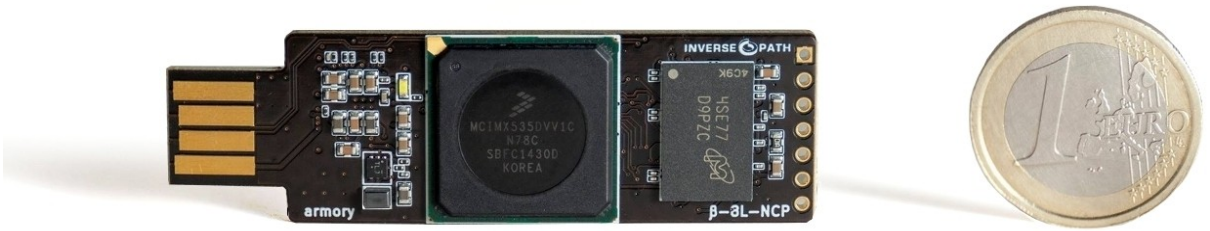
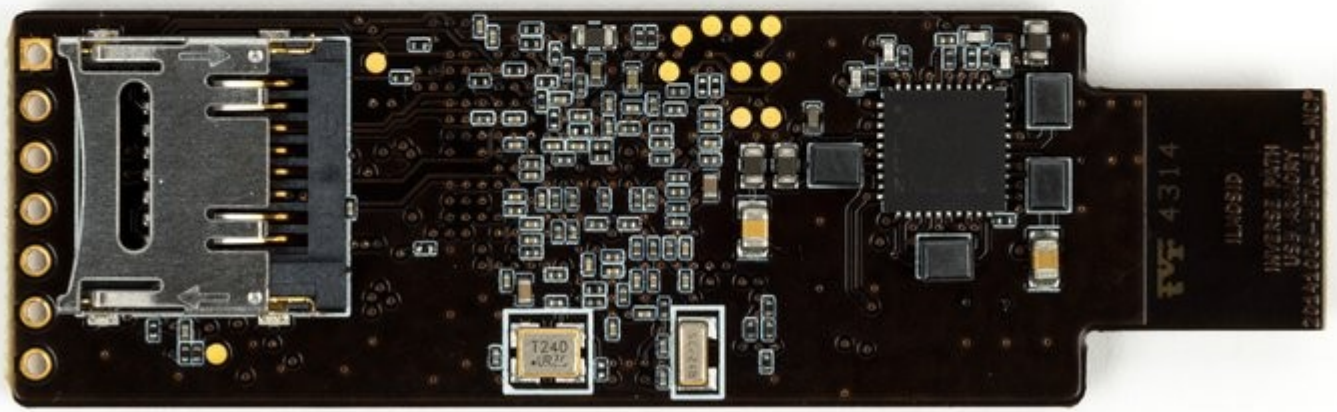
INVERSE PATH

Forging the USB armory

Andrea Barisani

<andrea@inversepath.com>





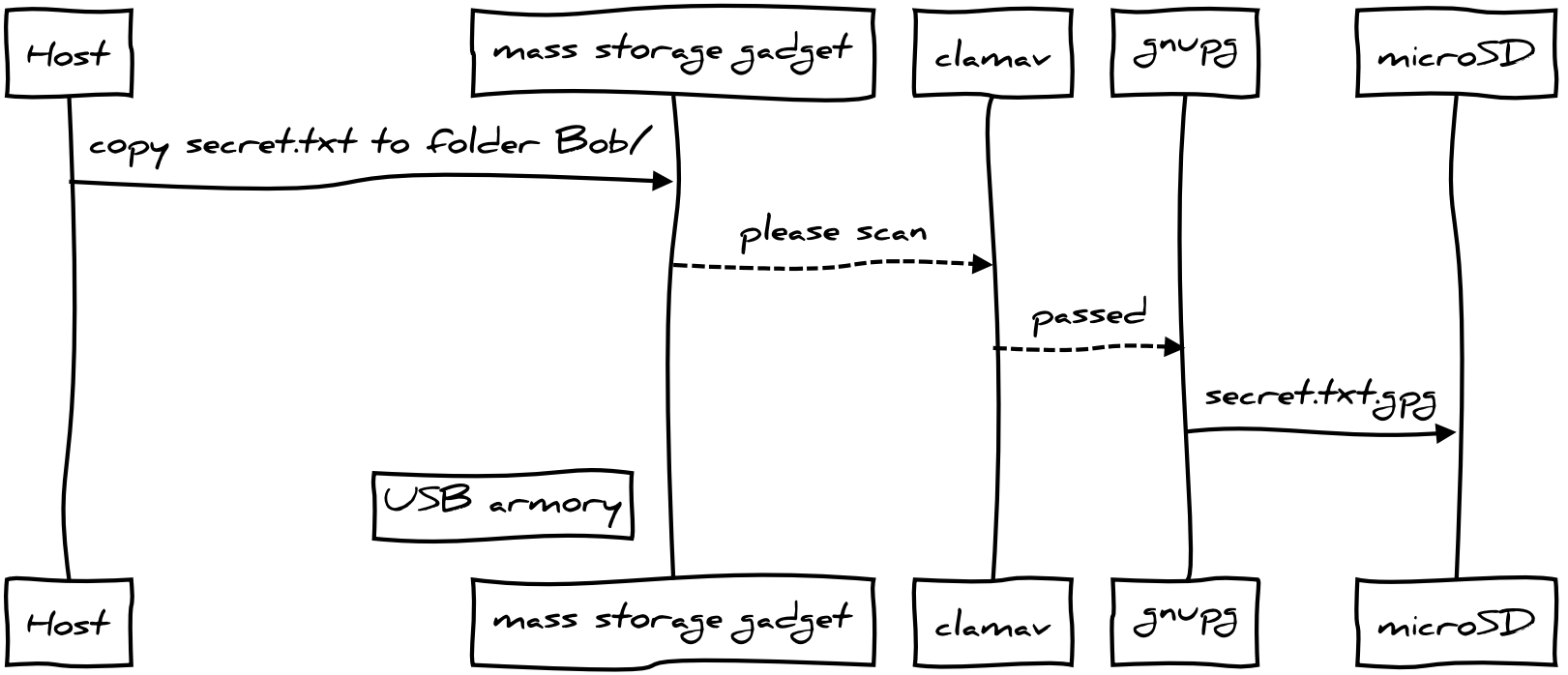
INVERSE PATH

Designed for personal security applications

- mass storage device with advanced features such as automatic encryption, virus scanning, host authentication and data self-destruct
- OpenSSH client and agent for untrusted hosts (kiosk)
- router for end-to-end VPN tunneling, Tor
- password manager with integrated web server
- electronic wallet (e.g. pocket Bitcoin wallet)
- authentication token
- portable penetration testing platform
- low level USB security testing

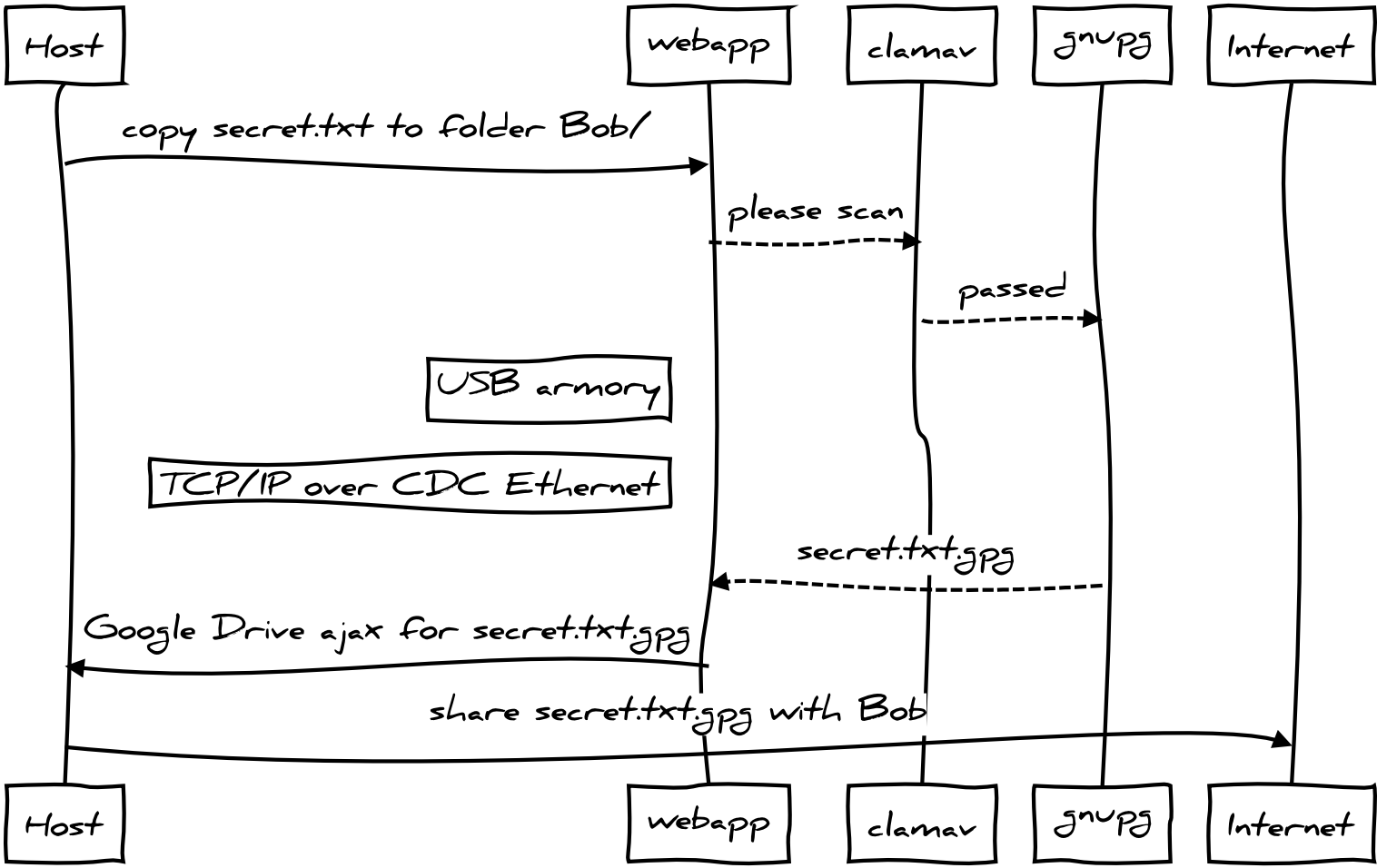
INVERSE PATH

enhanced mass storage



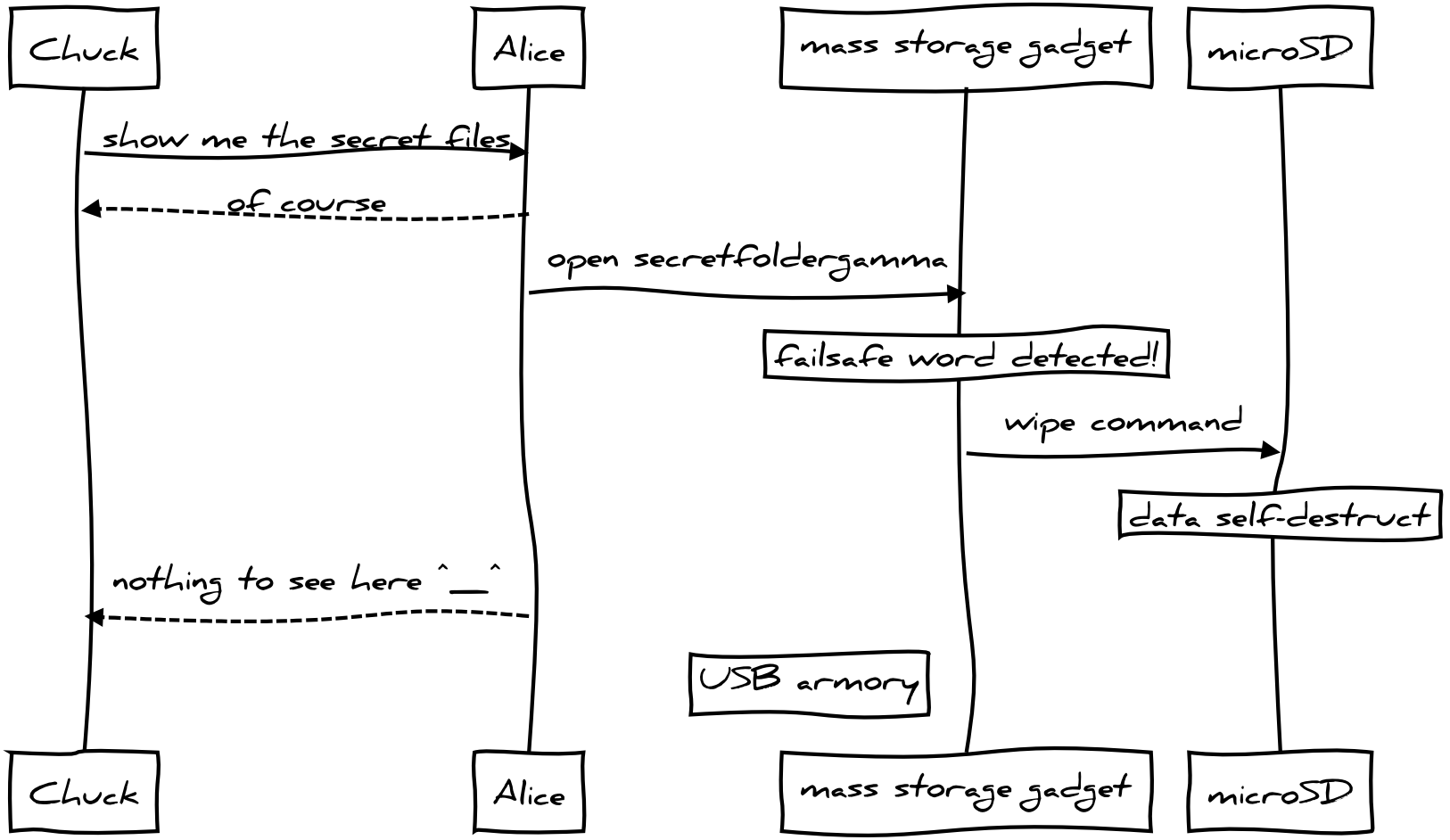
INVERSE PATH

enhanced mass storage



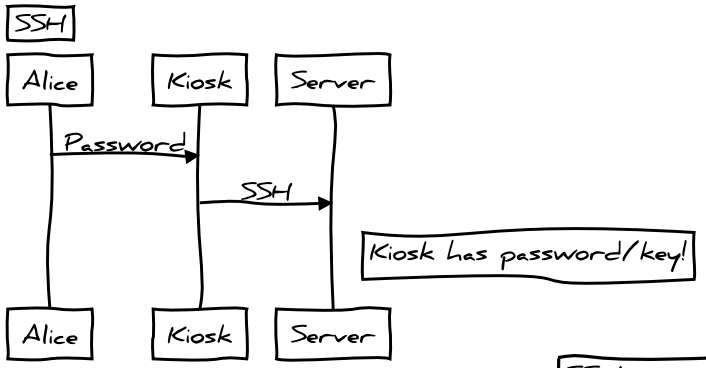
INVERSE PATH

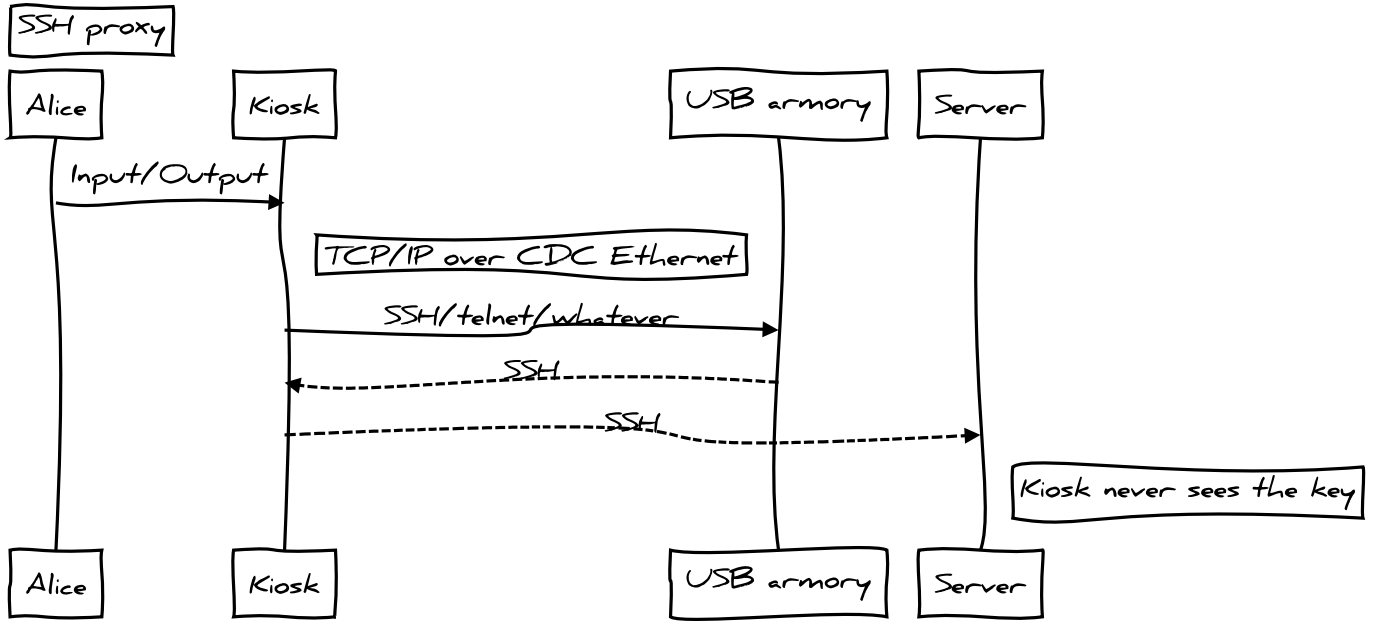
enhanced mass storage

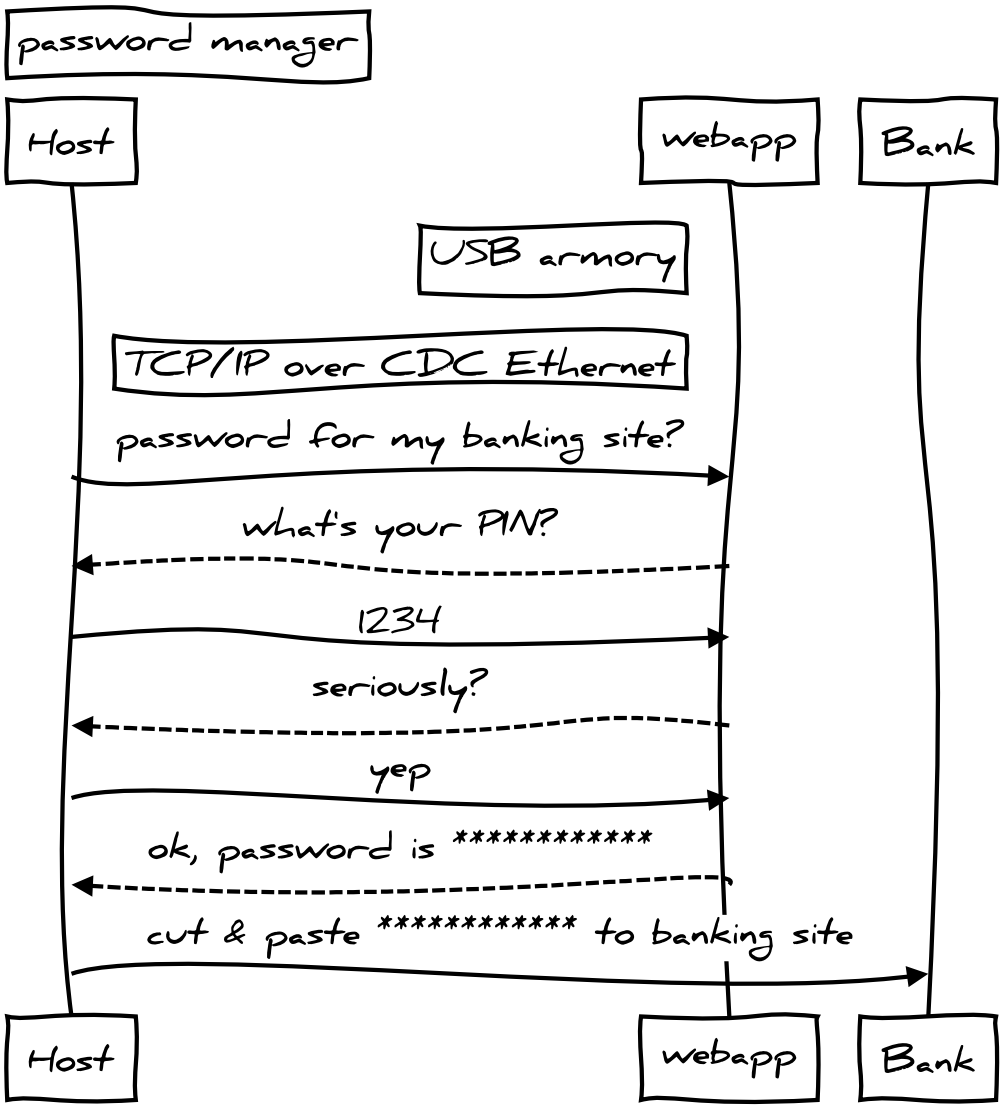


INVERSE PATH

SSH proxy





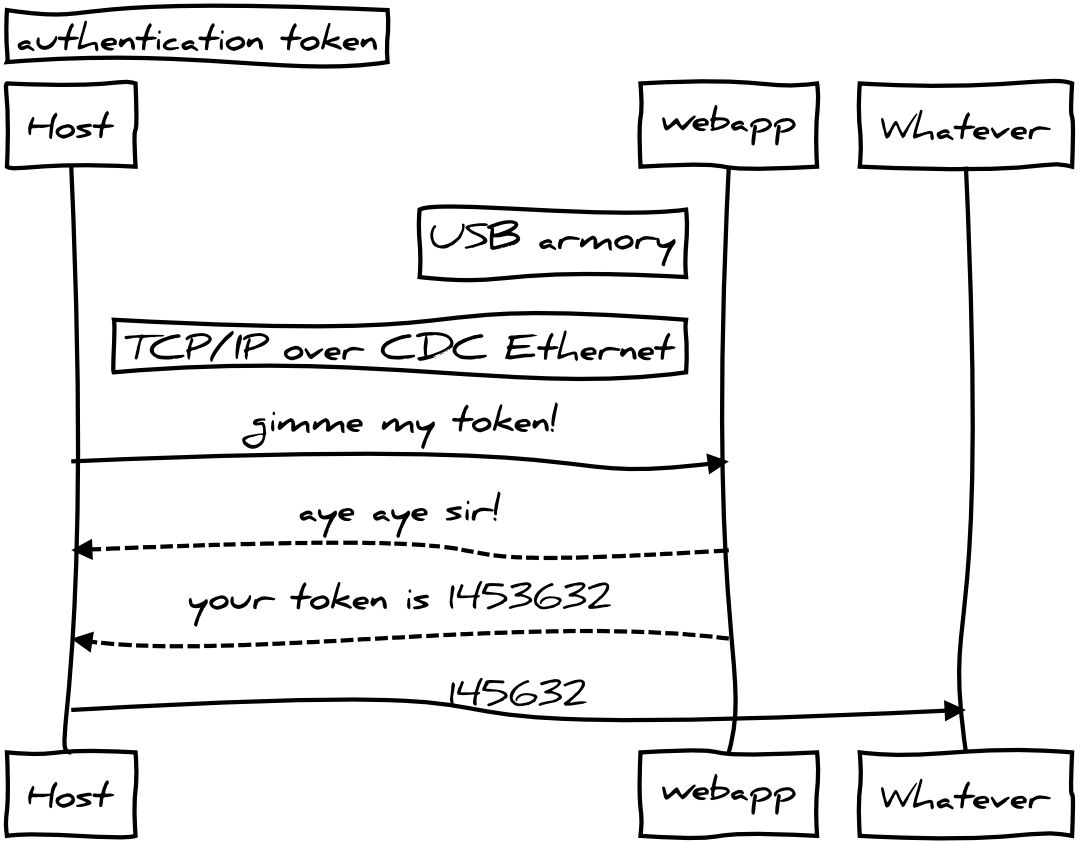


password manager*

**trivial and naive example*

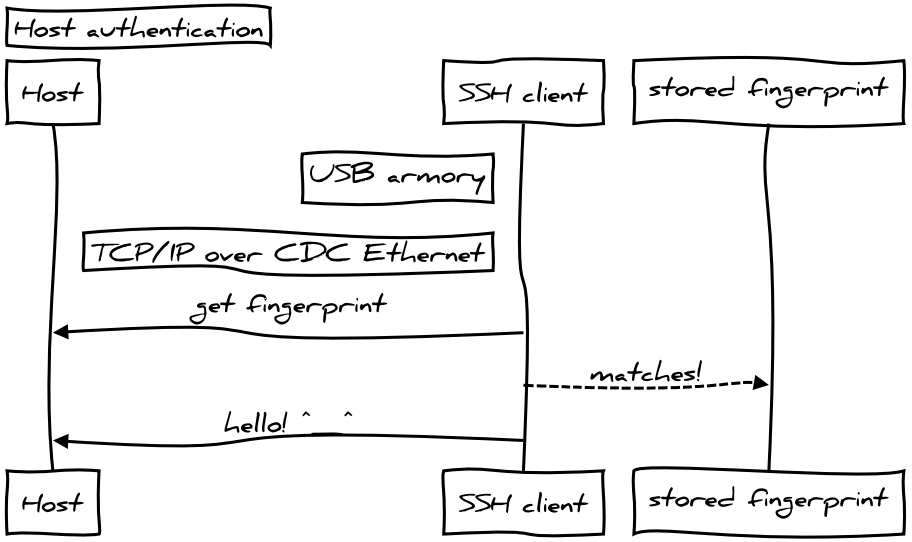
INVERSE PATH

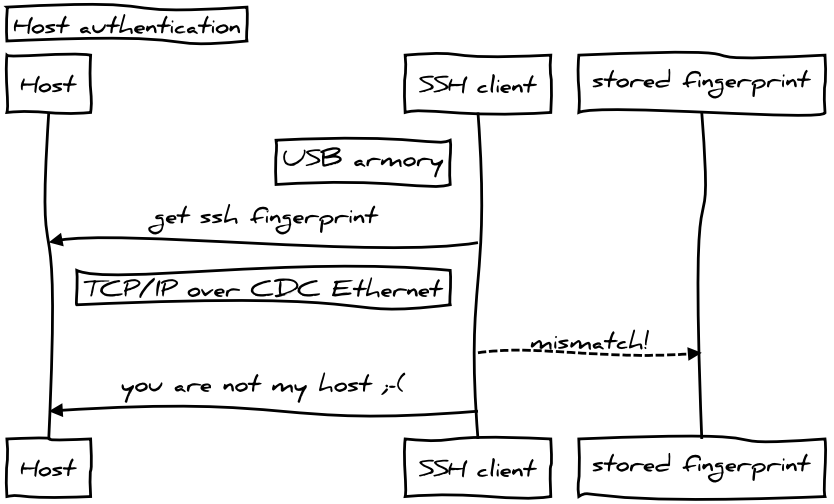
authentication token



INVERSE PATH

USB device authenticates host





INVERSE PATH

Design goals

Compact USB powered device

Fast CPU and generous RAM

Secure boot

Standard connectivity over USB

Familiar developing/execution environment

Open design

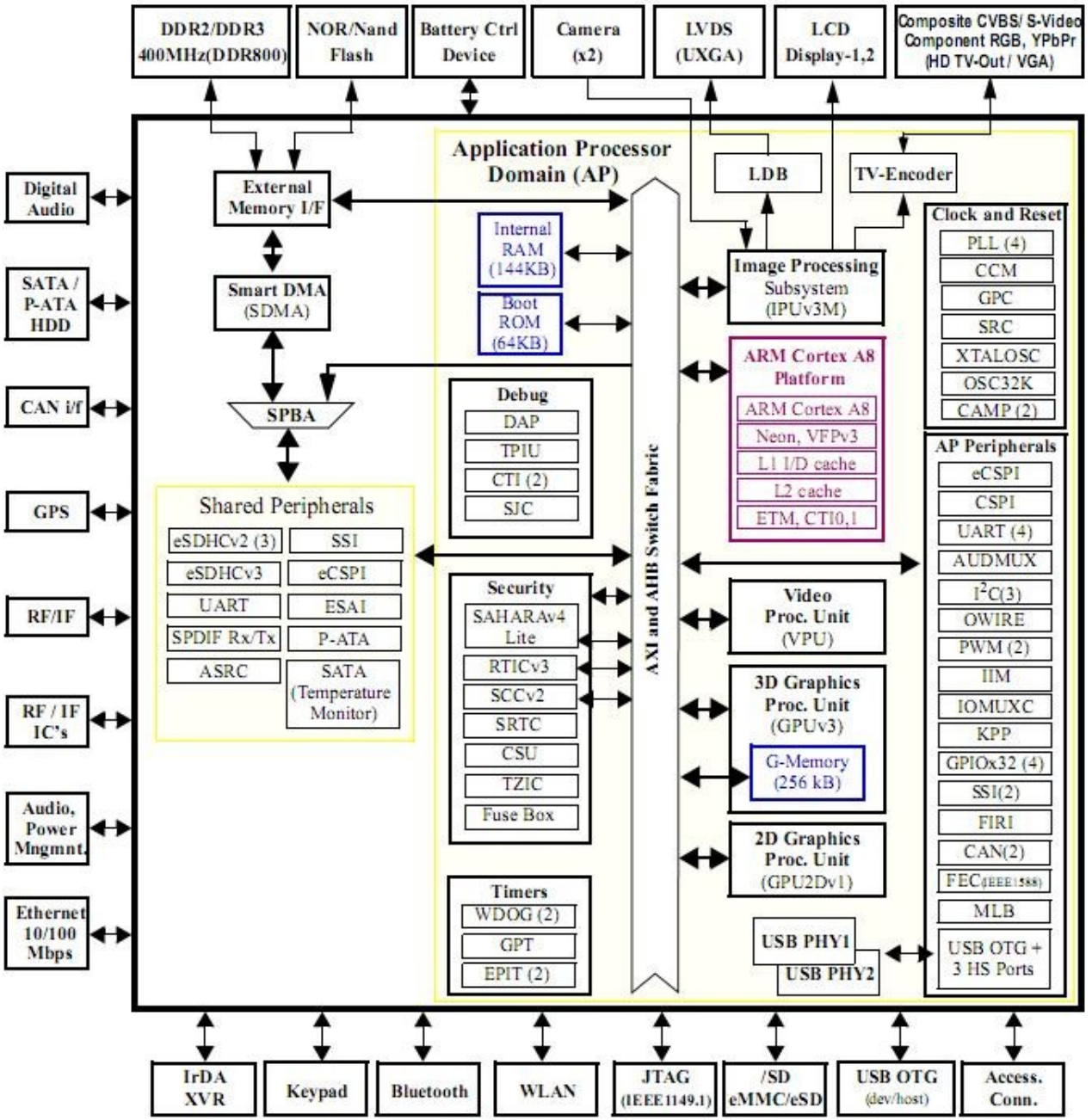
INVERSE PATH

Selecting the System on Chip (SoC)

NXP i.MX53

- ARM® Cortex™-A8 800-1200 Mhz
- almost all datasheets/manuals are public (no NDA required)
- Freescale datasheets are “ok” (far better than other vendors)
- HABv4, SCCv2, SAHARAv4 Lite, ARM® TrustZone®
- detailed power consumption guide available
- excellent native support (Android, Debian, Ubuntu, FreeBSD)
- good stock and production support guarantee

INVERSE PATH



INVERSE PATH

NXP i.MX53 – Hardware security features

High Assurance Boot (HABv4)

authentication of initial bootloader (a.k.a. Secure Boot)

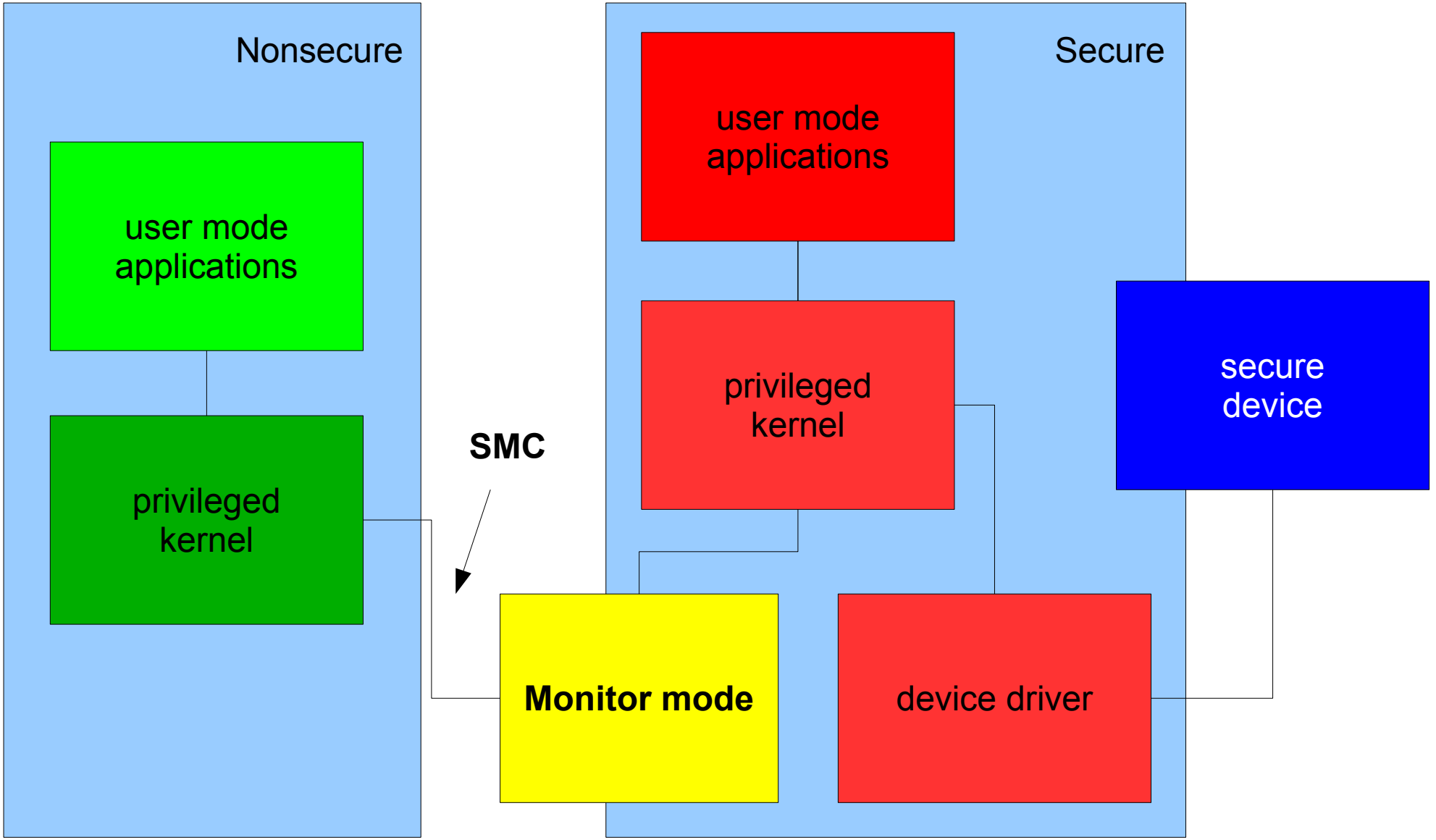
Security Controller (SCCv2)

security assurance hardware module with device specific hidden secret key for AES-256 encryption/decryption and secure RAM

Cryptographic accelerator (SAHARAv4 Lite)

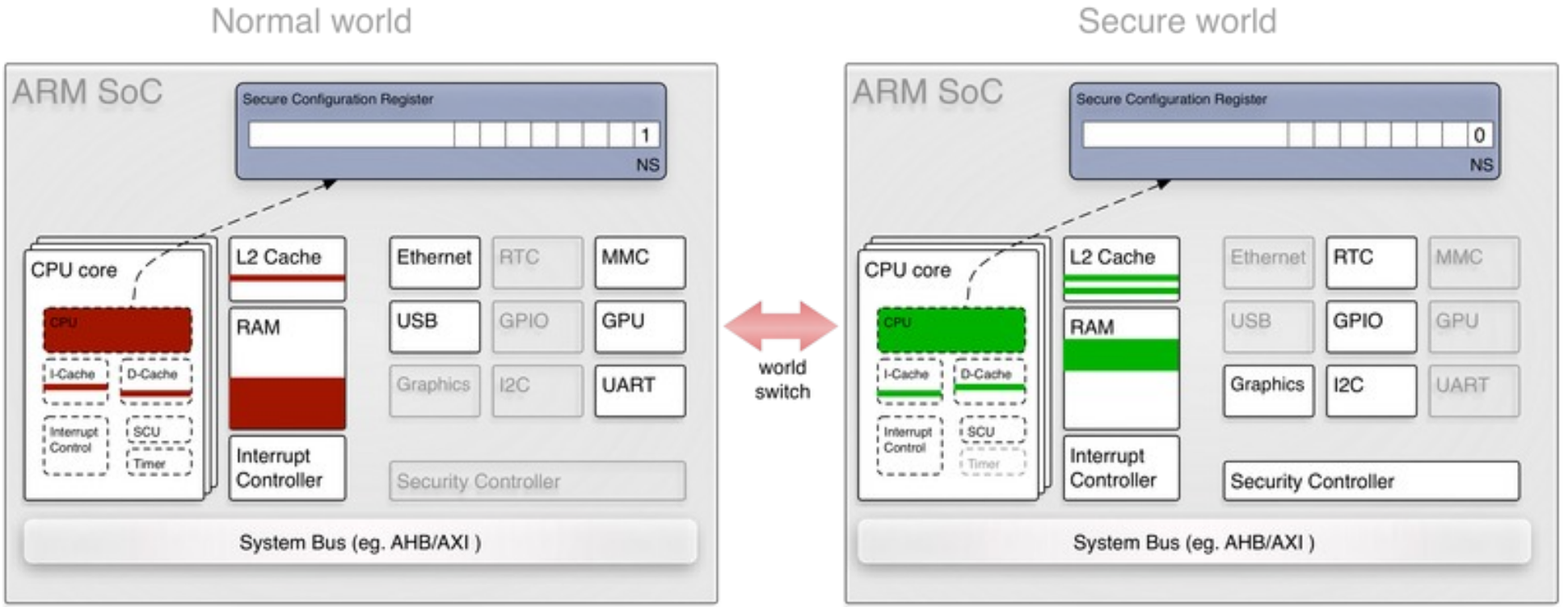
AES, DES, 3DES, RC4, C2, RSA, ECC, MD5, SHA-1, SHA-224, SHA-256 and TRNG

ARM® TrustZone®



INVERSE PATH

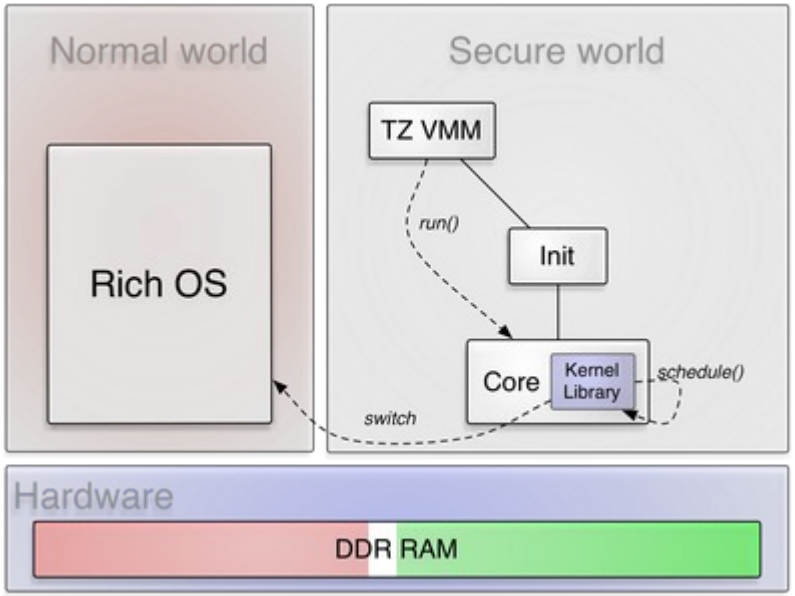
ARM® TrustZone®

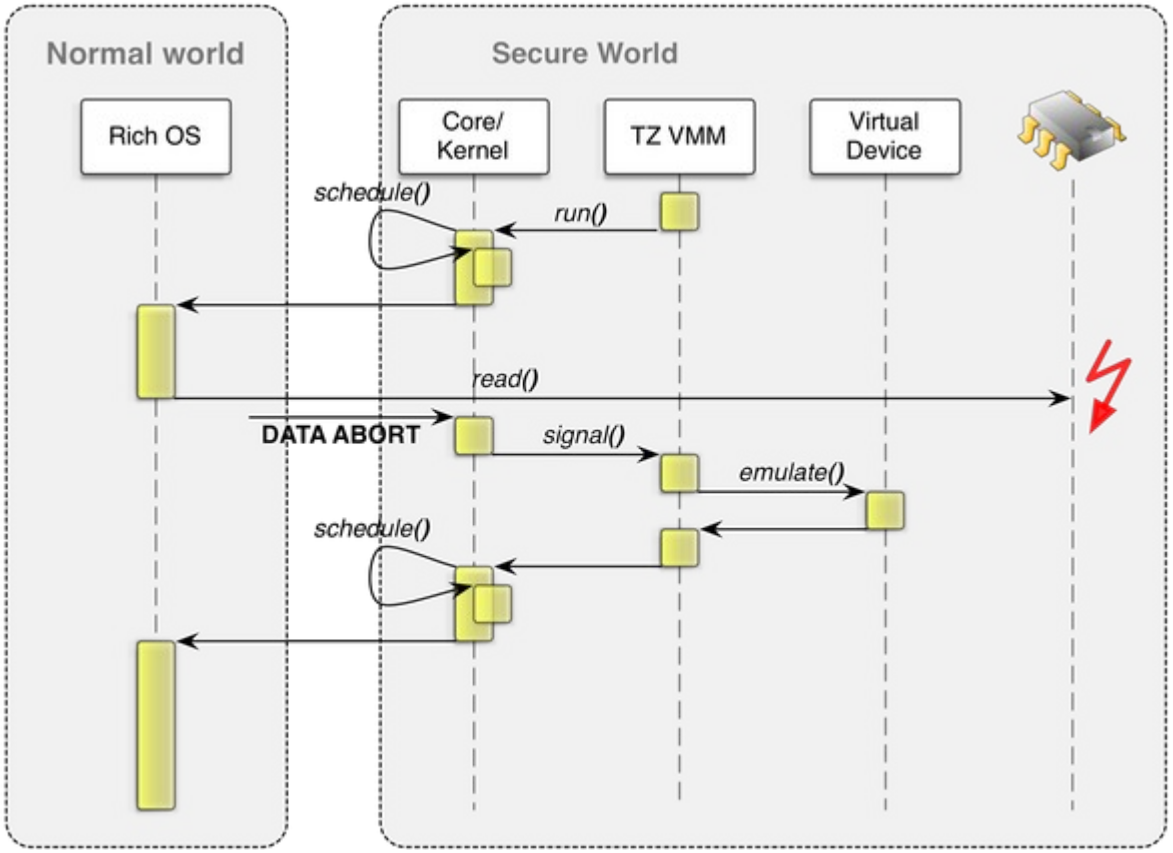


http://genode.org/documentation/articles/trustzone

INVERSE PATH

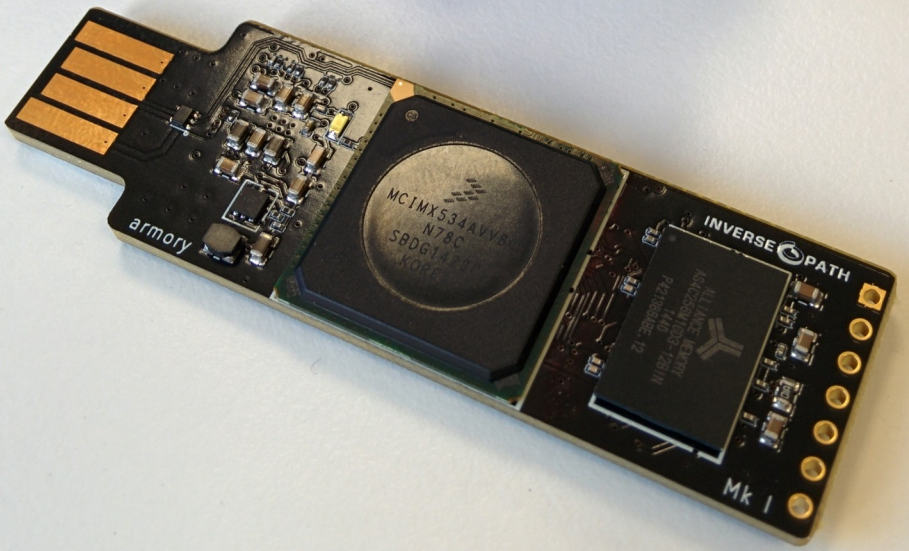
ARM® TrustZone®





http://genode.org/documentation/articles/trustzone

INVERSE PATH



open source hardware

open source

http://inversepath.com/usbarmory

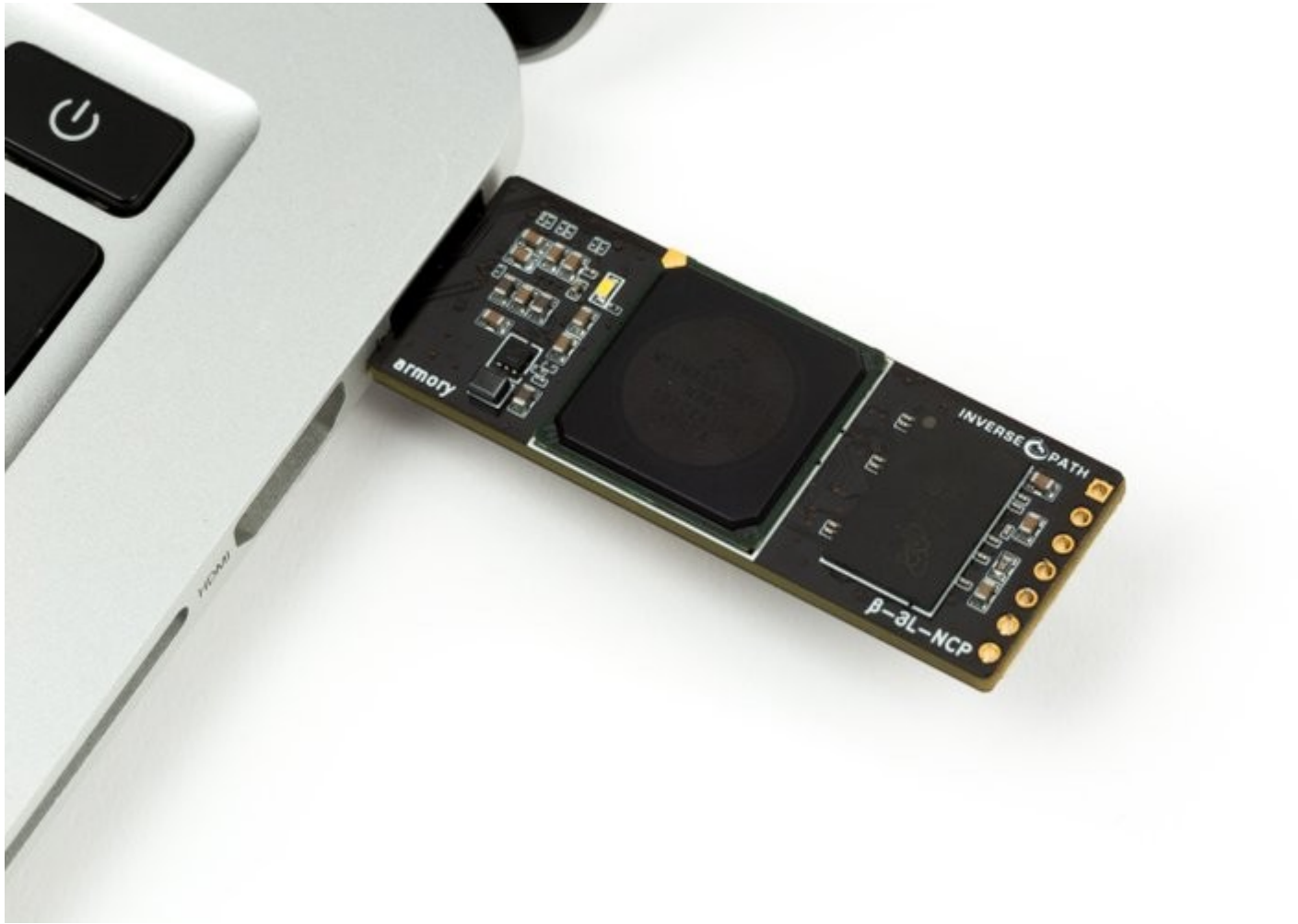
Forging the USB armory

INVERSE PATH

USB armory - Open source flash-drive-sized computer

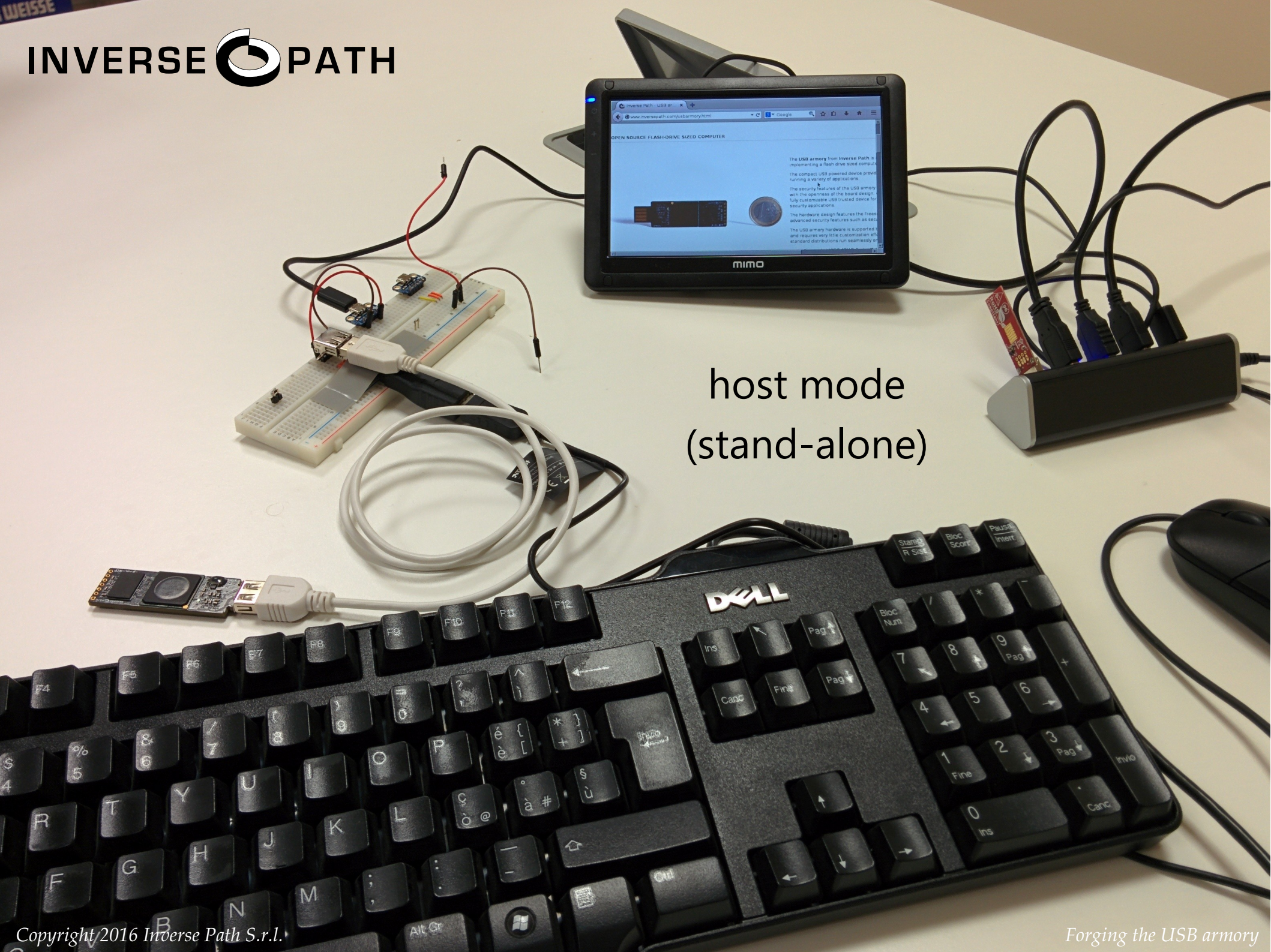
- NXP i.MX53 ARM® Cortex™-A8 800Mhz, 512MB RAM
- USB host powered (<500 mA) device with compact form factor (65 x 19 x 6 mm)
- ARM® TrustZone®, secure boot + storage + RAM
- microSD card slot
- 5-pin breakout header with GPIOs and UART
- customizable LED, including secure mode detection
- excellent native support:

  Debian, Ubuntu, Arch Linux ARM, Genode OS (w/ TZ)
- USB device emulation (CDC Ethernet, mass storage, HID, etc.)
- Open Hardware & Software

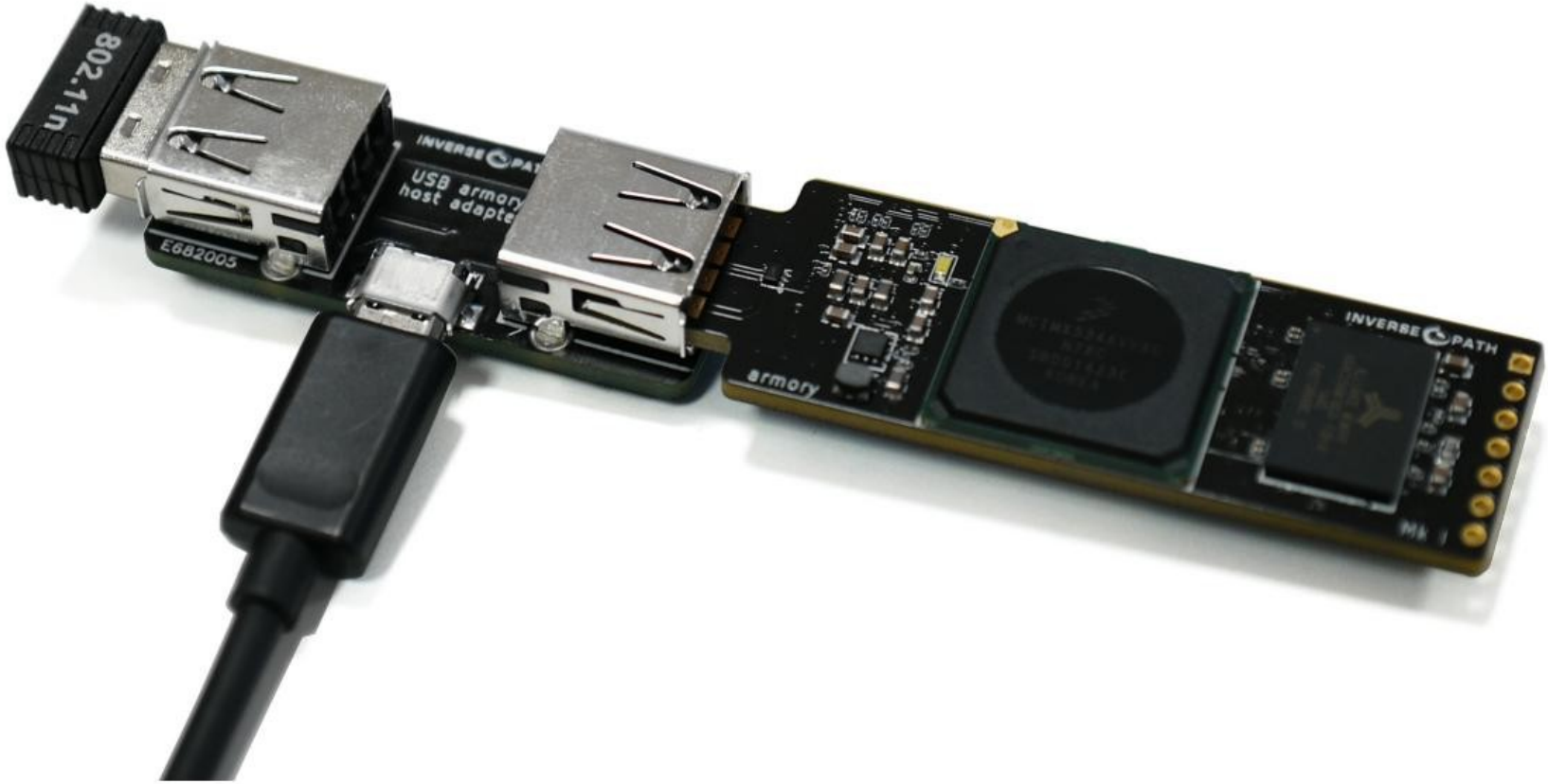


device mode

INVERSE PATH



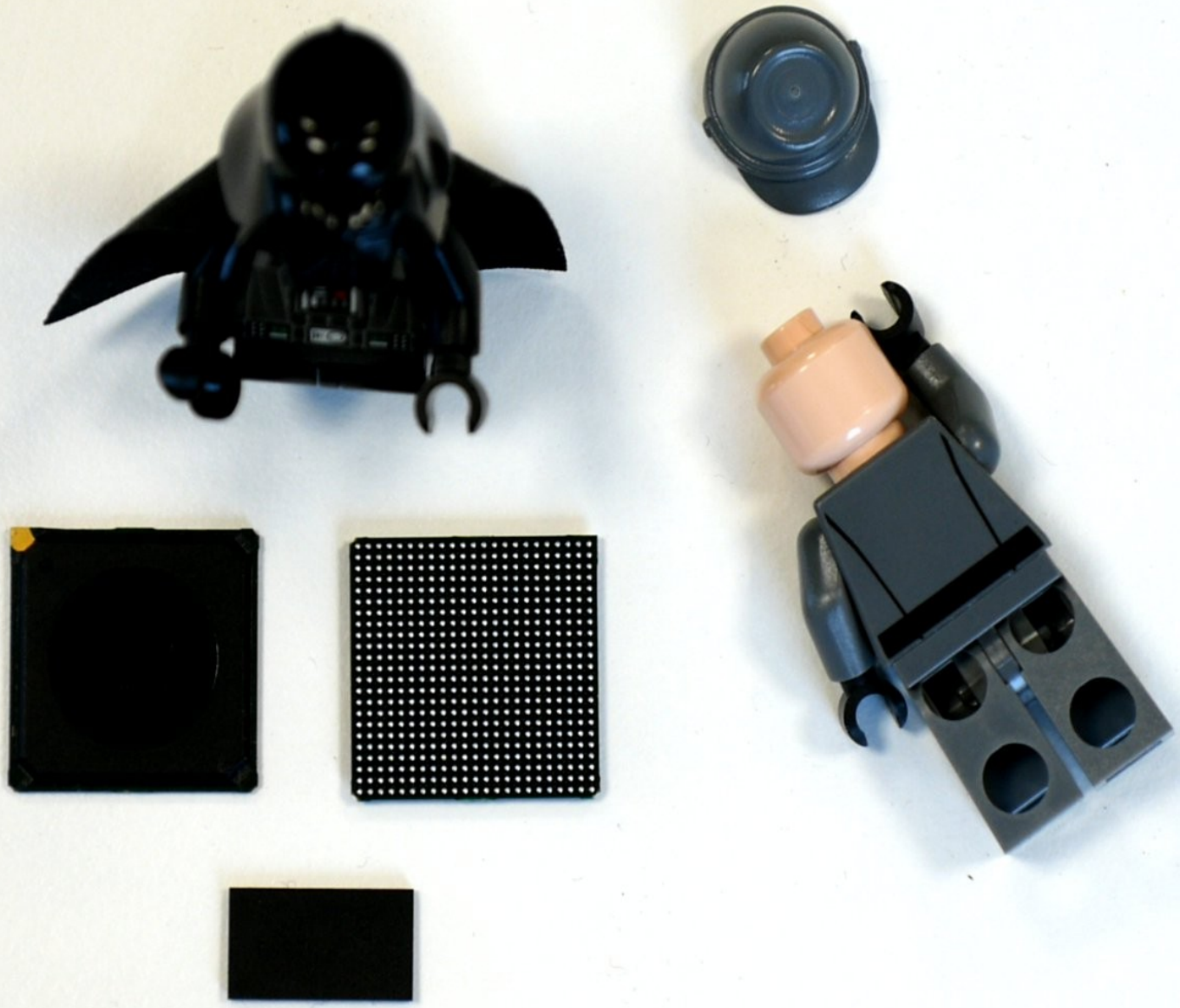
INVERSE PATH



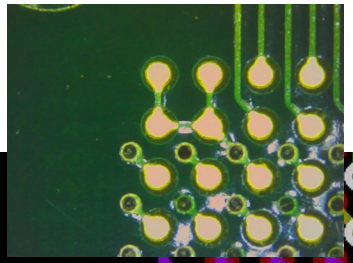
INVERSE PATH

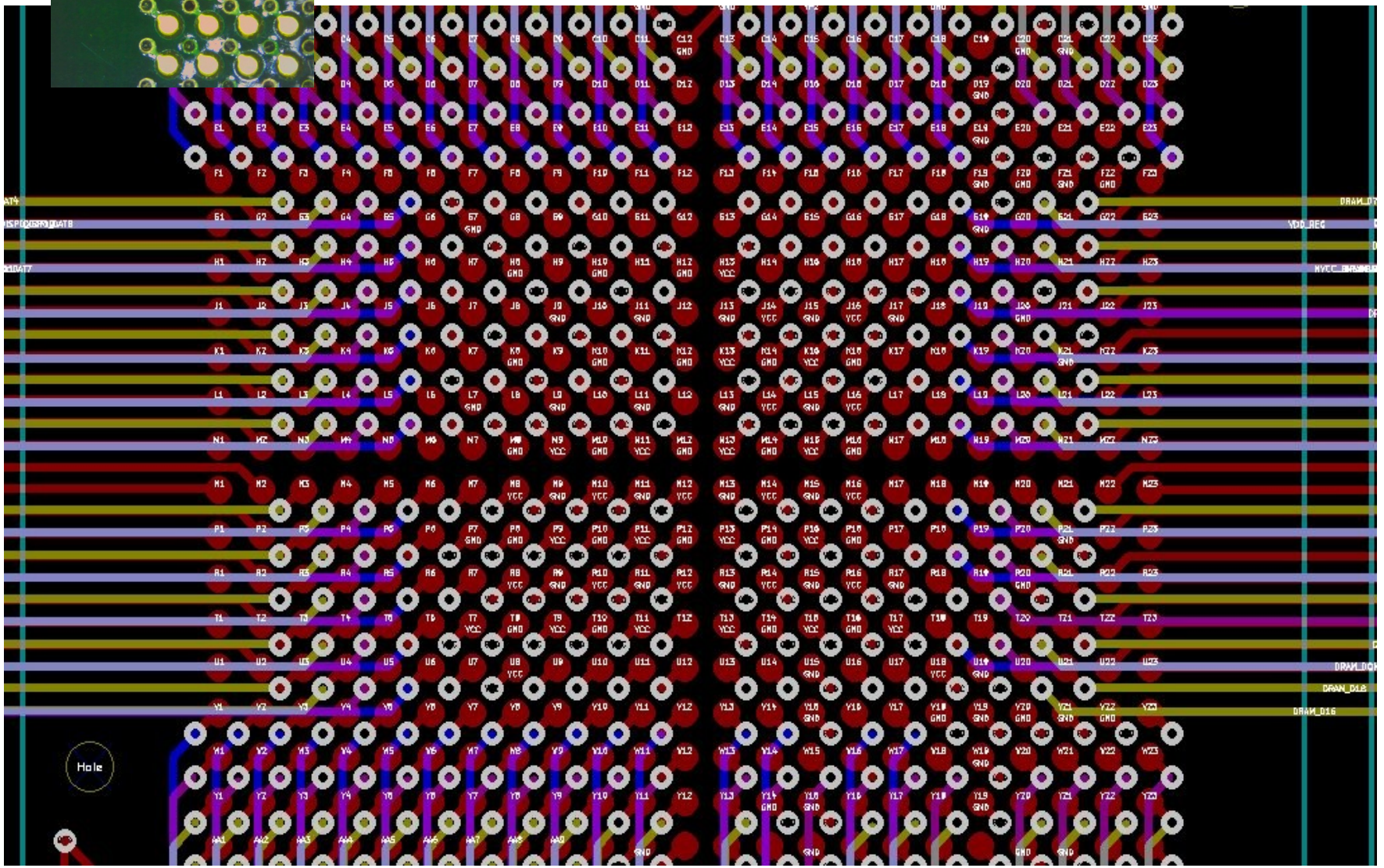
inversepath.com/usbarmory

INVERSE PATH

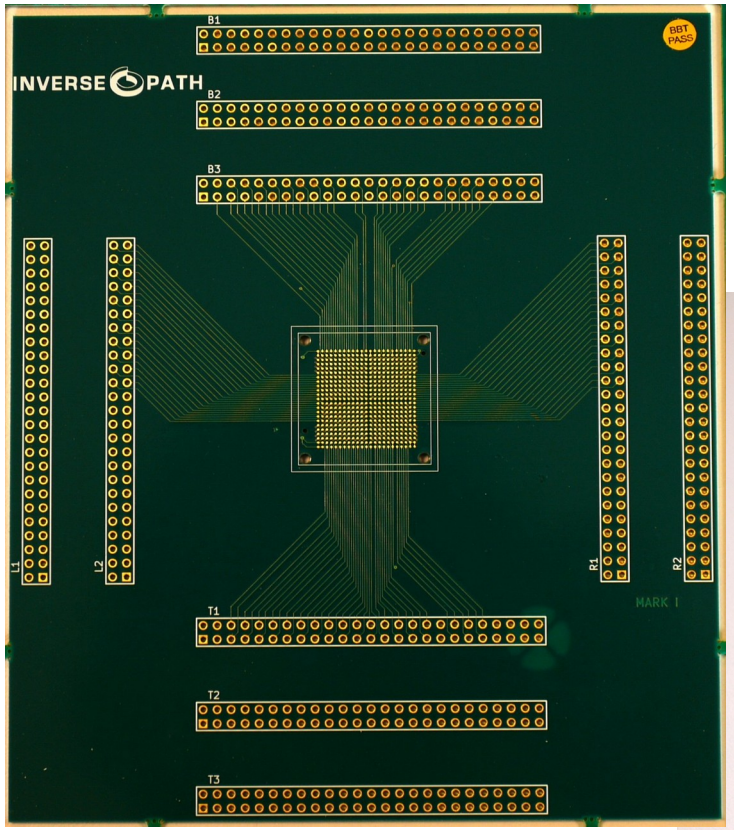


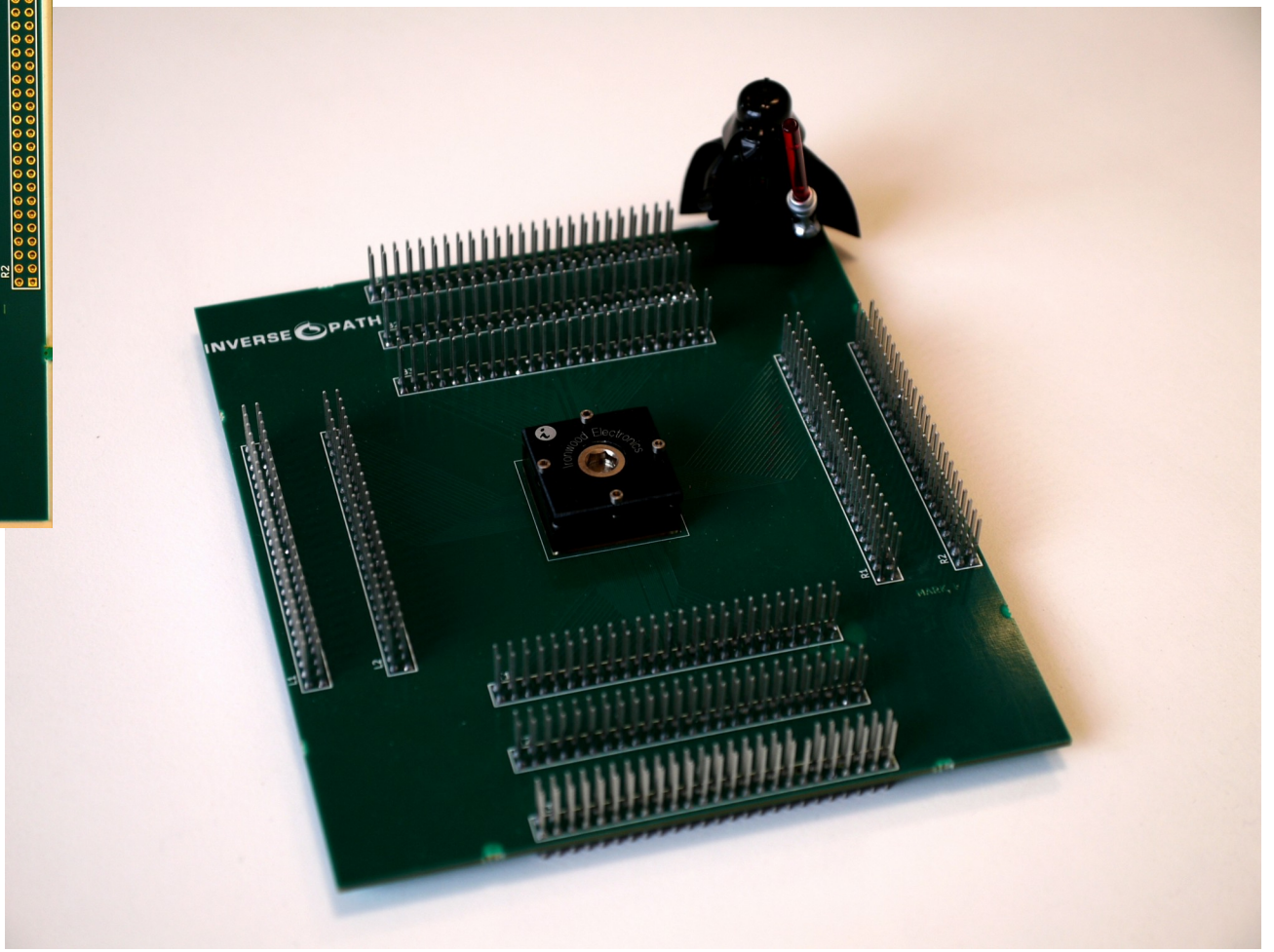
INVERSE PATH



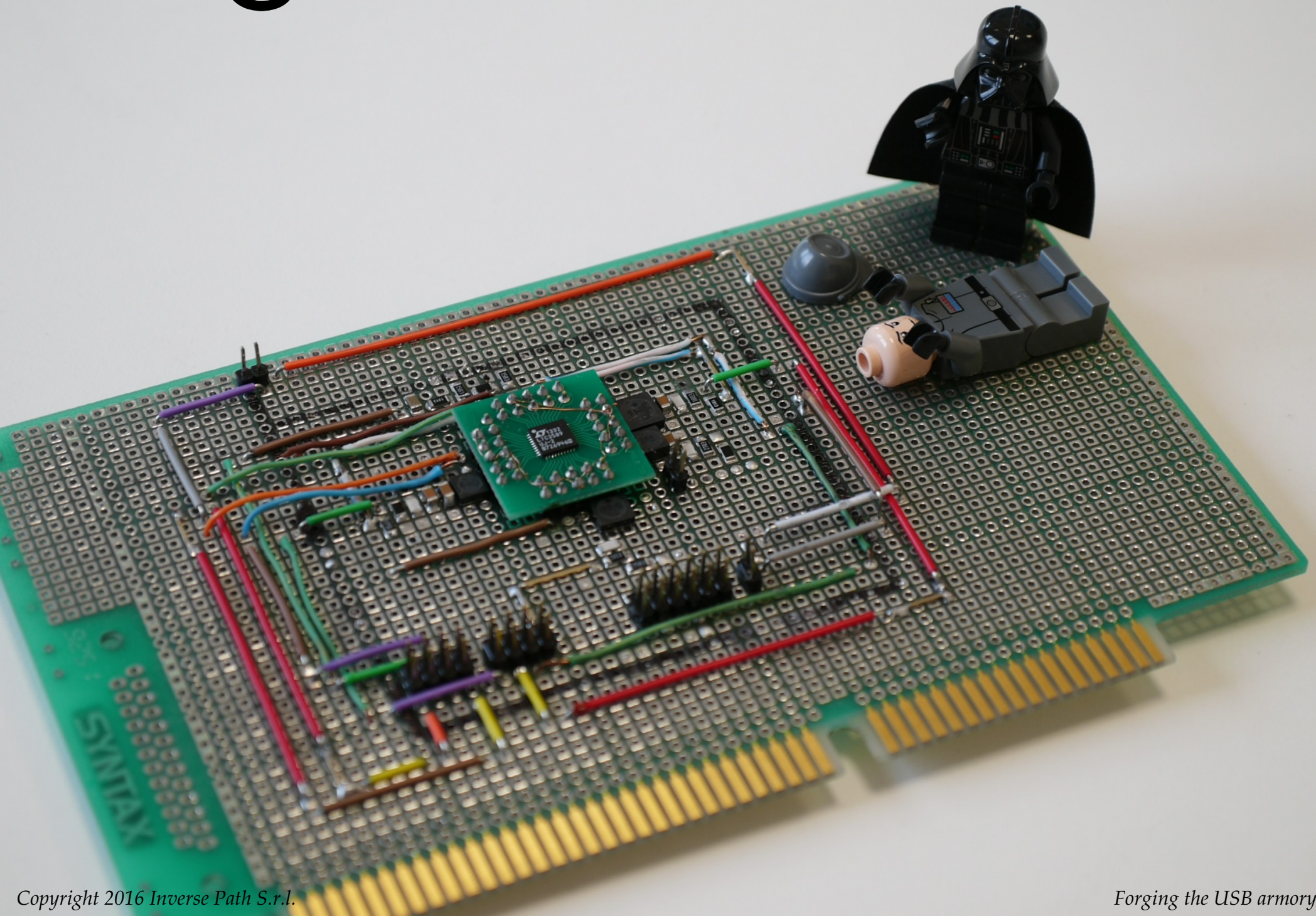


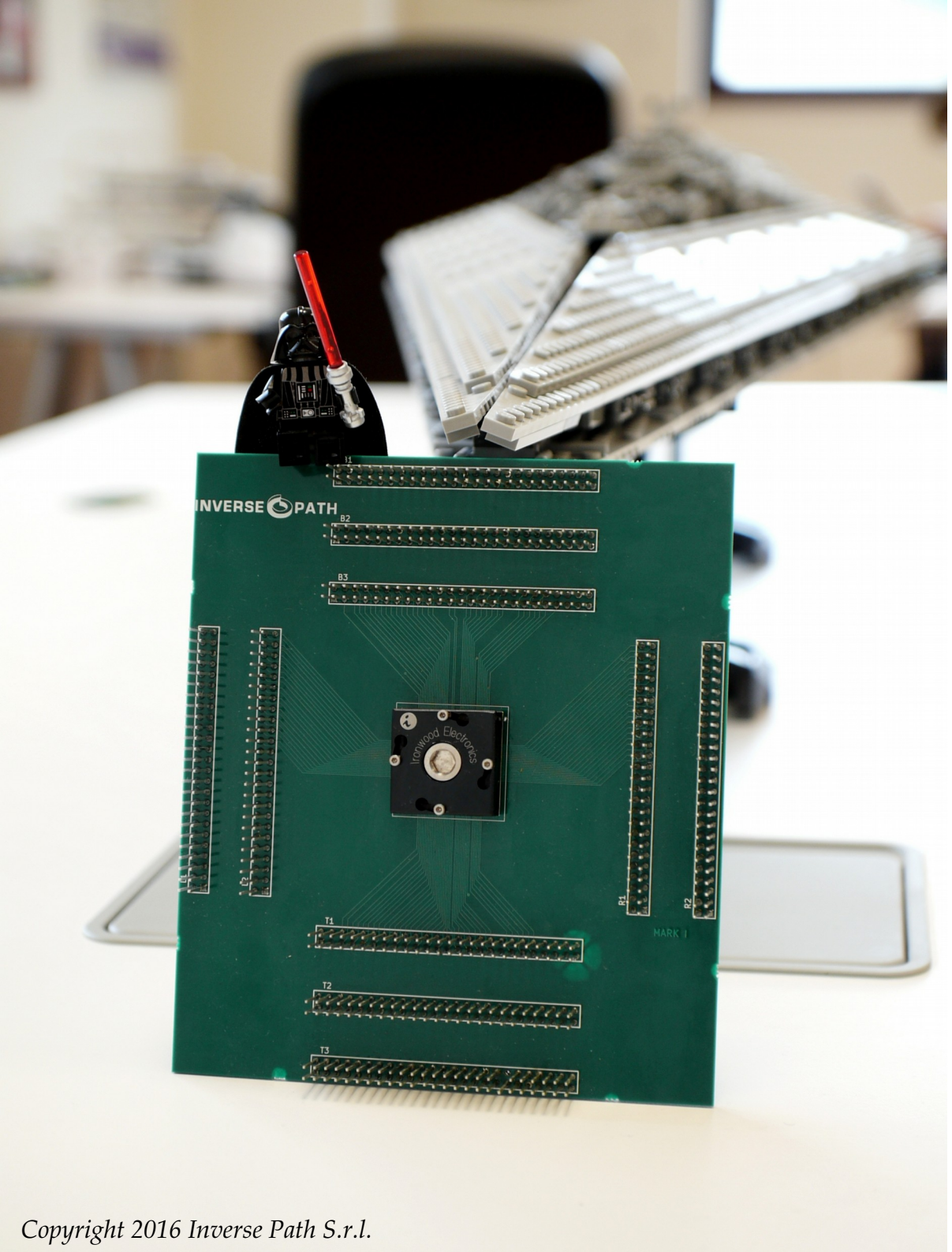
INVERSE PATH





INVERSE PATH

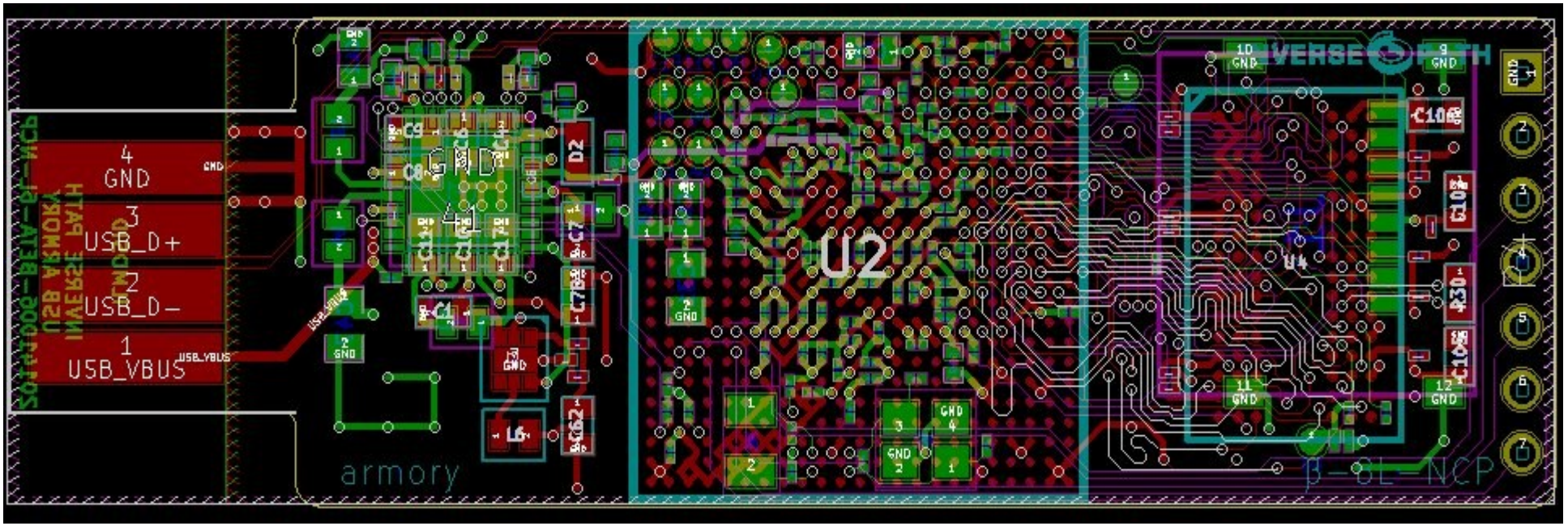


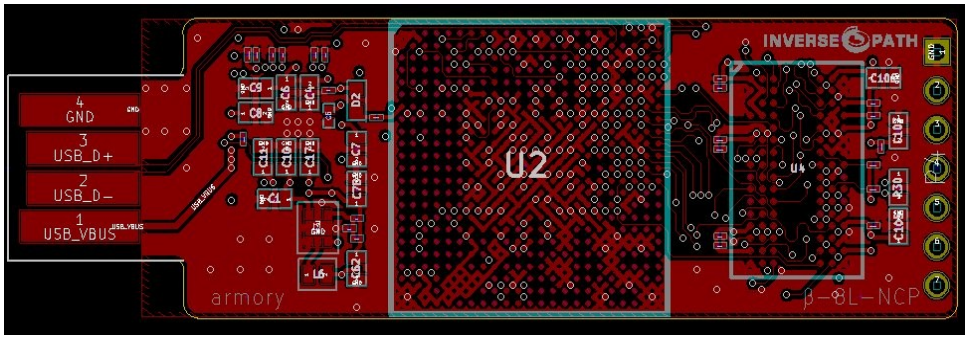


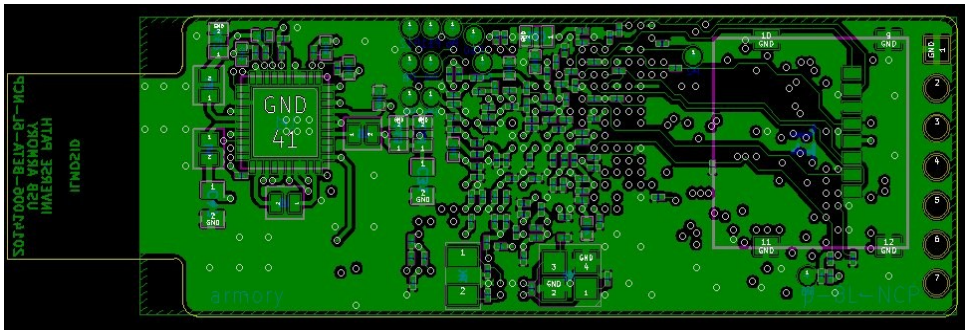
INVERSE PATH

Forging the USB armory

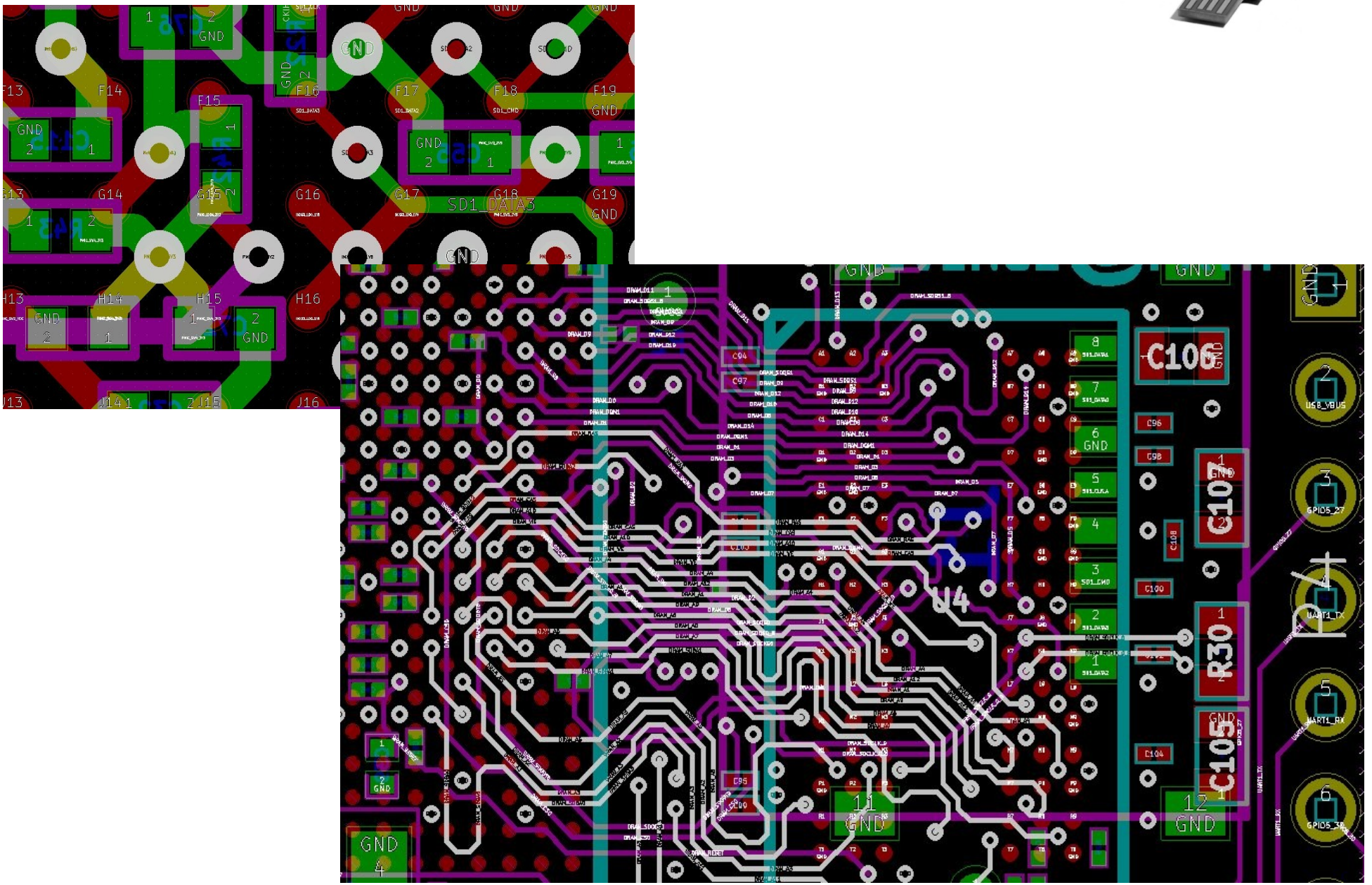
INVERSE PATH



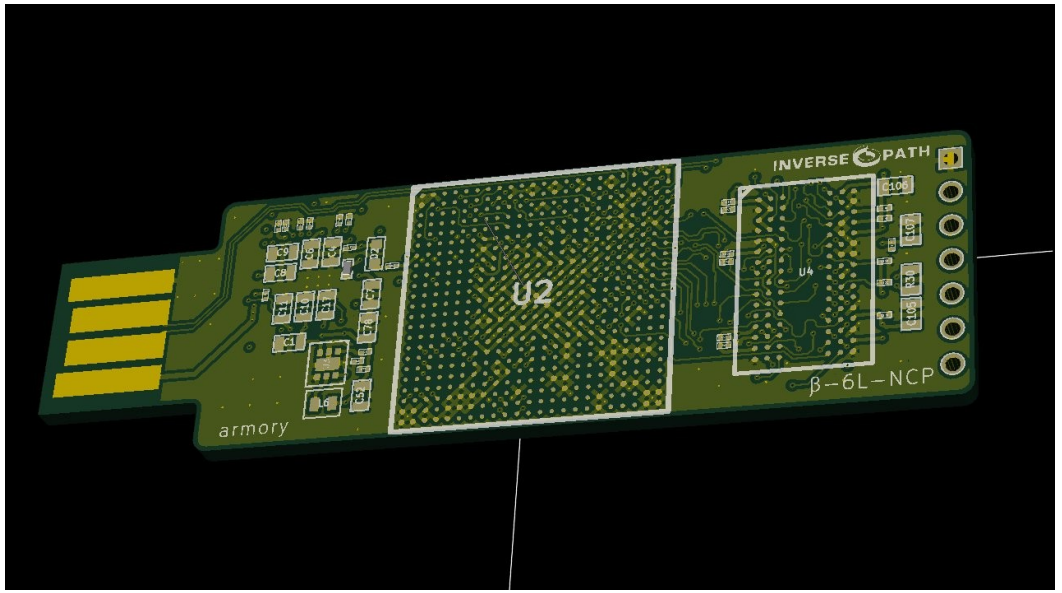


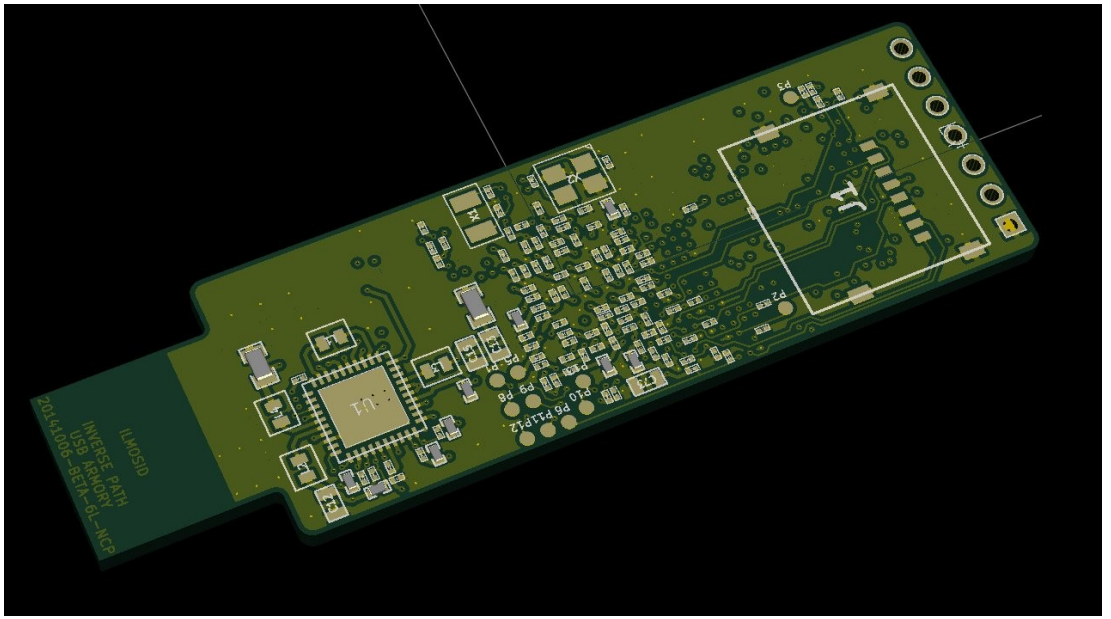


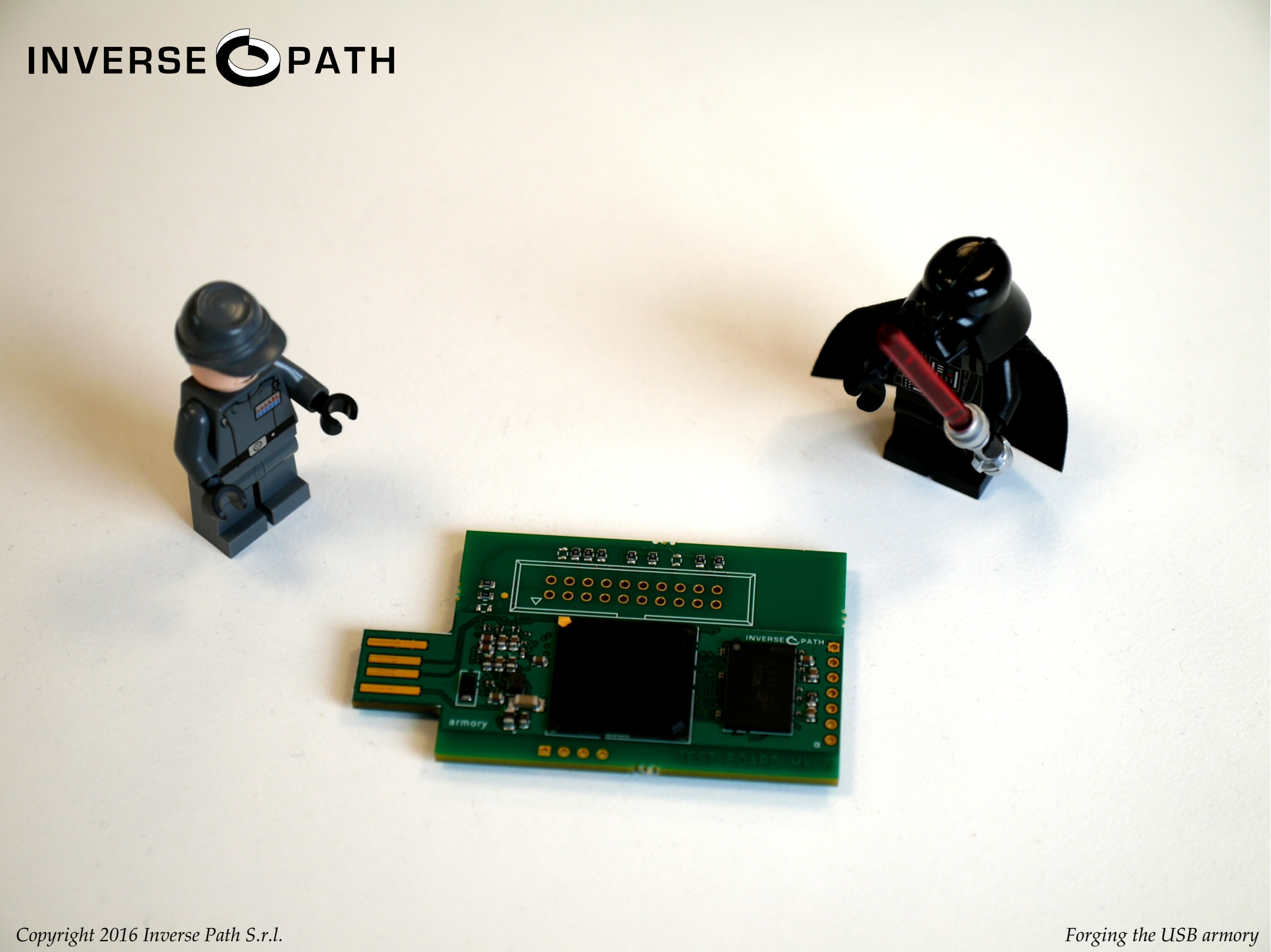
INVERSE PATH

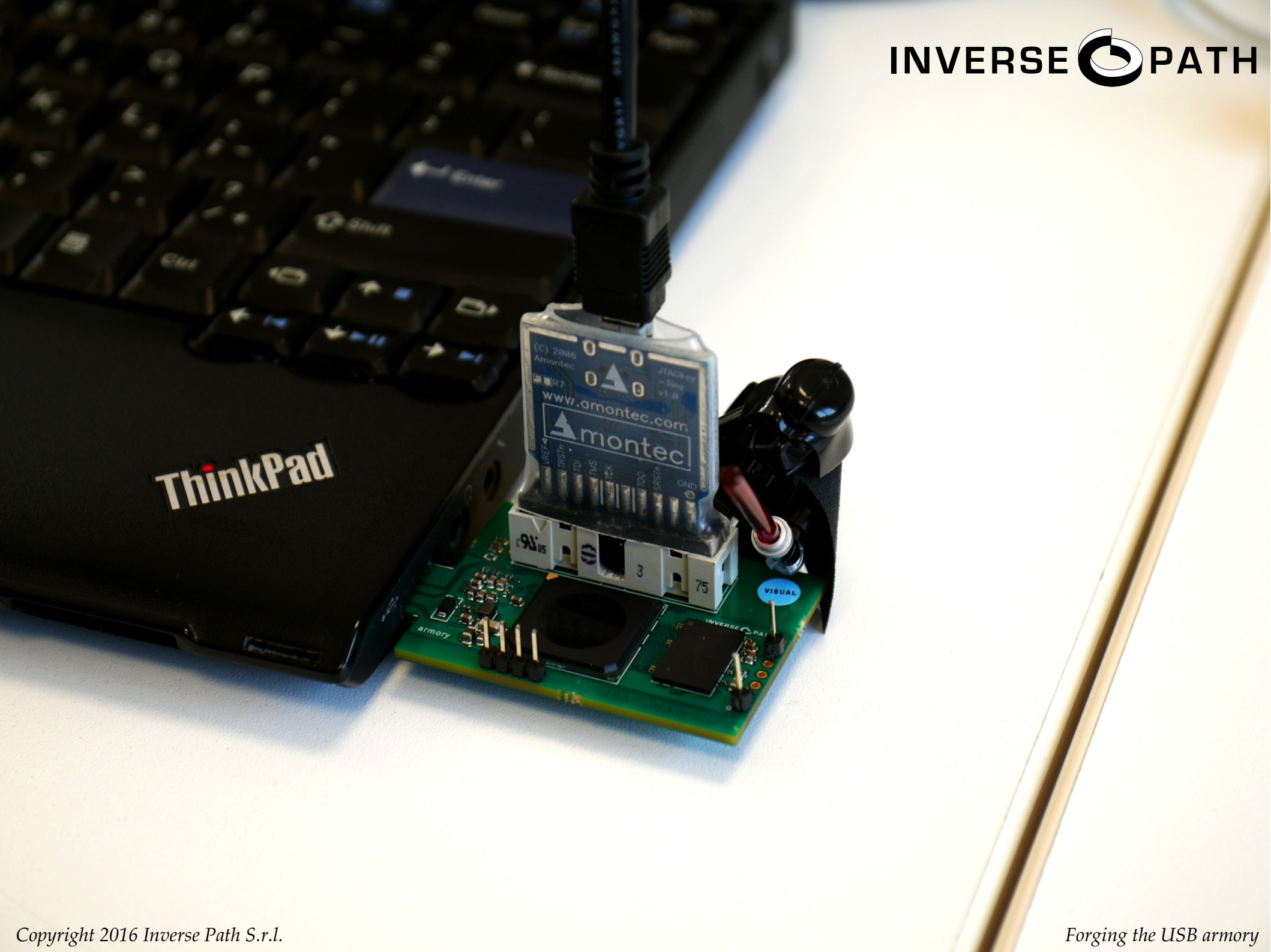


INVERSE PATH









INVERSE PATH

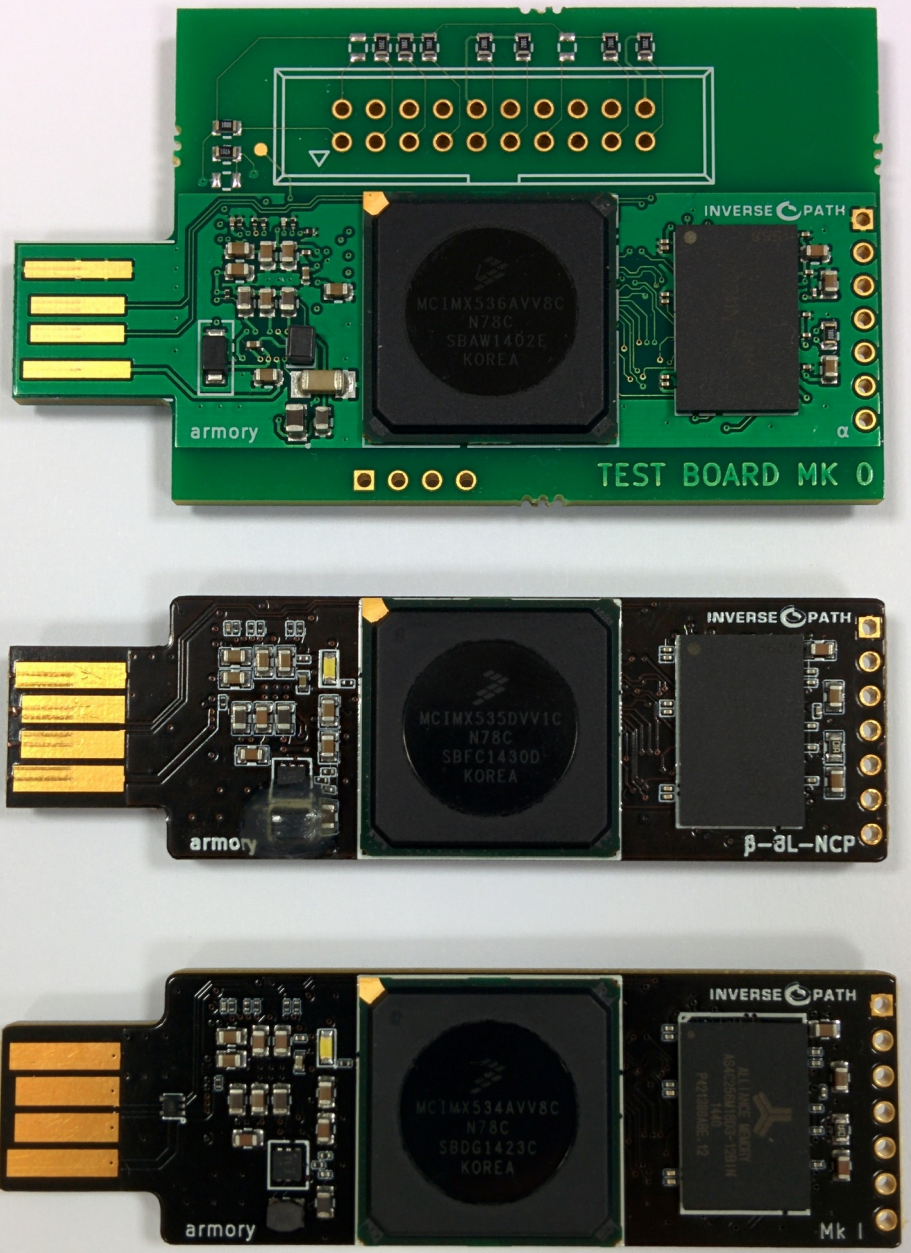
α

βs

8L-NOUSBH,8L, 8L-DDR-LDO, 8L-DDR-NCP

6L, 6L-DDR-LDO, 6L-DDR-NCP

Mk I



Forging the USB armory

Secure Boot

The USB armory SoC supports High Assurance Boot (HABv4), enabling boot image verification.

Up to four public keys (SRK) are used to generate a SHA256 hash for verification, the hash is fused on the SoC with a permanent, irreversible operation.

Unlike Secure Boot on modern PCs the activation can not be reset. This is a feature, not a bug.

Up to 3 keys out of 4 can be revoked.

Fuse name	IIM bank	IIM addr[bits]	Function
SRK_HASH[255:248]	1	0x0c04	SRK table hash (part 1)
SRK_HASH[247:160]	3	0x1404-0x142c	SRK table hash (part 2)
SRK_HASH[159:0]	3	0x1430-0x147c	SRK table hash (part 3)
SRK_LOCK	1	0x0c00[2]	lock for SRK_HASH[255:248]
SRK_LOCK88	3	0x1400[1]	lock for SRK_HASH[247:160]
SRK_LOCK160	3	0x1400[0]	lock for SRK_HASH[159:0]
SRK_REVOKE[2:0]	4	0x1810[2:0]	SRK keys revocation
SEC_CONFIG[1:0]	0	0x0810[1:0]	Security configuration
DIR_BT_DIS[1:0]	0	0x0814[0]	Direct external memory

```
# syntax: fuse prog [-y] <bank> <word> <hexval> [<hexval>...]
          program 1 or several fuse words, starting at 'word'
         (PERMANENT)
```

```
=> fuse prog -y 1 0x1 0xaa
=> fuse prog -y 3 0x1 0xbb 0xcc 0xdd 0xee 0xff 0xaa 0xbb 0xcc 0xdd 0xee 0xff
=> fuse prog -y 3 0xc 0xaa 0xbb 0xcc 0xdd 0xee 0xff 0xaa 0xbb 0xcc 0xdd 0xee
=> fuse prog -y 3 0x17 0xff 0xaa 0xbb 0xcc 0xdd 0xee 0xff 0xaa 0xbb
```

The U-Boot image can be signed with NXP Code Signing Tool from NXP (IMX_CST_TOOL).

Alternatively an OSS implementation is available in the USB armory repository.

Links:

https://github.com/inversepath/usbarmory/wiki/Secure-boot-(with-NXP-tools)

https://github.com/inversepath/usbarmory/wiki/Secure-boot

https://github.com/inversepath/usbarmory/tree/master/software/secure_boot

Hash generation:

```
$ ./usbarmory_srktool -h
Usage: usbarmory_srktool [OPTIONS]
  -1 | --key1  <public key path>     SRK public key 1 in PEM format
  -2 | --key2  <public key path>     SRK public key 2 in PEM format
  -3 | --key3  <public key path>     SRK public key 3 in PEM format
  -4 | --key4  <public key path>     SRK public key 4 in PEM format
  -o | --hash  <output filename>     Write SRK table hash to file
  -O | --table <output filename>     Write SRK table to file
```

Bootloader signing:

```
$ ./usbarmory_csftool -h
Usage: usbarmory_csftool [OPTIONS]
  -A | --csf_key <private key path>  CSF private key in PEM format
  -a | --csf_crt <public  key path>  CSF public  key in PEM format
  -B | --img_key <private key path>  IMG private key in PEM format
  -b | --img_crt <public  key path>  IMG public  key in PEM format
  -I | --table   <SRK table path>    Input SRK table (see usbarmory_srktool -O)
  -x | --index   <SRK key index>     Index for SRK key (1-4)
  -i | --image   <filename>          Image file w/ IVT header (e.g. u-boot.imx)
  -o | --output  <filename>          Write CSF to file
```

Secure Boot + Verified Boot

The U-Boot bootloader supports cryptographic verification of signed kernel images. A public key can be embedded in the bootloader image to verify the Linux kernel.

```
make ARCH=arm EXT_DTB=pubkey.dtb
```

The chain of trust, up to the kernel image, authenticates all executed code with user controlled certificates.

Secure Boot

```
U-Boot 2015.07 (Sep 10 2015 - 14:26:37 +0200)

CPU:   Freescale i.MX53 rev2.1 at 800 MHz
Reset cause: POR
Board: Inverse Path USB armory MkI
I2C:   ready
DRAM:  512 MiB
MMC:   FSL_SDHC: 0
In:    serial
Out:   serial
Err:   serial
Net:   CPU Net Initialization Failed
No ethernet found.
Hit any key to stop autoboot:  2
=> hab_status.

Secure boot enabled

HAB Configuration: 0xcc, HAB State: 0x99
No HAB Events Found!

=> boot
2301352 bytes read in 300 ms (7.3 MiB/s)
16670 bytes read in 178 ms (90.8 KiB/s)
## Booting kernel from Legacy Image at 70800000 ...
   Image Name:   Linux-4.2.0
```

Successfully booted signed U-Boot image.

Secure Boot

```
U-Boot 2015.07 (Sep 10 2015 - 14:26:37 +0200)

CPU:   Freescale i.MX53 rev2.1 at 800 MHz
Reset cause: POR
Board: Inverse Path USB armory MkI
I2C:   ready
DRAM:  512 MiB
MMC:   FSL_SDHC: 0
In:    serial
Out:   serial
Err:   serial
Net:   CPU Net Initialization Failed
No ethernet found.
Hit any key to stop autoboot:  2
=> hab_status.

Secure boot enabled

HAB Configuration: 0xf0, HAB State: 0x66

--------- HAB Event 1 -----------------
event data: ...

STS = HAB_FAILURE (0x33)
RSN = HAB_INV_SIGNATURE (0x18)
```

Failed attempt shown in verification mode, before final activation.

When active, failures hang the SoC without any printed info (this is a feature, not a bug).

Verified Boot

```
## Loading kernel from FIT Image at 70800000 ...
   Using 'conf@1' configuration
   Verifying Hash Integrity ... sha256,rsa2048:usbarmory+ sha256,rsa2048:usbarmory+ OK
   Trying 'kernel@1' kernel subimage
     Description:  Vanilla Linux kernel
     Type:         Kernel Image (no loading done)
     Compression:  uncompressed
     Data Start:   0x708000cc
     Data Size:    8724448 Bytes = 8.3 MiB
     Hash algo:    sha256
     Hash value:   c5949bf26f792c62fcd793f70041397240d8a17dab240f0e6b71f72dc59298b2
   Verifying Hash Integrity ... sha256+ OK
## Loading fdt from FIT Image at 70800000 ...
   Using 'conf@1' configuration
   Trying 'fdt@1' fdt subimage
     Description:  USB armory devicetree blob
     Type:         Flat Device Tree
     Compression:  uncompressed
     Data Start:   0x710521c4
     Data Size:    16670 Bytes = 16.3 KiB
     Architecture: ARM
     Hash algo:    sha256
     Hash value:   8d60182befa80d8ab8ebd4fce490a2226acd473ba815b578518867d9eeb71aaf
   Verifying Hash Integrity ... sha256+ OK
   Booting using the fdt blob at 0x710521c4
   XIP Kernel Image (no loading done) ... OK
   Loading Device Tree to 8f54d000, end 8f55411d ... OK
```

Security Controller (SCCv2)

The SCCv2 is a built-in hardware module that implements secure RAM and a dedicated AES cryptographic engine for encryption/decryption operations.

A device specific random 256-bit SCC key is fused in each SoC at manufacturing time, this key is unreadable and can only be used with the SCCv2 for AES encryption/decryption of user data.

The SCCv2 internal key is available only when Secure Boot (HAB) is enabled, otherwise the AES-256 NIST standard test key is used.

Useful to derive device-specific secrets for FDE.

Security Controller (SCCv2)

https://github.com/inversepath/mxc-scc2

```
$ sudo modprobe scc2
$ sudo modprobe scc2_aes

SCC2_AES: Secure State detected
```

Ruby example:

```
scc = File.open(“/dev/scc2_aes”, "r+")

# encryption
scc.ioctl(SET_MODE, ENCRYPT_CBC)
scc.ioctl(SET_IV, iv)

scc.write(plaintext)
ciphertext = scc.read(plaintext.size)

# decryption
scc.ioctl(SET_MODE, DECRYPT_CBC)
scc.ioctl(SET_IV, iv)

scc.write(ciphertext)
plaintext = scc.read(ciphertext.size)
```

INVERSE PATH

INTERLOCK

https://github.com/inversepath/interlock

Open source file encryption front-end developed, but not limited to, usage with the USB armory.

Provides a web accessible file manager to unlock/lock LUKS encrypted partition and perform additional symmetric/asymmetric encryption on stored files.

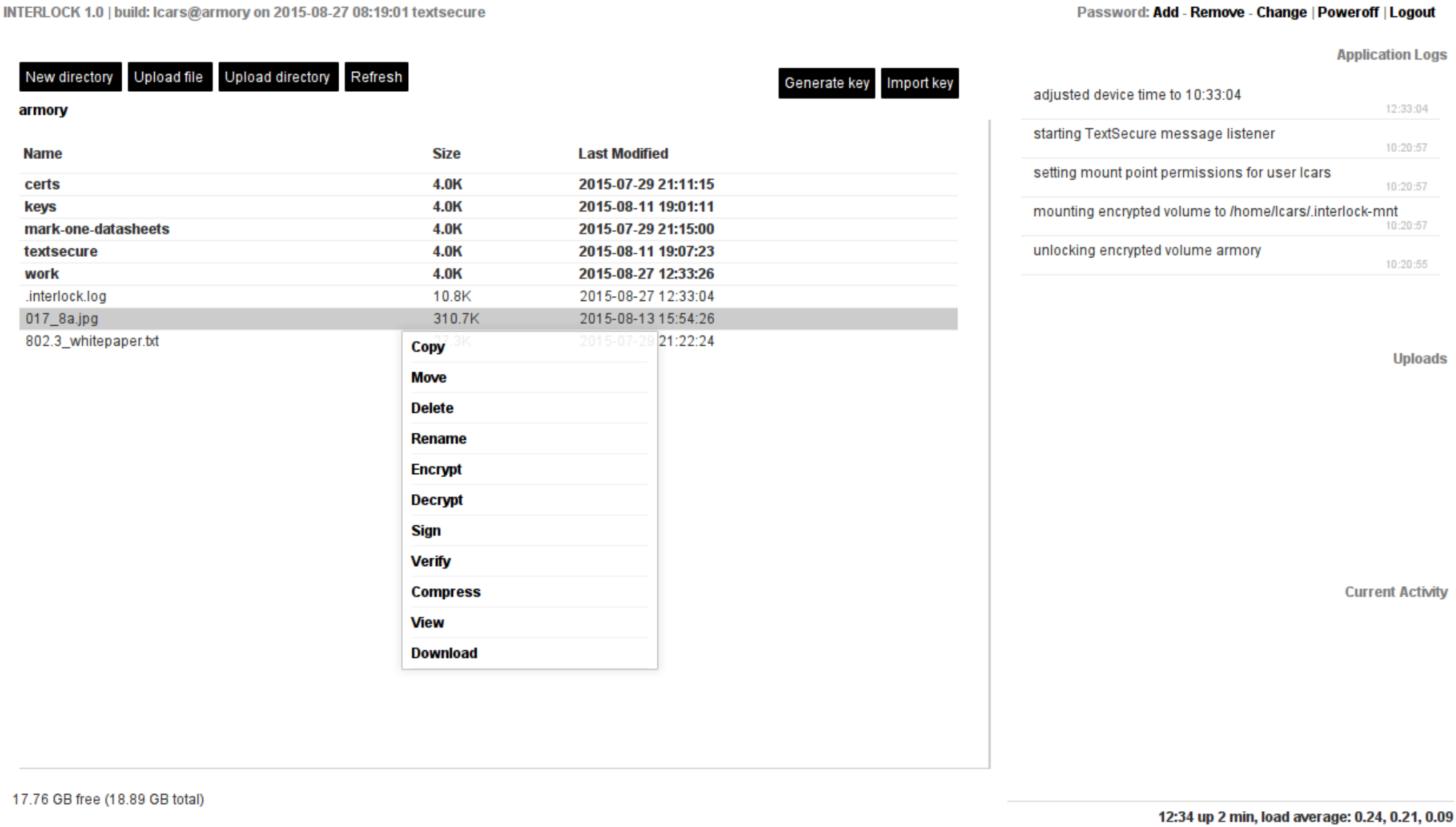
Takes advantage of disposable passwords.

Design Goals

Clear separation between presentation and server layer to ease auditability and integration.

Minimum amount of external dependencies and footprint.

Encrypted volumes:	LUKS encrypted partitions
Asymmetric ciphers:	OpenPGP
Symmetric ciphers:	AES-256-OFB w/ PBKDF2 + HMAC
Security tokens:	TOTP (Google Authenticator)
Messaging:	Signal
HSM:	SCCv2 for device specific LUKS password, AES-256-SCC cipher, protected TLS key

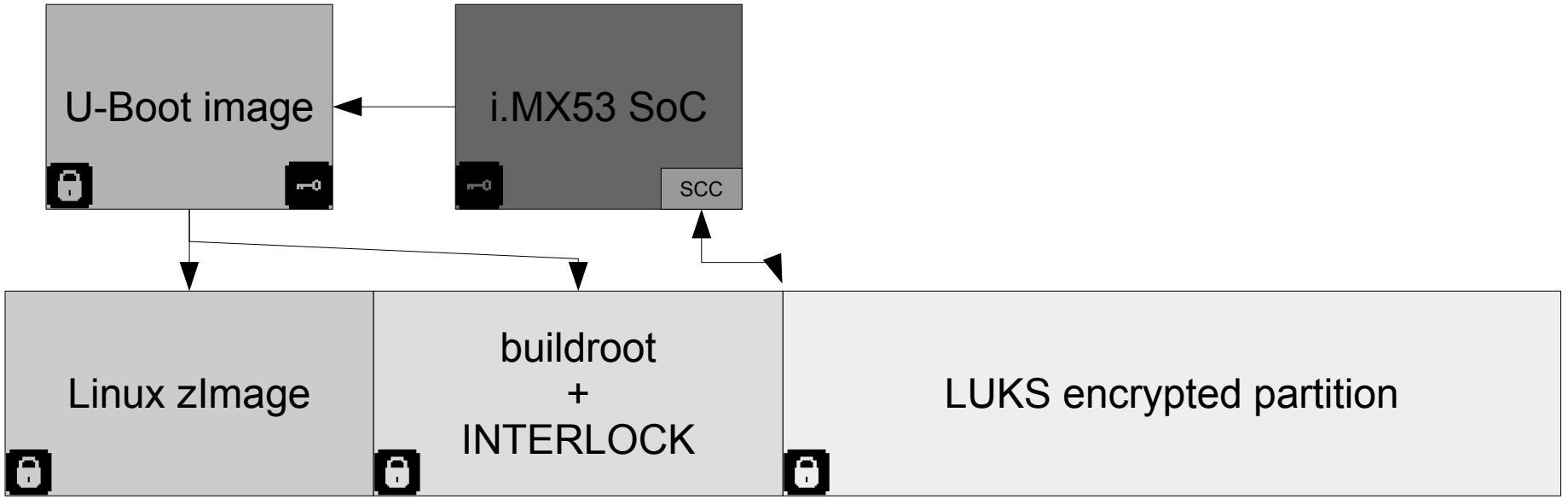


Embedded INTERLOCK distribution

A customized Buildroot environment allows compilation of bootloader and Linux kernel with embedded rootfs exposing INTERLOCK with automatic encrypted partition setup.

```
make BR2_EXTERNAL=${USBARMORY_GIT}/software/buildroot usbarmory_mark_one_defconfig
make BR2_EXTERNAL=${USBARMORY_GIT}/software/buildroot # yes, it's that easy!
```

The chain of trust is assured by secure + verified boot and SCCv2.



"Offensive" uses

USB descriptors + drivers manipulation/fuzzing, passive sniffing, DNS hijacking and traffic diversion.

Some nice papers involving the USB armory:

Jeroen van Kessel, Nick Petros Triantafyllidis
"USB Armory as an Offensive Attack Platform"

Roland Schilling, Frieder Steinmetz
"USB devices phoning home"

Matthias Neugschwandtner, Anton Beitler, Anil Kurmus
"A Transparent Defense Against USB Eavesdropping Attacks"

CVE-2016-1734

Malicious RNDIS device emulation.

Found and investigated on OS X.

Applies to iOS < 9.3.

Works on locked session w/o user intervention.

iOS 9.3

- **AppleUSBNetworking**

  Available for: iPhone 4s and later, iPod touch (5th generation) and later, iPad 2 and later

  Impact: An application may be able to execute arbitrary code with kernel privileges

  Description: A memory corruption issue existed in the parsing of data from USB devices. This issue was addressed through improved input validation.

  CVE-ID

  CVE-2016-1734 : Andrea Barisani and Andrej Rosano of Inverse Path

CVE-2016-1734

A high-speed USB device, advertising specific descriptors which match full-speed values, causes OS X to panic and reboot upon specific messages sent from the device towards the host.

The memory corruption is triggered regardless of the user being logged in or not.

```
bInterval      set to 32           in the Interrupt type Endpoint Descriptors
wMaxPacketSize set to 1 x 64 bytes in the Bulk type Endpoint Descriptors

These values are incorrect as normally bInterval is 9 and wMaxPacketSize is
512 bytes on high-speed devices.
```

CVE-2016-1734

A panic report can be triggered by sending a specifically crafted network packet from a RNDIS/Ethernet device (USB armory).

While a single (IPv6) packet can be used as reproducer the panic is not always generated as the resulting effect depends on the transmitted payload.

```
*** Panic Report ***
panic(cpu 0 caller 0xffffff8005f6bc0c): Failed mbuf validity check: mbuf
0xffffff809ac55400 len -8 type 1 flags 0x2 data 0xffffff809ac554b8 rcvif en8
ifflags 0x8863
Backtrace (CPU 0), Frame : Return Address
0xffffff809f60be20 : 0xffffff8005ce5307
0xffffff809f60bea0 : 0xffffff8005f6bc0c
0xffffff809f60bf60 : 0xffffff8005f6e819
0xffffff809f60bfb0 : 0xffffff8005dd15d7
```

In cases where the panic is not triggered, and OS X “survives”, incoming packets are received misaligned/overlapped.

For instance the payload of previously transmitted packets can be seen as header of following ones, generating invalid frames. This aspect is clear indication of packet buffer misalignment.

mbuf exploitation over IPv6 on BSD systems reminds of the following OpenBSD issue:

https://www.coresecurity.com/content/open-bsd-advisorie

Regardless of the impact, the finding makes a good case for the efficiency of using the USB armory for such testing. The ability to manipulate descriptors as well as presenting a full kernel driver enhances testing possibilities.

```
static struct usb_interface_descriptor ms_interfacedesc = {
     ...
     .bLength = USB_DT_ENDPOINT_SIZE,
     .wMaxPacketSize = cpu_to_le16(512),
     ...
};

static int do_inquiry(struct fsg_common *common, struct fsg_buffhd *bh) {
     struct fsg_lun *curlun = common->curlun;
     u8      *buf = (u8 *) bh->buf;

     if (!curlun) {
          common->bad_lun_okay = 1;
          memset(buf, 0, 36);
          buf[0] = TYPE_NO_LUN;
          buf[4] = 31;            /* Additional length */
          return 36;
     }
     ...
```

INVERSE PATH

Thank you!

info@inversepath.com
https://inversepath.com/usbarmory
https://github.com/inversepath/usbarmory
https://github.com/inversepath/interlock