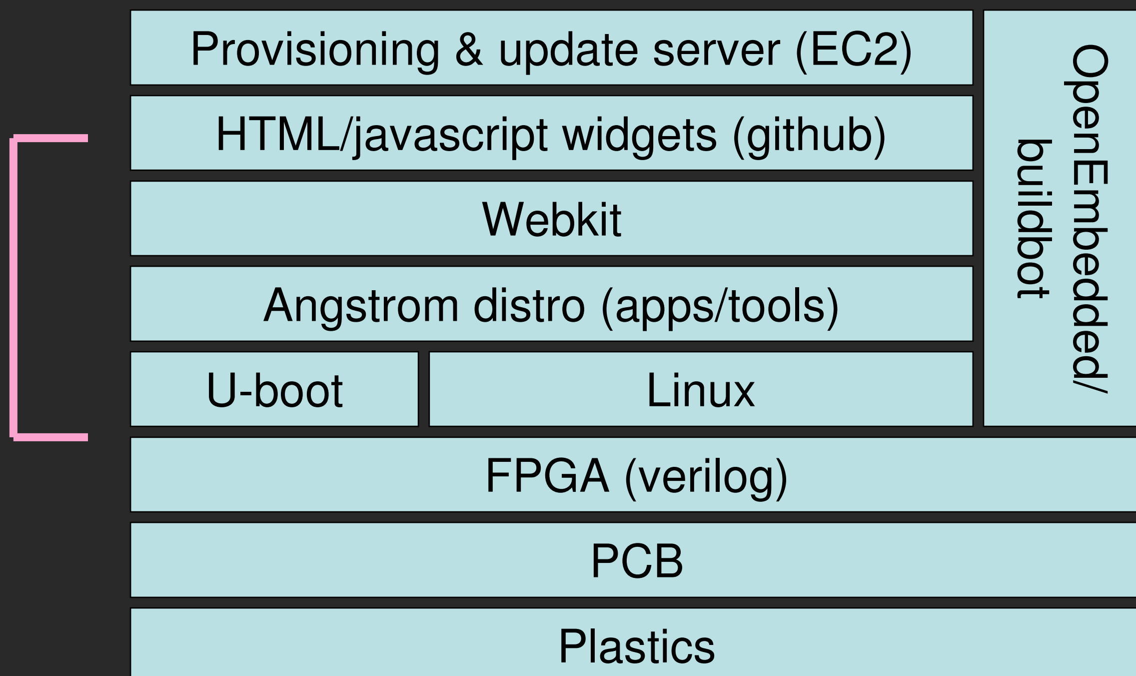
Optimizations

- Key caching
  - Every video source/sink pair has a constant shared secret
  - K_m is cached after first computation to improve system robustness
- EDID caching
  - More important because without EDID caching, users will see a double-blink of the screen
    - First blink is to measure the TV's capabilities
    - Then we compute the intersection of the TV capabilities and NeTV capabilities
    - Second blink is to override the capabilities we don't support

The Bigger System Picture



A Complete Open Stack



Application Environment

- TV overlay apps are web pages
  - CSS configured to put “magic pink” as background
  - Apps are javascript/HTML programs
  - But you can extend to any infrastructure that can write to /dev/fb0 (SDL, Flash, etc.)
- Our demo apps are open source and stored in a github repo
  - Updating apps consists of doing a git pull on the client
  - Configured to pull every reboot
- Firmware updates served from EC2 infrastructure
  - Public AMI provided so you can make your own
  - More on this later

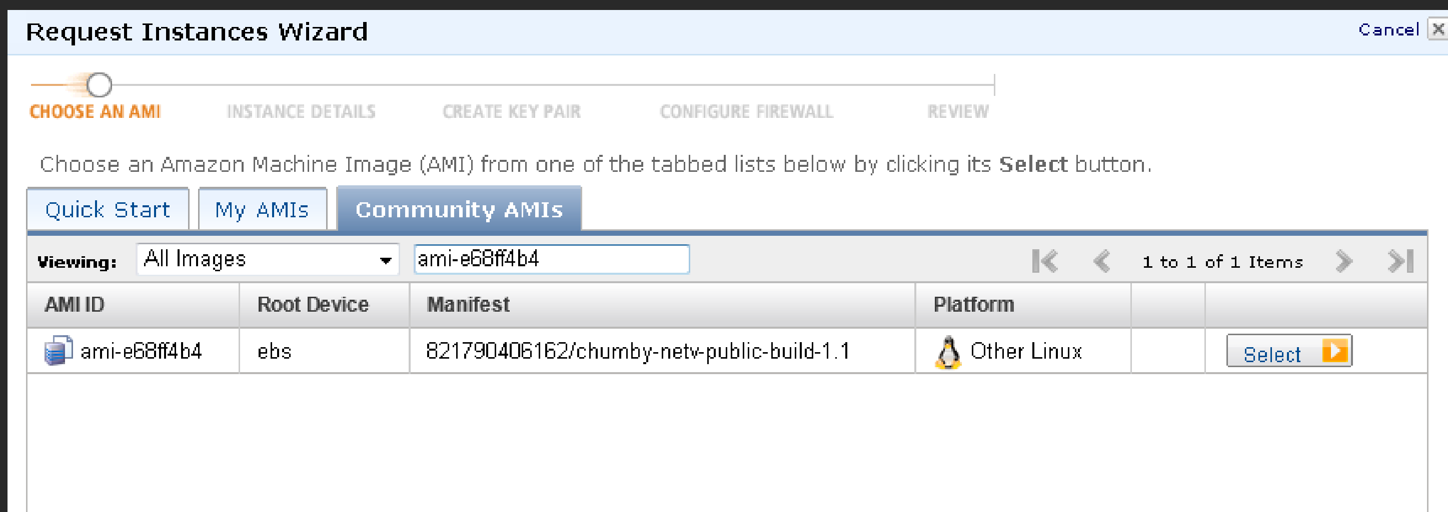
HTTP API

- Zeroconf solution for networked interaction with TV
  - API provides method to send events to NeTV
    - So, a smartphone can:
      - Discover NeTV with Bonjour
      - Send events (such as SMS) to the NeTV using HTTP GET
      - NeTV renders these events on your TV
    - Also provides a method for file upload to enable photosharing to the TV
  - Fast, easy integration into “smarthome” environment
  - Example call: http://10.0.88.1/bridge?cmd=tickerevent&message=Hello%World&title=Hello%20World
  - Each API call can be restricted to just localhost for security

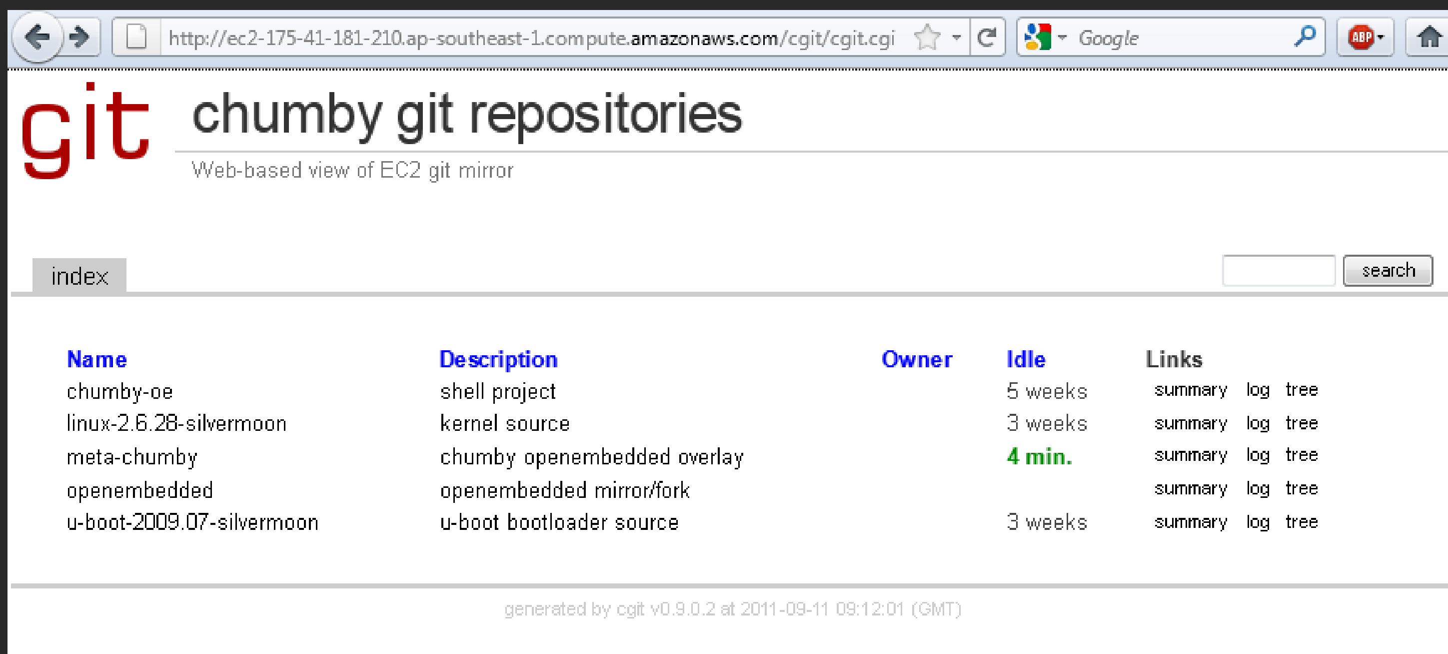
Turnkey Build System

- Public Amazon EC2 instance with pre-built Angstrom distribution
  - Saves hours of effort downloading & building sources
  - Instance comes configured with local git repo and buildbot to manage builds
  - Built images configured to fetch updates from your own instance

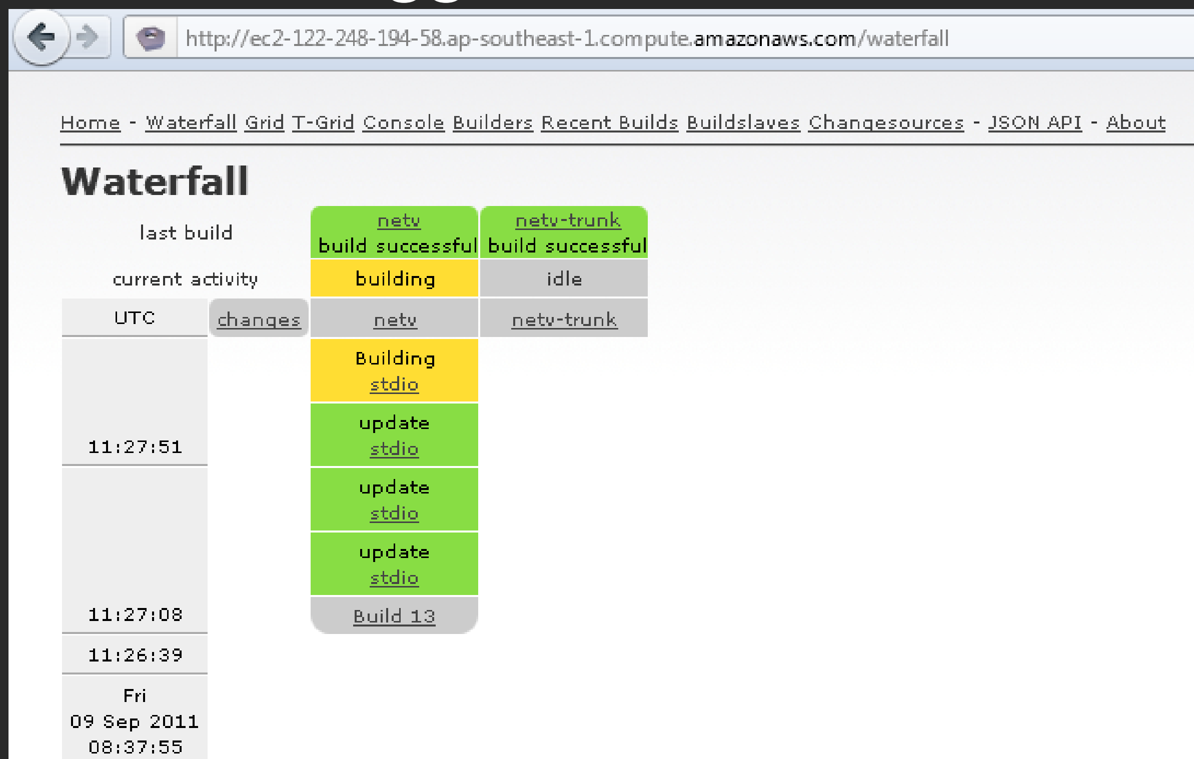
Launching an EC2 AMI



Local cgit repo

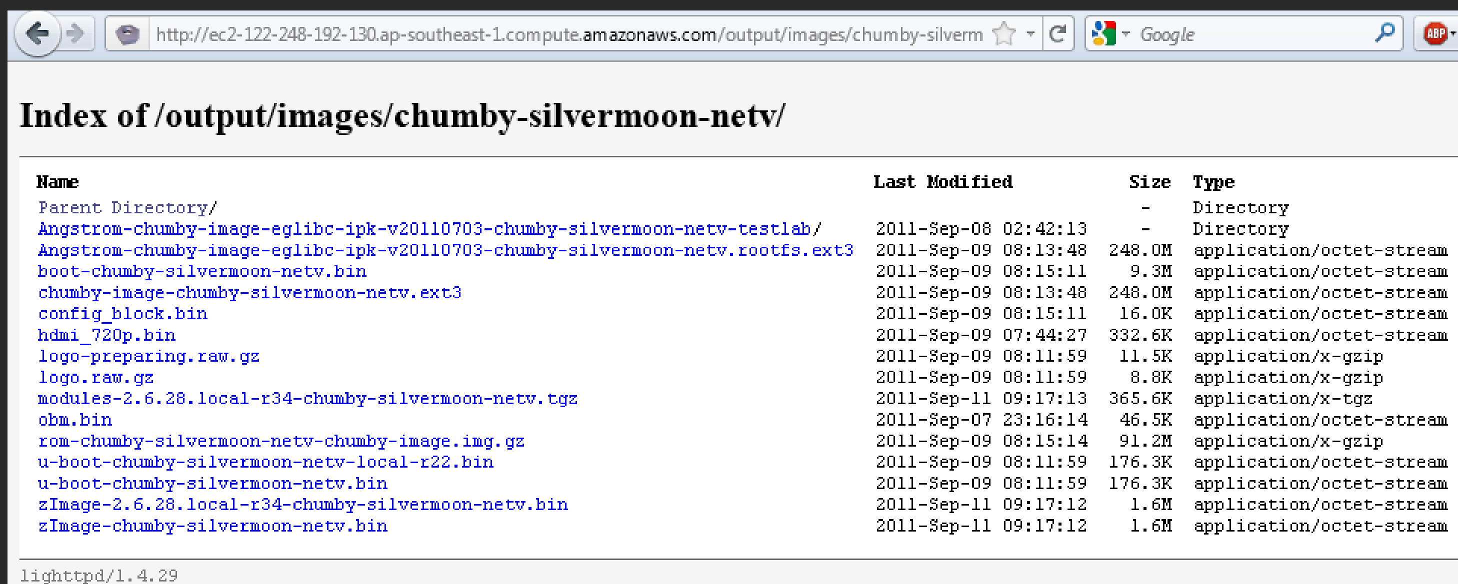


Auto-build triggers based on commits



Distribute Finished Builds

- Image once, auto-update forever

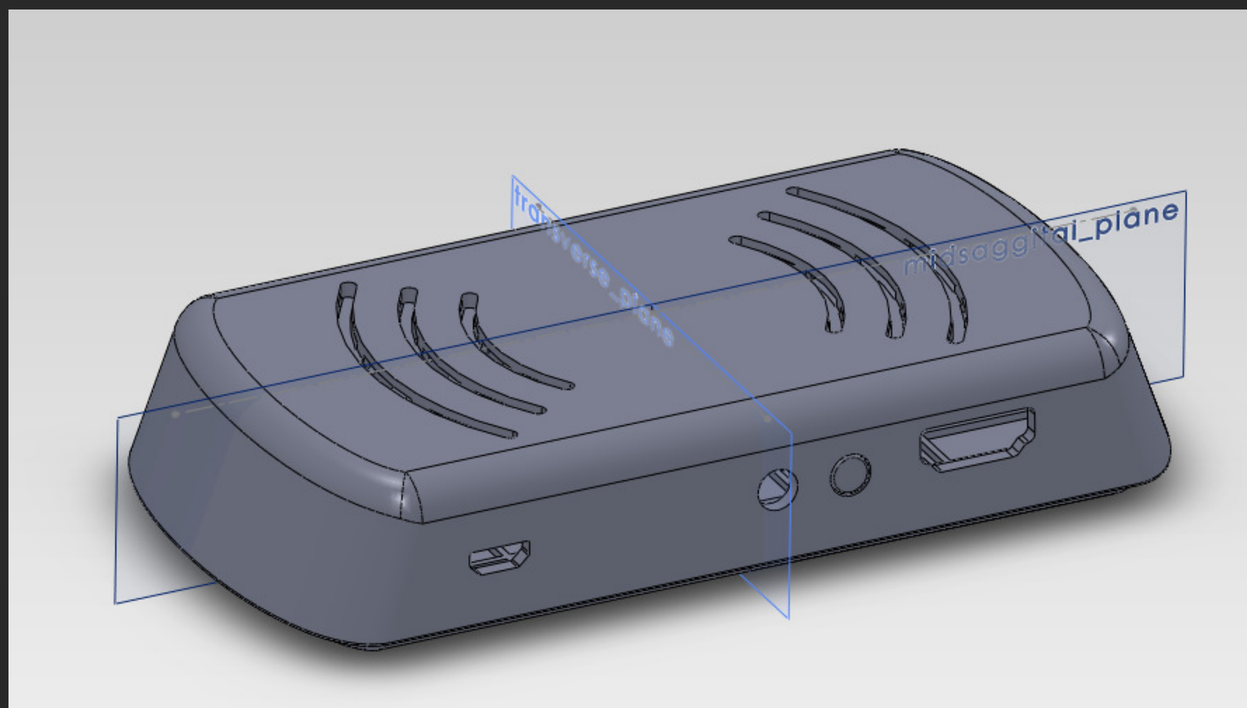


Index of /output/images/chumby-silvermoon-netv/

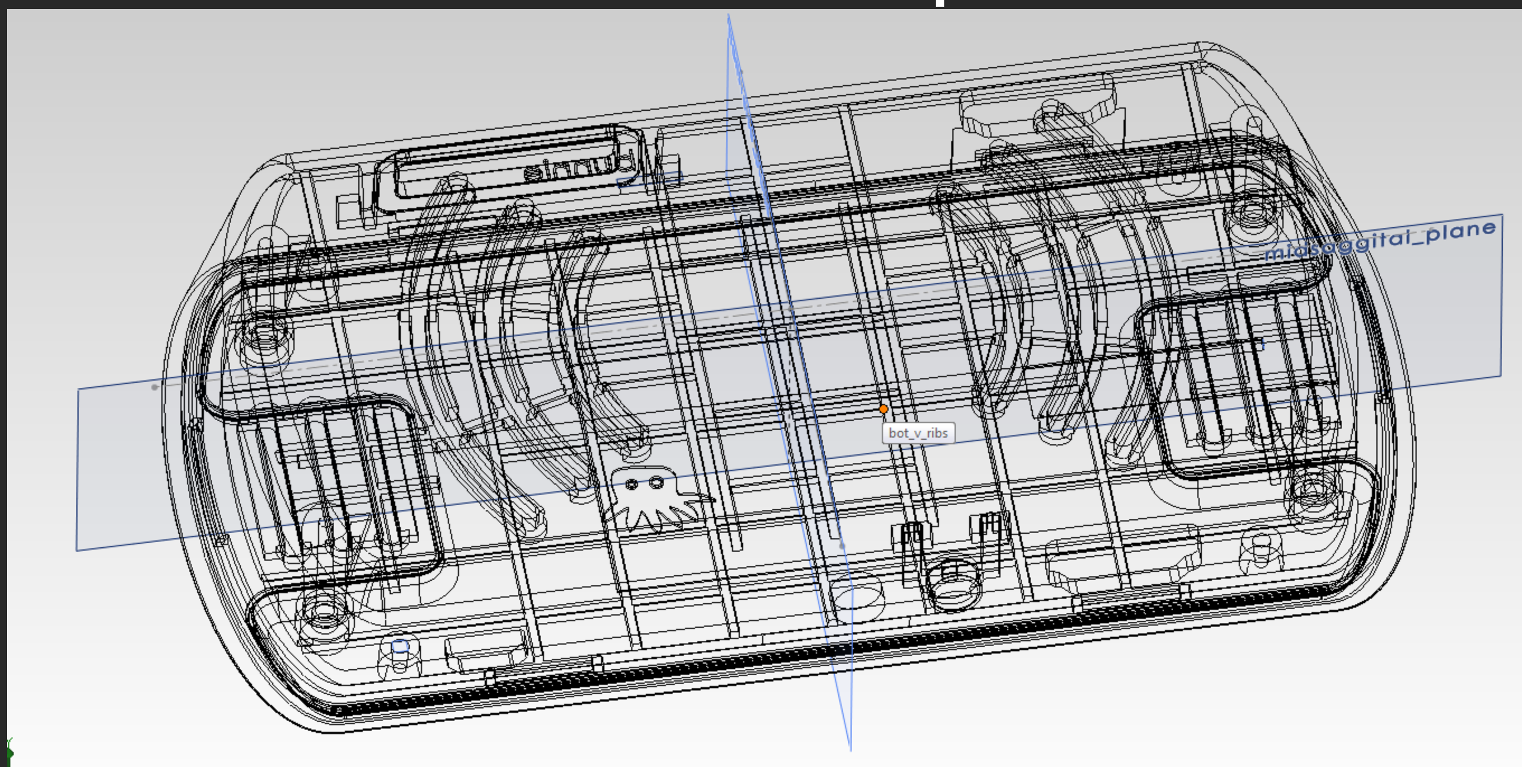
Name	Last Modified	Size	Type
Parent Directory/		-	Directory
Angstrom-chumby-image-eglibc-ipk-v20110703-chumby-silvermoon-netv-testlab/	2011-Sep-08 02:42:13	-	Directory
Angstrom-chumby-image-eglibc-ipk-v20110703-chumby-silvermoon-netv.rootfs.ext3	2011-Sep-09 08:13:48	248.0M	application/octet-stream
boot-chumby-silvermoon-netv.bin	2011-Sep-09 08:15:11	9.3M	application/octet-stream
chumby-image-chumby-silvermoon-netv.ext3	2011-Sep-09 08:13:48	248.0M	application/octet-stream
config_block.bin	2011-Sep-09 08:15:11	16.0K	application/octet-stream
hdmi_720p.bin	2011-Sep-09 07:44:27	332.6K	application/octet-stream
logo-preparing.raw.gz	2011-Sep-09 08:11:59	11.5K	application/x-gzip
logo.raw.gz	2011-Sep-09 08:11:59	8.8K	application/x-gzip
modules-2.6.28.local-r34-chumby-silvermoon-netv.tgz	2011-Sep-11 09:17:13	365.6K	application/x-tgz
obm.bin	2011-Sep-07 23:16:14	46.5K	application/octet-stream
rom-chumby-silvermoon-netv-chumby-image.img.gz	2011-Sep-09 08:15:14	91.2M	application/x-gzip
u-boot-chumby-silvermoon-netv-local-r22.bin	2011-Sep-09 08:11:59	176.3K	application/octet-stream
u-boot-chumby-silvermoon-netv.bin	2011-Sep-09 08:11:59	176.3K	application/octet-stream
zImage-2.6.28.local-r34-chumby-silvermoon-netv.bin	2011-Sep-11 09:17:12	1.6M	application/octet-stream
zImage-chumby-silvermoon-netv.bin	2011-Sep-11 09:17:12	1.6M	application/octet-stream

lighttpd/1.4.29

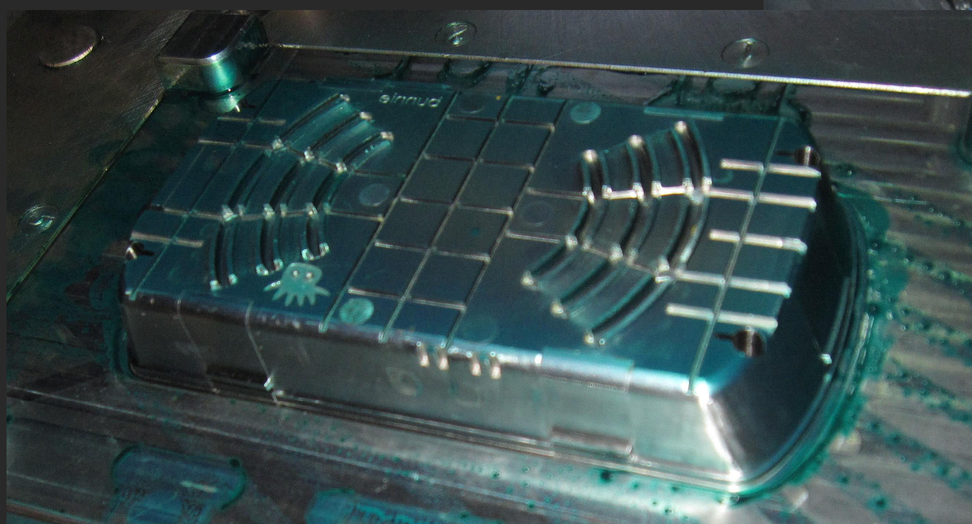
Hardware is Open

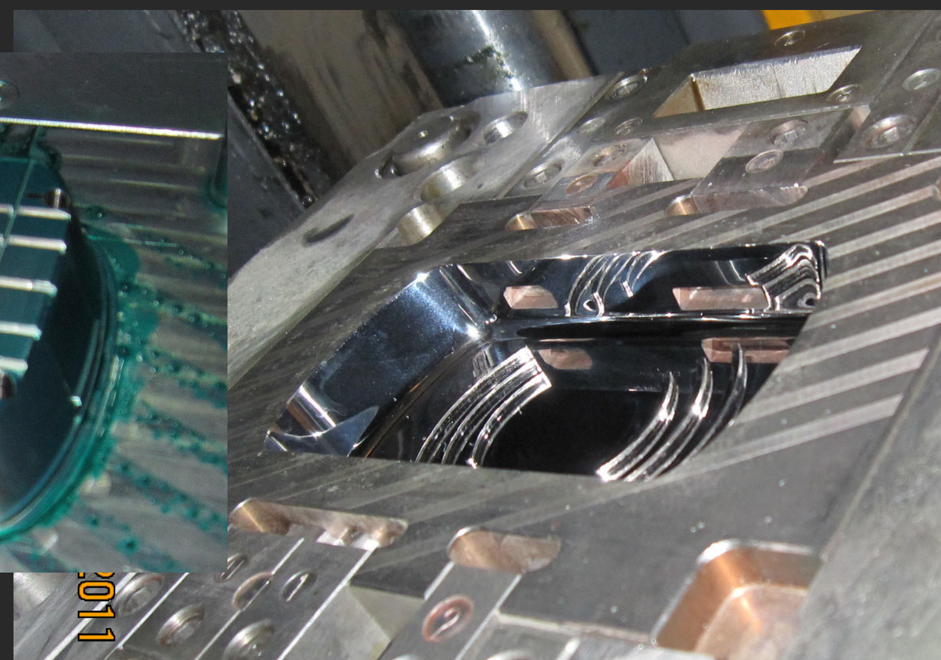


Plastics are Open

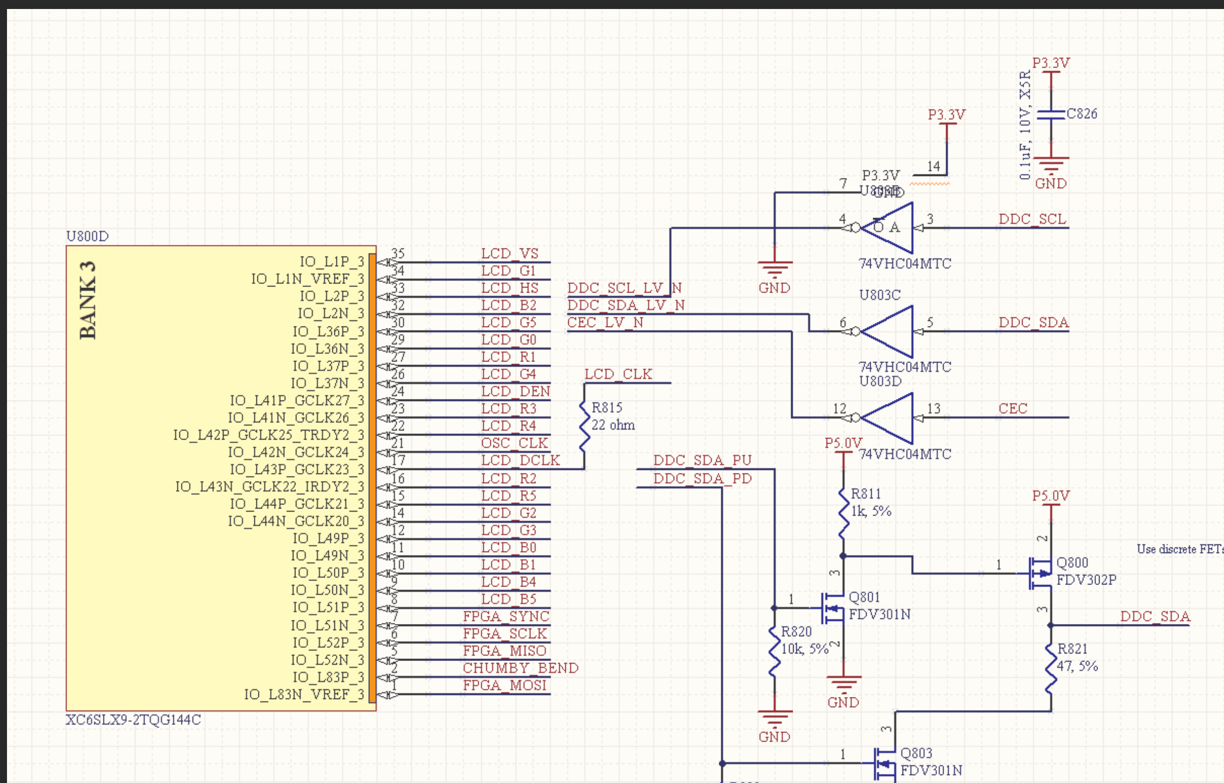


Pr0n



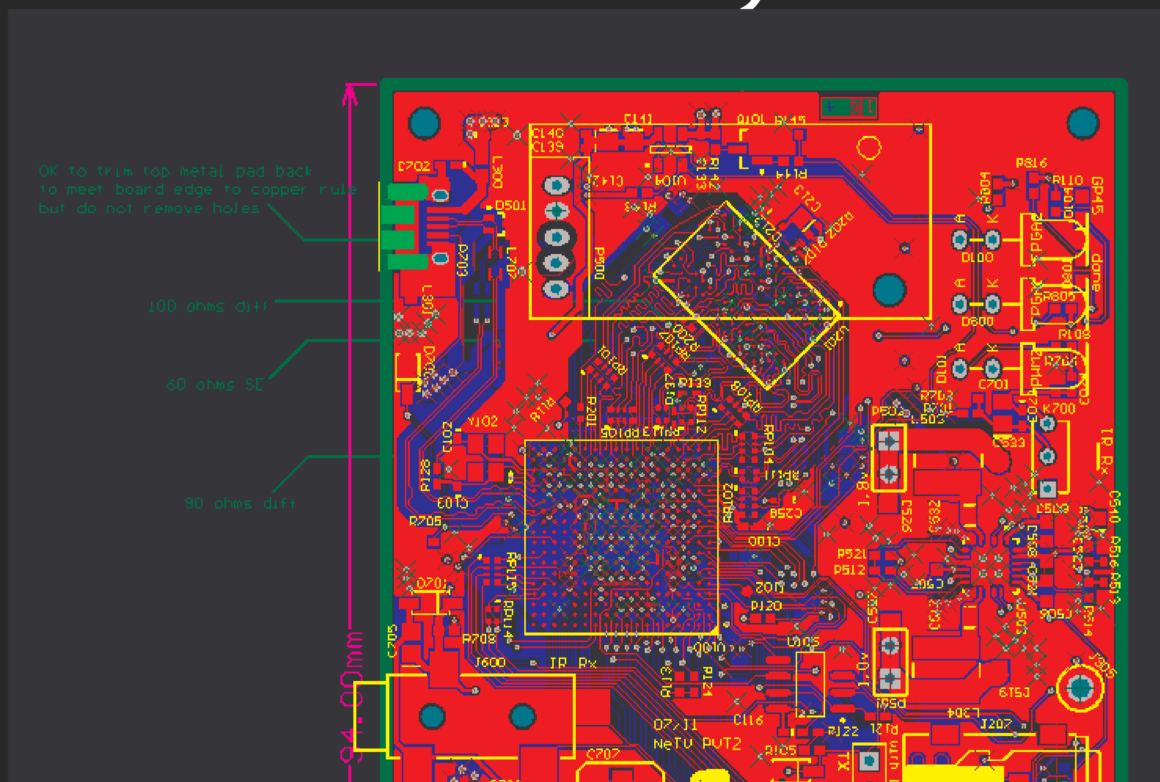


Schematics



U800D
BANK 3
IO_L1P_3
IO_L1N_VREF_3
IO_L2P_3
IO_L2N_3
IO_L36P_3
IO_L36N_3
IO_L37P_3
IO_L37N_3
IO_L41P_GCLK27_3
IO_L41N_GCLK26_3
IO_L42P_GCLK25_TRDY2_3
IO_L42N_GCLK24_3
IO_L43P_GCLK23_3
IO_L43N_GCLK22_IRDY2_3
IO_L44P_GCLK21_3
IO_L44N_GCLK20_3
IO_L49P_3
IO_L49N_3
IO_L50P_3
IO_L50N_3
IO_L51P_3
IO_L51N_3
IO_L52P_3
IO_L52N_3
IO_L83P_3
IO_L83N_VREF_3
XC6SLX9-2TQG144C
LCD_VS
LCD_G1
LCD_HS
LCD_B2
LCD_G5
LCD_G0
LCD_R1
LCD_G4
LCD_DEN
LCD_R3
LCD_R4
OSC_CLK
LCD_DCLK
LCD_R2
LCD_R5
LCD_G2
LCD_G3
LCD_B0
LCD_B1
LCD_B4
LCD_B5
FPGA_SYNC
FPGA_SCLK
FPGA_MISO
CHUMBY_BEND
FPGA_MOSI
DDC_SCL_LV_N
DDC_SDA_LV_N
CEC_LV_N
LCD_CLK
R815
22 ohm
DDC_SDA_PU
DDC_SDA_PD
P3.3V
C826
0.1uF, 10V, X5R
GND
74VHC04MTC
U803C
U803D
DDC_SCL
DDC_SDA
CEC
P5.0V
R811
1k, 5%
Q801
FDV301N
R820
10k, 5%
Q800
FDV302P
R821
47, 5%
Q803
FDV301N
R822
Use discrete FETs

And PCB Layout



A Complete Open Solution to HDCP MITM

- Hardware
  - Schematics, PCB, industrial design, FPGA
- Software
  - Complete, turn-key cloud-based build environment
    - Half an hour from start to production-grade deployment
- Available at adafruit.com (http://www.adafruit.com/products/609)

Recap: HDCP MITM Implementation

- Complete HDCP MITM solution demonstrated
  - Intercept key exchange on the fly
  - Derive shared secrets & synchronize Tx ciphers
  - Multiplex overlay video using chroma key
  - Avoids decrypting data, therefore DMCA-safe
  - Modifies EDID records to force compatibility
- Enables video compositing functionality
  - Enables unconnected legacy TVs to now have connected TV capabilities
  - Enables you to modify your video content (stop/modify ads, show live internet commentary, etc.)
- A completely open hardware/software stack

Non-Infringing Use of HDCP Master Key!

- Embodiment of a bona-fide, non-infringing and commercially useful application of the HDCP master key
- Blurs the association of the HDCP master key with piracy
  - Prior to this exploit, the only application of the HDCP master key was to circumvent the encryption on copyrighted data
  - Now, there is a non-circumventing application for the HDCP master key

Q&A