Embedded Linux Integrity

David Safford

6/30/2013

Abstract

Linux is in widespread use in embedded devices, but these devices typically lack critical security features found in higher-end Linux systems. They typically do not have any way to validate their firmware, they do not have hardware roots of trust for trusted or secure boot, they do not have provisions for physical presence, to protect firmware from remote modification, and they do not have secure update. Vendors claim that these features are either too large, or too expensive to fit in their embedded devices.

This paper summarizes the recent widespread vulnerabilities and compromises of embedded devices, and shows how the given security features would defeat such attacks. It relates the concepts to the NIST SP800 guidelines for BIOS measurement and protection, and to the ongoing work on Linux secure boot for higher end devices. It looks at four typical embedded devices, shows how all of these features can be added at zero cost.

Introduction

Linux runs on an incredible range of devices from very small embedded devices, to the largest supercomputers. The devices cover a staggering 12 orders of magnitude in memory size, and 7 orders of magnitude in cost.

Embedded Linux devices typically consist of just three small chips – an SoC, flash, and RAM. The SoC (System on a Chip) normally includes a 32 bit ARM or MIPS CPU, along with flash, RAM, USB, ethernet and wireless interfaces. The flash is typically a 4 or 8MB SPI device, and the RAM is usually 32 – 64 MB. The firmware on these small devices includes a Linux kernel, stripped and compressed to under 1MB, and a squashed root filesystem under 3 MB. There is no initramfs.

For this class of embedded devices we are mainly interested in four foundational integrity features:

- initial firmware validation
- run-time firmware protection
- firmware update validation
- boot time integrity validation

While higher-end Linux devices in the mobile, PC, and server categories have one or more of these features, typical embedded devices have *none* of them. Table 1 shows some categories of Linux devices, and their typical integrity features.

Category	Cost	Size	Typical Integrity Features
Server	$10K+	PB	4768 Crypto card Trusted and Secure Boot
PC	$1K	TB	TPM Trusted and Secure Boot (Win8)
mobile	$500	GB	Restricted Boot
embedded	$50	MB	Nothing
Sensor	$10	KB	Nothing

Table 1: Linux Spectrum

This material is based on research sponsored by the Department of Homeland Security (DHS) Science and Technology Directorate, Homeland Security Advanced Research Projects Agency, Cyber Security Division (DHS S&T/HSARPA/CSD), BAA 11-02 and Air Force Research Laboratory, Information Directorate under agreement number FA8750-12-2-0243. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Department of Homeland Security, Air Force Research Laboratory or the U.S. Government.

For this paper, we selected four example embedded linux devices as representative: Linksys WRT54G, TP-Link MR3020, D-Link DIR-505, and the Pogoplug model 2.

The WRT54g was the first wireless access point to run embedded Linux. Linksys published all the GPL source code to this device, and today virtually all wireless devices use a derivative of this original embedded Linux. Openwrt is a community supported derivative that runs on most past and current wireless devices.

The TP-link and D-link are small, travel-sized versions with significant additional features, including support for USB attached network storage and media serving. Figure 1 shows images for these four selected devices.

Figure 1: Example Devices

Embedded Linux devices like these four have been extensively compromised recently. In 2012 4.5 Million home routers were compromised in Brazil [1]. In this one attack, many different Broadcom based devices, from multiple vendors and across four ISPs were compromised, redirecting all home client devices to malicious DNS servers. While the vulnerability was present only on the internal networks, the basic exploit was so simple that using CSRF (cross site request forgery) to “bounce” the attack off local browsers was quite effective. The basic (trivial) exploit to get a plaintext copy of the admin password was:

```
  get.pl http://192.168.1.1/password.cgi
```

In 2013, the D-Link DIR-645 home router was similarly found to give away the admin password in a trivial exploit [2]:

```
  curl -d SERVICES=DEVICE.ACCOUNT
     http://<device ip>/getcfg.php
```

In 2013, five new vulnerabilities in Linksys routers were discovered[3]. The WRT54GL (one of the devices selected), was found to have a CSRF vulnerability allowing unauthenticated upload of arbitrary firmware. The EA2700 was found to have a CSRF file path traversal vulnerability which can expose all files (including /etc/passwd):

```
POST /apply.cgi
submit_button=Wireless_Basic&change_acti
on=gozila_cgi&next_page=/etc/passwd
```

It also had an interesting CSRF vulnerability returning the source code of any page. (This is one of the few known single character exploits, as adding trailing / is all that is needed.) For example:

```
http://192.168.1.1/Management.asp/
```

In 2013 researchers hacked over a dozen home routers [4] with two remote (CSRF) root exploits on the Belkin N300, and Belkin N900, and four local (WLAN) root exploits on the same Belkin devices, plus the Netgear WNDR4700.

The one-line root exploits were (N300):

```
<form name="belkinN300"
action="http://192.168.2.1/apply.cgi"
method="post"/>
```

N900 exploit:

```
<form name="belkinN900"
action="http://192.168.2.1/util_system.h
tml"
```

With so many of these devices vulnerable to such simple exploits, their integrity is at significant risk. While all of these vulnerabilities are in the web management interfaces of the devices, integrity measurement and protection mechanisms at the device level can detect and prevent successful exploits of all of these vulnerabilities.

Threat Model and Integrity Goals

The threat model we consider includes supply chain attacks, and remote software attack. Can the attacker compromise the integrity of the device's firmware – either before purchase, or over the internet once it is installed? We are not concerned with local physical attack, as this does not scale, and is quite difficult to protect against. In fact, physical presence will explicitly be trusted as part of the proposed defenses.

So what integrity features are possible and appropriate for embedded devices? One starting point is to look at the NIST guidelines for BIOS Integrity Measurement and Protection. While these guidelines were intended for PC and server class machines, they are a good starting point. Currently there are two specific guidelines: NIST-SP800-155-December-2011 BIOS Integrity Measurement (Draft) [5], and NIST-SP800-147-April-2011 BIOS Integrity Protection [6].

The first presents guidelines for "trusted boot" - the incorporation of a hardware chip as a root of trust for measuring and attesting to the integrity of the BIOS itself, and of subsequent software as it is booted and executed. Note that this guideline does not discuss "secure boot", the much more common hardware root of trust which validates signatures on software before booting.

The second guideline addresses integrity protection for the BIOS - the requirements that BIOS integrity should be protected from remote software attack, that any updates need to be either authenticated, or done in some physically protected local manner, and that the mechanisms for protection must not be by-passable.

From these guidelines we can derive four related integrity features desirable in the embedded device category. For this class of embedded devices we are mainly interested in four foundational integrity features (the NIST guideline terminology is in parenthesis):

- initial firmware validation
  ("BIOS measurement")
- run-time firmware protection
  ("BIOS protection")
- firmware update validation
  ("Secure/local updates")
- boot time integrity verification
  ("Secure boot");

Table 2 shows the four sample embedded devices, and that most of these requirements are not met.

Device	Measure BIOS?	Lock BIOS?	Secure-local updates?	Secure Boot?
Pogoplug	Yes - SATA	No	No	No
D-Link DIR-505	No	No	No	No
TP-Link MR3020	No	No	No	No
Linksys WRT54G	Yes - JTAG	No	No	No

Table 2: Initial Integrity Features

The first exception was that the WRT54GL does have a JTAG interface through which a user can read (and write) the firmware in the flash chip. Normally JTAG interfaces and corresponding software are quite expensive and complex, but the WRT community has articles showing how to do this with free software and roughly $10 for a parallel port connecting cable.

The second exception is that the pogoplug is based on a SoC with the built-in ROM code to boot from a SATA disk drive, and articles show

how to boot firmware inspection tools securely from a trusted SATA disk image.

What can be done to cover all of the desired functions in all of the example devices? One problem is that vendors say that these features are simply too large (they won't fit in flash/RAM), or are too expensive (adding a $0.75 TPM chip is simply not feasible). So in the remainder of this paper we show how all of the features can be added at zero cost and no additional storage space.

Initial Firmware Validation

How do we verify that a BIOS is authentic? We can't just ask it while it is running, because it will lie if it is malicious. We already mentioned the two methods used by the Pogoplug and the WRT54: JTAG and trusted immutable boot ROM.

Another method similar to JTAG is to use the flash's SPI (Serial Peripheral Interface) to read the contents directly. Most embedded Linux devices use SPI flash for the firmware, and the D-Link and TP-Link devices both do. The SPI bus was designed to be sharable if it is properly buffered, and many PC motherboards feature buffered SPI interfaces for their BIOS for ease of modification (and the subsequent un-bricking).

Unfortunately the D-link and TP-Link do not properly buffer the SPI bus between the SoC and the flash, so any attempt to attach a hardware reader to the bus results in contention, and neither the SoC nor the reader function correctly. So at first this appeared to be an unworkable solution.

While there are many other theoretical ways to read the flash contents, such as real-time passive monitoring and reconstruction of the SPI data, or even power or RF monitoring, these methods are quite complex and expensive to implement. The only other known way to validate the flash's contents is to unsolder the flash chip from the mother board, so it can be read by the SPI reader without interference. Having actually done this, we concluded this was not an acceptable method for routine validation.

Going back to the SPI bus reading method, we looked for the simplest, low cost way that an owner (or even better, the vendor) could properly buffer the SPI bus. Both the D-Link and TP-Link devices are based on the same Atheros SoC, so any solution would work on both devices. Our preferred method for reading/programming SPI flash devices is the Buspirate [9] shown in figure 2, combined with the open source flashrom software application [10].

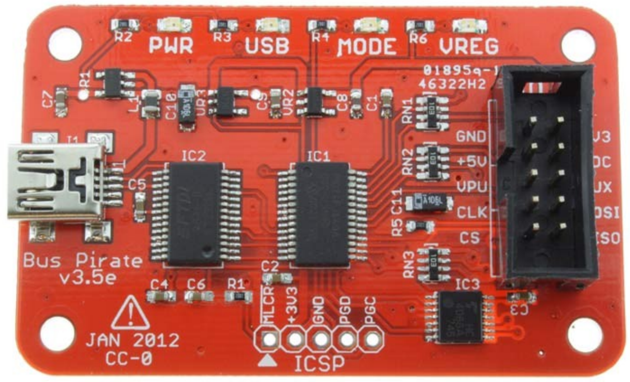

Figure 2: Buspirate SPI programmer

The buspirate is a $30 device with a USB interface for power and control from a PC running flashrom. It in turn has a 10 pin header with power supply outputs for optionally powering the chip, and input/output pins for reading/writing. While there are cheaper parallel port cables for SPI programming, bit-banging the SPI lines through a parallel port is much slower than using the buspirate, which has circuits for generating the needed serial clock and data signals directly.

Using the buspirate and flashrom, we experimentally determined that the SPI bus could be sufficiently buffered for in circuit programming with just three additional resistors as shown schematically in Figure 3. Figure 4 shows the buspirate connected to the MR-3020 while reading the flash contents, and figure 5 shows a more detailed view of the (crudely) added buffering resistors. The D-Link and TP-Link circuit boards already have a large number of resistors; adding three more would cost less than 1 cent in quantity, so we think this qualifies as a zero cost modification (and is much more convenient than chip unsoldering).

```
Atheros          SPI Flash           Bus
                                     Pirate

-----1Kohm-----CS     (pin 1)-----CS
-----1Kohm-----CLK    (pin 6)-----CLK
----200ohm-----SI     (pin 5)-----MOSI
---------------SO     (pin 2)-----MISO
---------------V+     (pin 8)-----3.3v
---------------GND    (pin 4)-----GND
---------------!WP    (pin 3)
---------------!hold  (pin 7)
```

Figure 3: SPI buffering Schematic

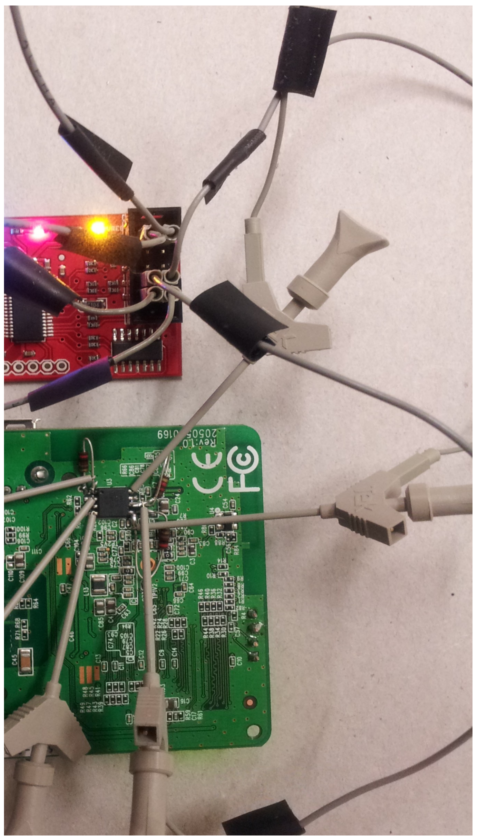

Figure 4: SPI in-circuit-programming

Run-time Firmware Protection and Firmware Update Protection

The second main integrity goal is to protect the integrity of the firmware from remote software attack. While this could easily be done by permanently wiring the flash's !WP pin low, this would prevent valid firmware updates and re-configurations. So the closely related third goal is to ensure that updates to the firmware are allowed, but validated or locally authorized in some way that is not by-passable by remote software attack.

The flash chips in all four of the sample embedded devices have a Hardware Protection Mode (HPM) which can block all writes to all or selected parts of the flash based on a combination of forcing the !WP pin low, and then setting a control register bit appropriately. The HPM control bit is non-volatile, and survives power cycles, so the only way to exit HPM mode is to force !WP high, and then reset the HPM control bit.

HPM leads to a very simple method for providing both flash locking, and secure local update. If the !WP pin on the flash is normally held low, with a physical momentary push button that can force the pin high, then the firmware bootstrap (u-boot [12]) can simply set the HPM mode bit, disabling all writes to the chip, locking the chip against any updates, including remote software attacks. Then, if an update or reconfiguration is desired, unlocking HPM mode can be done only if someone presses the button, establishing physical presence, for a secure local update.

The MR-3020 uses a Spansion S25FL032A [11] flash chip. This chip has HPM support, with the addition of 4 control register bits, which can select a subset of the chip's address space to protect. Table 3 shows the MR-3020's memory layout. For this prototype, the entire chip was locked, requiring physical presence for any update or configuration of the device, but by using the control bits, a subset could be protected to trade-off security and convenience.

The MR-3020 also has a convenient “WPS” momentary push button, normally used to begin Wireless Protected Setup for establishing a secure connection with a new client. This button has a default high output, which is forced low while the button is pushed, which is perfect for controlling the !WP pin. The button can be used for both functions, as the software context can understand which function is being requested, although the normal WPS function should be disabled anyway, due to its security weaknesses.

Figures 5 and 6 show the MR-3020 with both the one wire !WP modification, along with the three resistors for SPI buffering. These are the only hardware modifications needed for this device.

Part.	Name	Size	Contents
mtd0	“boot”	64KB	u-boot
mtd1	“kernel”	1024KB	Linux Kernel
mtd2	“rootfs”	2816KB	Linux root filesystem
mtd3	“config”	64KB	config data
mtd4	“ART”	64KB	radio config data

Table 3: MR-3020 Flash Layout

U-boot was modified to demonstrate/test the HPM with physical presence control. At boot time, u-boot attempts to lock the chip, in case it was unlocked before, and then attempts to unlock it. If the WPS button is pressed, then the unlock will succeed, and updates can be applied. If the WPS button is not pressed, then the unlock will fail, and updates will not be possible. In either case, before booting the Linux kernel, u-boot will re-lock the flash.

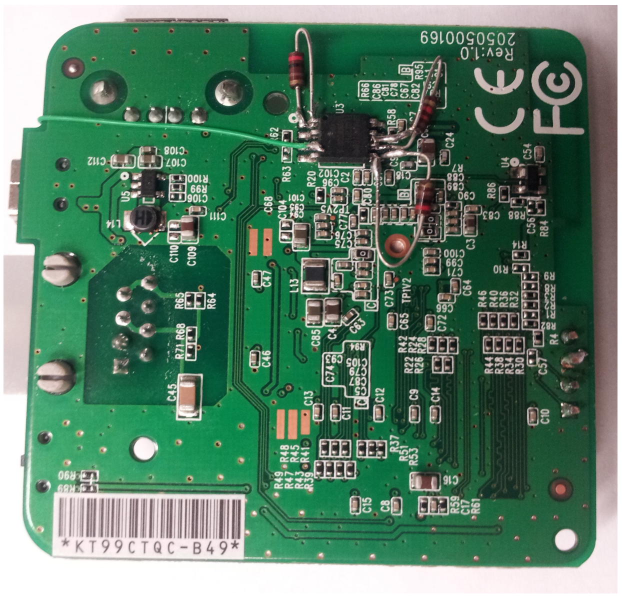

Figure 5: modified MR-3020 Bottom View

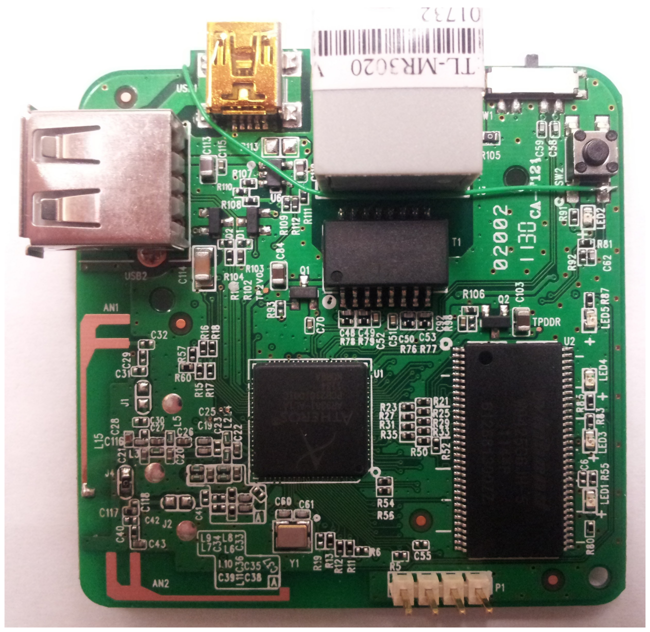

Figure 6: modified MR-3020 Top View

The following u-boot console log shows debugging output with the button pressed and not pressed. A status register value of 2 is unlocked, and 9c or 9e is locked:

```
Write_protect: starting SR = 2
Write_protect: ending SR = 9c
Write_unprotect: starting SR = 9e
Write_unprotect: ending SR = 9e
Write_unprotect failed.
…
Write_protect: starting SR = 2
Write_protect: ending SR = 9c
Write_unprotect: starting SR = 9e
Write_unprotect: ending SR = 2
Write_unprotect succeeded.
```

Integrity Protection on DIR-505

The DIR-505 has a similar WPS momentary push button for !WP control. It also has an interesting sliding switch for controlling the operating mode of the device, to choose between Router, Repeater, or Hotspot modes. This sliding switch actually is a four position switch, with the fourth position unused. The fourth position can be accessed simply by trimming the plastic slide a bit, so that the fourth position can be reached, and used to force the !WP pin high to allow updates/reconfigurations. Figure 7 shows the modified DIR-505 slide switch.

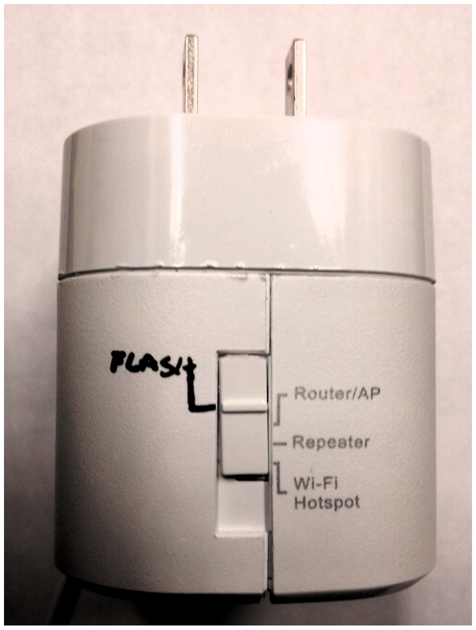

Figure 7: modified DIR-505

Boot-time Integrity Verification

The fourth integrity goal is to validate the firmware at boot time ("secure boot"). Assuming that the u-boot partition is locked with HPM as discussed, then it can be a trusted root to validate the linux kernel before loading and booting it. If the kernel is signed with a private key, and the corresponding public key is stored in the protected u-boot partition, then u-boot can do the validation, and the key can be changed only with physical presence.

This secure boot, with physical presence controlled key management was implemented on the MR-3020. The MR-3020 was chosen as it provided the greatest challenge. With the smallest flash chip (4MB), its entire u-boot partition is only 64K bytes, and the existing u-boot code used 54K, leaving just 10K bytes to implement all of the needed functions (HPM flash locking with physical presence control, RSA signature validation, and public key storage and management.

The RSA signature verification code was derived from the PolarSSL library [13] by stripping out everything not needed. The kernel signature was created with standard openssl commands, and the resultant binary signature simply appended to the end of the kernel. The (single) validating public key was stored in binary form at the end of the u-boot partition. The combined flash locking and signature verification code added roughly 8K bytes to u-boot, increasing its total size to 62K, which with the public key still fits within the 64K partition.

The following u-boot console debugging output shows hex formatted output of the sha1 hash of the kernel, the public key modulus, the binary PKCS1.5 signature, and the results of the verification.

```
## Booting image at 9f020000 ...
kernel sha1
E9321D87C091F971C8D955C399EBA53807429A61
modulus:
9292758453063D803DD603D5E777D7888ED1D5BF
35786190FA2F23EBC0848AEA
DDA92CA6C3D80B32C4D109BE0F36D6AE7130B9CE
D7ACDF54CFC7555AC14EEBAB
93A89813FBF3C4F8066D2D800F7C38A81AE31942
917403FF4946B0A83D3D3E05
EE57C6F5F5606FB5D4BC6CD34EE0801A5E94BB77
B07507233A0BC7BAC8F90F79
signature:
2CB0F653FF3BBCFF2E31ACC0840F02A84975B716
7291BB36EEE3F74D02EB3B6A
ACADE02CBCF6E2326230C296E4D8A8D70F309479
B388A99591AD5C41938280E3
F51EA9865ED8A0360A0F5BD6A6C676C363B43E54
61D9CCF00D46E1B5449CB262
BDE36CAD4AFBEE51ED731BBF48340F290DF8DD84
4791D81259CEDF99CD1CA2E6
rsa verify kernel succeeded
   Uncompressing Kernel Image ... OK
```

Summary

Table 4 shows the final results of these modifications on the D-Link and TP-Link devices, and similar modifications on the Pogoplug and Linksys devices. With essentially zero cost hardware and software modifications we can meet all four integrity goals on all four example devices. With firmware measurement, we can detect supply chain or other firmware modification. With HPM locking, we can protect the firmware from remote modification, even if the remote attacker gets the root password as in all of the earlier described web management vulnerabilities. As physical presence is needed to unlock the flash, we provide secure local update. If the kernel partition is not locked for convenience, secure boot can provide strong validation, with secure local update of the validating public key.

Device	Measure BIOS?	Lock BIOS?	Signed-local updates?	Secure Boot?
Pogoplug	Yes - SATA	Yes	Yes	Yes
D-Link DIR-505	Yes Buspirate	Yes	Yes	Yes
TP-Link MR3020	Yes Buspirate	Yes	Yes	Yes
Linksys WRT54G	Yes - JTAG	Yes	Yes	Yes

Table 4: Integrity features after modification.

References

[1] Fabio Assolini, “The tale of one thousand and one DSL modems”, Securelist, 2012,
`https://www.securelist.com/en/blog/208193852/The_tale_of_one_thousand_and_one_DSL_modems`

[2] roberto@greyhats.it, Bugtraq Mailing List
27 February 2013
`http://archives.neohapsis.com/archives/bugtraq/2013-02/0151.html`

[3] Phil Purviance, “Don't Use Linksys Routers”, March 2013
`https://superevr.com/blog/2013/dont-use-linksys-routers/`

[4] ISE, “Exploiting SOHO Routers”, April 2013
`http://securityevaluators.com//content/case-studies/routers/soho_router_hacks.jsp`

[5] NIST, BIOS Integrity Measurement Guidelines (Draft) SP 800-155, December 2011
`http://csrc.nist.gov/publications/drafts/800-155/draft-SP800-155_Dec2011.pdf`

[6] NIST, BIOS Protection Guidelines
NIST-SP800-147, April 2011
`http://csrc.nist.gov/publications/nistpubs/800-147/NIST-SP800-147-April2011.pdf`

[7] Trusted Computing Group, “Trusted Boot”
`http://www.trustedcomputinggroup.org/resources/trusted_boot/`

[8] UEFI, Secure boot
UEFI specification 2.3.1, section 1.8.1
`http://www.uefi.org/specs/download/2_3_1_D.zip`

[9] Buspirate
`http://dangerousprototypes.com/docs/Bus_Pirate`

[10] Flashrom
`http://flashrom.org`

[11] Spansion, S25FL032A reference manual
`http://www.spansion.com/Support/Datasheets/S25FL032A_00.pdf`

[12] u-boot
`http://www.denx.de/wiki/U-Boot/WebHome`

[13] Polar SSL library
`https://polarssl.org/`