Defending Against Rogue Hard Disks

Anitha Gollamudi, Andres Perez
Ivan Tadeu Ferreira Antunes Filho, Ben Yuan

May 14, 2015

Abstract

Recent revelations as to the ability of APTs to embed code in hard disk firmware, as well as the existence of prior public literature describing similar work, raise concerns as to the future potential proliferation and weaponization of this technique. We describe previous work in this domain, including both recent revelations and prior research, and then discuss our strategy for mitigating the impact of similar techniques with the technology available today for us to use. Specifically, we describe a procedure that sets up a Secure Boot hierarchy at install time, which may be used to verify the integrity of an installed bootloader, and sets up full disk encryption to protect kernel and user code and data. The combination of these two approaches greatly increases the difficulty of a useful hard-disk-mediated attack; we describe the considerable remaining room for improvement.

1 Introduction

In the past decade, computer security attacks have gotten very sophisticated. It is becoming more and more important to not only have forms of security for all kinds of devices, but to also have safety measures. Being able to determine when trusted firmware has been modified provides us with an adequate safety.

Traditional hard drives have a dominant role in the market for persistent data storage. They can store many classes of data, including program code, operating system code, and application and user information. Modern hard drives implement a high degree of intelligence, from caching to data encoding to hard- ware encryption, and much of this intelligent capability is implemented on programmable micro-controllers, updatable using proprietary commands issued by vendor-supplied tools.

The two general assumptions most people tend to have when dealing with hard disks is that only trusted sources can modify the firmware, and that the firmware is doing what the manufacturer has agree with. In other words, the firmware reads data that is actually on the hard disk, without hiding any, and always writes data that the operating system has told it to write. But how can we tell when the latter assumption no longer holds, and how can we tell that there is persistent malware living in our hard disk firmware? In this paper we'll discuss previous attacks on hard disk firmware, and on how to detect when systems have been compromised.

2 Background

2.1 Previous work

One day in 2009, researchers were victims to one of the most sophisticated attacks to date. A package containing a CD was intercepted by group now know as Equation Group, and it's contents were injected with malware.[1] Kaspersky lab decided to investigate this group's attacks further and discovered sophisticated malware that was able to change the firmware on many different types of hard disks, like Samsung, Western Digital, Seagate and others.[1]

The allegations made by Kaspersky Lab regarding the development of advanced hard disk programming capability [2] do not represent the first appearance of this technique in the literature; prior work was undertaken by Jeroen Domburg (Sprite) on hard disks manufactured by Western Digital [3]. This work involved extraction and reverse engineering of firmware from a hard disk using hardware debugging and disassembly techniques. After six months of work, Domburg was able to reverse-engineer the Western Digital firmware and update mechanism and add working exploit code that modifies a Linux system password database on demand; a significant amount of code from this effort is available for examination. Concurrently and independently, Zaddach et al., [4] presented a practical data exfiltration backdoor for a SATA hard disk. The backdoor is self-contained and is installable. Despite no implementation, the paper discusses applicable detection and defensive techniques (e.g., encryption of data at rest, signed firmware updates) as well as their ramifications. Both Zaddach et al and Domburg dumped the firmware and relied heavily on debugging tools for reverse-engineering.

2.2 Full-disk encryption

Full-disk encryption, or FDE, refers to any scheme that aims to provide encryption of an entire hard disk with minimal impact on performance. It is commonly used to protect data against retrieval by offline attackers; to access a FDE-protected volume successfully, a user must have access to a corresponding encryption key. With specialized operating system tools, full disk encryption can even be applied to system volumes; a scheme with such support will install a specialized bootloader and pre-boot environment capable of decrypting the system at boot time.

Full-disk encryption solutions are generally symmetric-key schemes, protecting a symmetric encryption key using one or more user passwords. This is done in such a way that the unprotected symmetric key is never written to the disk; it can be derived from a password directly using a password-based key derivation function, or it can itself be encrypted using such a password-derived key and written in that form to a volume header.

Any encryption solution aims to provide confidentiality: it should be "hard" for an adversary to gain any information about the original plaintext without access to the encryption key. Additionally, a good encryption solution will aim to provide non-malleability: it should be "hard" for an adversary to make changes to the ciphertext that map cleanly to changes in the plaintext. Understanding the degree to which current solutions provide these properties in practice, and thus the degree of protection they afford from a malicious hard disk, requires understanding the algorithms used and their limitations.

2.2.1 Algorithms used

The block device environment is challenging for encryption due to the necessary support for random access and the desired attribute of parallelizability, both for encryption and for decryption. Additionally, the nature of block devices as error-prone physical media raises an additional challenge, namely in the limitation of error propagation to prevent single-bit errors from invalidating large sections of a disk. Accordingly, block ciphers and cipher modes have to be designed and selected with these constraints in mind.

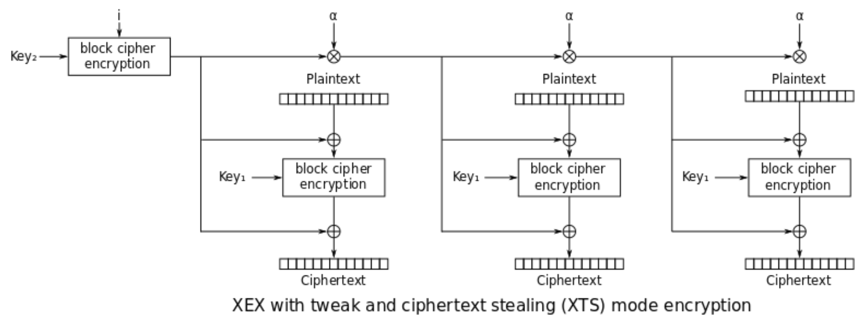

i
α
Key₂
block cipher encryption
Plaintext
Key₁
Ciphertext
XEX with tweak and ciphertext stealing (XTS) mode encryption

Figure 1: Diagram of the XTS encryption mode.

The current "gold standard" supported by modern FDE solutions is AES-XTS, the well-regarded AES block cipher using the XTS cipher mode. XTS, or "XEX-based tweaked-codebook mode with ciphertext stealing", may be described as follows [5]:

- Two symmetric keys Key_1 and Key_2 are generated.
- Every disk sector (typically 512 bytes), partitionable into 16-byte blocks, receives an initialization vector i, assigned consecutively starting from an arbitrarily chosen non-negative integer. This IV is encrypted using Key_2 and multiplied j times by α, a primitive element of $GF\left(2^{128}\right)$, to compute a value T_j for block j of the sector.
- For encryption, the plaintext P_j for block j is first XORed with T; the output is then encrypted using Key_1, and then XORed again with T to produce the ciphertext C_j.
- Decryption is the inverse of the above procedure.
- The technique of ciphertext stealing may be used to handle unevenly sized sectors. Consumer hard disks generally have 512-byte or 4096-byte sectors, so this is generally not necessary.

While XTS provides effective parallelizable encryption and decryption, it does suffer from several limitations. Because any given sector receives the same initialization vector, and because blocks are independent of each other, any 16-byte plaintext present in the exact same location of the disk, presuming the same keys are used, will necessarily encrypt to the same 16-byte ciphertext. While not as problematic as ECB, which is location-invariant in this property, an attacker capable of taking multiple disk snapshots over time can still observe data patterns over time. Additionally, XTS does afford a limited degree of malleability, contrary to our earlier desideratum; while it is not known to be possible to do better than randomization of arbitrarily chosen 16-byte blocks, 16 bytes is unfortunately small in relation to the x86 instruction set. Arguably, it is possible to introduce a security hole in existing code without introducing an outright crash [6], presuming that an attacker knows where system code resides on the disk. While such attacks could be mitigated by randomization of on-disk program location at install time, it is conceivable that disk access patterns could reveal the location of code segments over time.

In terms of the actual protection afforded by current full-disk encryption solutions, then, AES-XTS offers a significant amount of room for improvement. However, in terms of what is readily deployable as of this paper, AES-XTS enjoys a broad spectrum of support matched only by the significantly weaker AES-CBC-ESSIV, which exhibits bit-level malleability. We hence use AES-XTS as our algorithm of choice, with the caution that separate methods are necessary to verify code and data integrity.

The actual implementation we use for FDE is provided by the `dm-crypt` Linux kernel module, using the LUKS (Linux Unified Key Setup) system to manage encryption keys [7]. LUKS generates a master symmetric key used to encrypt the rest of the volume; this key is encrypted using a user password and then stored in a key slot on disk. This approach enables changing the disk password and allowing support for multiple passwords without needing to re-encrypt the entire disk. `dm-crypt` enjoys native kernel support from Linux as well as well-developed support in the Ubuntu boot sequence. Additionally, the GRUB boot loader also supports `dm-crypt` natively, decreasing the amount of necessary plaintext exposure.

2.3 UEFI Secure Boot

Though full-disk encryption offers protection for the kernel, initramfs, and user applications, it still leaves the bootloader unencrypted. Secure boot solves this problem by offering us a way to detect unsigned bootloaders.

Unified extensible hardware interface (UEFI[8]) is an abstract specification that defines interface between operating system (OS) and platform hardware. It is the successor of Extensible Firmware Interface (EFI). The interface is in the form of data tables that contain platform-related information, boot and runtime service calls that are available to the OS loader. From [9], UEFI secure boot *is* *a protocol that can secure the boot process by preventing the loading of drivers or OS loaders that are not signed with an acceptable digital signature.* In short, it verifies boot path sequence.

Figure 2 presents the overview of secure boot. The device(hardware) comes with a platform key secured in its non-volatile memory. Once powered on, after some setup, UEFI boot manager verifies the signature of bootloader using PK and loads it if successful. The verified bootloader in turn verifies OS using additional keys referred to as Key Exchange Keys stored in signature databases. UEFI binaries are signed with KEK_{priv} and verified with KEK_{pub} stored in databases. We describe this process in detail in section 2.3.1. Once OS is verified, bootloader loads the OS which can inturn verify device drivers. A signature mismatch at any stage during the booting leads to a failure.

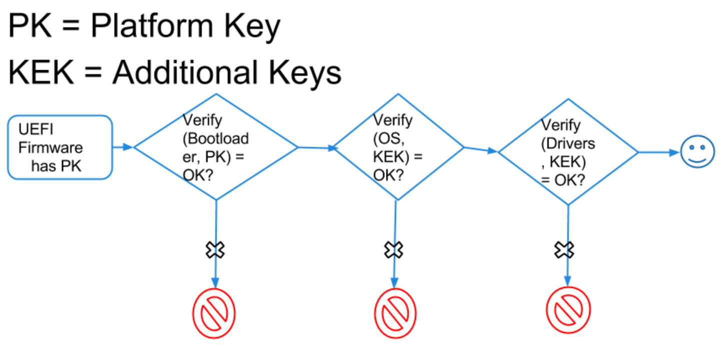


Figure 2: Secure Boot Overview

2.3.1 Enrolling Keys

Secure boot uses 2 important keys - Platform Key (PK) and Key Exchange Key (KEK).

- Platform Key: Establishes trust between platform owner and firmware. Owner enrolls PK_{pub} into the firmware and uses PK_{priv} to authenticate any changes later on. PK must be stored in non-volatile storage which is tamper and delete resistant
- Key Exchange Key: Establishes trust between operating system (OS) and the

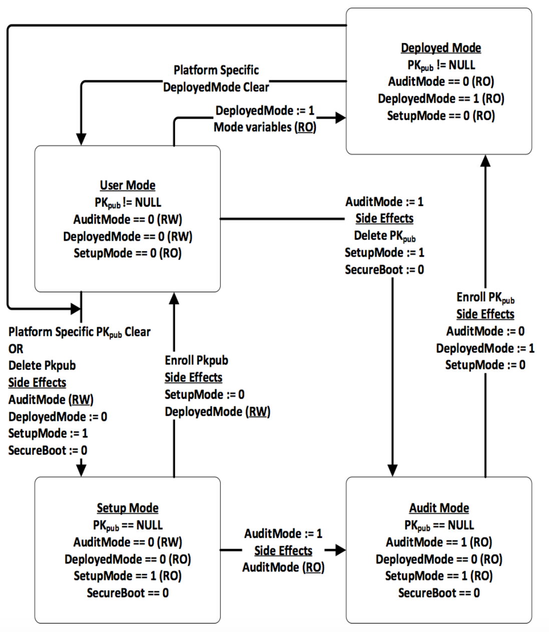


Figure 3: Secure Boot Modes

firmware. Each OS enrolls a KEK_{pub}. KEKs must be stored in non-volatile storage which is tamper resistant.

A secure boot operates in 2 primary modes - setup and user. UEFI also specifies Audit and Deployed modes which we skip for brevity. Figure 3 illustrates the mode transitions.

- When no PK is enrolled, setup mode is turned on. In this mode, no authentication is required to updated PK. Once PK is enrolled, setup mode gets turned off.
- Until a platform key is cleared, user mode is turned on.

All secure boot keys are stored in UEFI authenticated variables. Keys can be updated in user mode only if authentication step succeeds.

2.3.2 Enrolling Platform Key (PK)

Platform Keys are enrolled by platform owner using UEFI Boot Service setVariable(). In setup mode, PK_{pub} is enrolled by signing it with its counterpart PK_{priv}. In user mode, PK_{pub} must be signed by current PK_{priv}. The platform vendor may provide a default $PK_{default}$ which can be used to transition from setup mode to user mode.

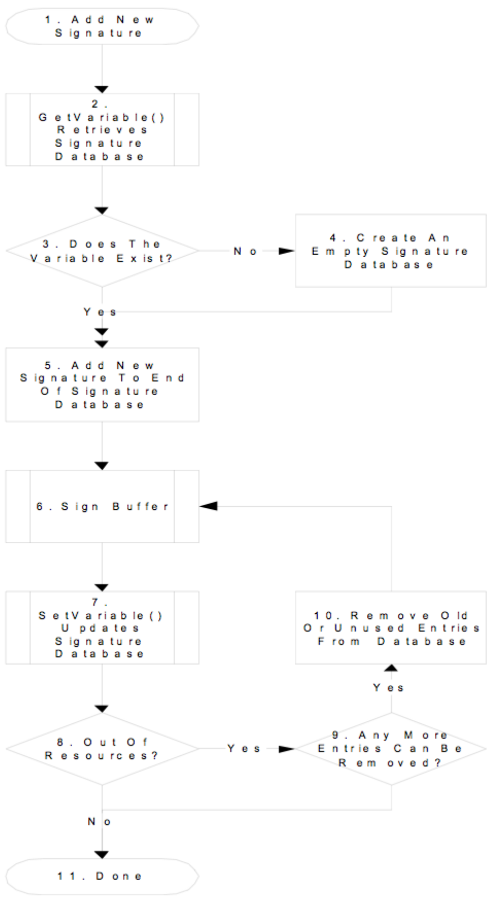

1. Add New Signature
2. GetVariable() Retrieves Signature Database
3. Does The Variable Exist?
No
4. Create An Empty Signature Database
Yes
5. Add New Signature To End Of Signature Database
6. Sign Buffer
7. SetVariable() Updates Signature Database
10. Remove Old Or Unused Entries From Database
Yes
8. Out Of Resources?
Yes
9. Any More Entries Can Be Removed?
No
11. Done

Figure 4: Adding a new signature by the OS

2.3.3 Enrolling Key Exchange Keys (KEKs)

KEKs can be written in setup mode
or in user mode. In setup mode it should comply to format of UEFI authenticated variable while in user mode it needs to be signed by the current PK_{priv}.

2.3.4 Signature Database Update

Signature database can be updated in setup or user mode. In user mode, the provided variable is signed with previously enrolled KEK_{priv} or PK_{priv}. OS can add new keys to the database. The process is illustrated in figure 4 [8, figure 79 page 1800]

2.3.5 Pros and Cons

Secure boot can be used to detect tampered OS and bootloaders, altered boot sequences. It detects rootkit attacks. When complemented with FDE, it can greatly limit the impact of malicious hard disk firmware. There are few limitations as well.

1. When UEFI secure boot is enabled, attempts to boot non-UEFI OS fail
2. Currently, some kernel level features like kprobe, kdump, kexec had to be disabled to comply with the nature of secure boot
3. Can be politicised to discriminate and deter new market entrants.

2.3.6 Open Source Secure Boot Tools

[10] has a set of utilities for signing secure boot images and updating keys. In particular ”sbkeysync” can be used to update signature databases from within an OS. In our project, we used these tools to enroll PK and KEK enabling secure boot

3 Our method

We have a partially automated procedure tested in QEMU with OVMF, ”a project to enable UEFI support for Virtual Machines”, that sets up full disk encryption and secure boot, offering protection against offline attacks similar to those achievable by hard disk firmware.

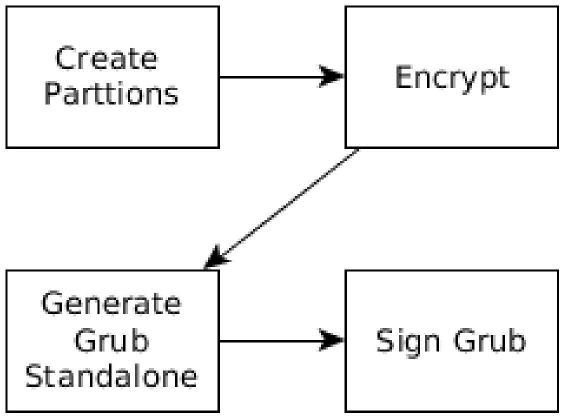
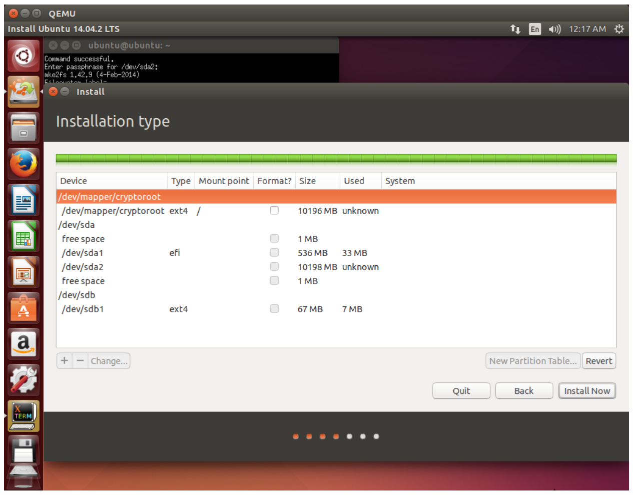


Figure 5: Procedure Overview

- Create a primary partition and an EFI partition using parted.
- Format the EFI partition, the partiton where the unencrypted Grub standalone will reside.
- Set up LUKS, Linux Unified Key Setup, to encrypt our primary partition in AES-XTS mode.
- Set cryptodisk enabled in /etc/default/grub
- Install Ubuntu from the live CD. The install will fail, when it tries to install Grub, because of the full disk encryption. This will copy all the other files necessary for a running system, however.
- Write an entry for the main cryptodisk to /etc/crypttab, the list of encrypted devices that are set up during boot.
- Make sure that GRUB_ENABLE_CRYPTODISK=y in /etc/default/grub and produce a standalone Grub boot loader using `grub-mkstandalone`, ensuring all necessary modules will be linked into the main `core.img` file.
- Generate a key and a self-signed certificate using openssl. [11]
- Use sbsiglist to create an EFI_SIGNATURE_LIST signature database containing the certificate.

- Use sbvarsign to generate signed updates, i.e. authenticated variables, for the EFI signature databases.
- Copy the keys and certificate to the standard locations in "/etc/secureboot/keys/" , "/etc/secureboot/keys/PK/", "/etc/secureboot/keys/KEK/" and "/etc/secureboot/keys/db/", for them to be found by sbkeysync.
- Use sbkeysync to add our keys to the firmware database.
- Use sbsign to sign the standalone Grub generated previously, as well as the Shim first-stage bootloader and the kernel to boot. (In each case, ensure signing input is from the CD or the live environment RAN disk; don't try to directly sign what's already on the disk.)
- Copy the signed Grub and Shim to the EFI partition.
- Add Shim to the EFI boot entry using efibootmgr.



```
Command successful.
Enter passphrase for /dev/sda2:
mke2fs 1.42.9 (4-Feb-2014)
```

Device	Type	Mount point	Format?	Size	Used	System
/dev/mapper/cryptoroot						
/dev/mapper/cryptoroot	ext4	/	☐	10196 MB	unknown	
/dev/sda						
free space			☐	1 MB		
/dev/sda1	efi		☐	536 MB	33 MB	
/dev/sda2			☐	10198 MB	unknown	
free space			☐	1 MB		
/dev/sdb						
/dev/sdb1	ext4		☐	67 MB	7 MB	

Figure 6: Image of the Ubuntu setup process in the QEMU VM.

The only plain text that the hard drive ever sees is the Grub standalone image and Shim first-stage bootloader; however, since they are signed, if they are modified it won't run anymore. This setup gives us a trusted boot path that we control completely, making future tampering by any party much more difficult.

4 Limitations

Our approach provides no real integrity assurances beyond the bootloader and kernel. It is very hard for the disk drive to run code generated by itself, or for it to specifically choose programs to run, since they are encrypted; however, it can still cause execution of randomly generated user-space code. To protect against this, we would need filesystem-level signing and a security policy denying unsigned file access, or something that provides similar functionality.

Key management becomes an important issue. Control over the file disk encryption key and the secure boot keys implies control over the whole running system, and so careful risk assessment becomes important in determining how those keys should be controlled. In an ideal world, one would use a hardware security module, but these are expensive. One could save the keys in an external USB storage device, but those have known issues, such as the BadUSB firmware exploit.

The current installation procedure still has a manual component, which makes execution difficult. To make the installation procedure more feasible, some extra development work would be needed; the current procedure is strictly a proof-of-concept and makes many assumptions.

In order to provide kernel verification capability to GRUB, we install the Shim first-stage bootloader, which exposes an API for this purpose to GRUB. The version of Shim that ships with the current Ubuntu installer has the Canonical signing public key embedded, and the disk does not appear to ship with sufficient prerequisites to compile a new Shim. Adding the necessary ingredients to the Ubuntu live CD would patch up this potential security concern.

The version of GRUB shipped with Ubuntu appears to happily boot unsigned kernels under certain conditions - for instance, if a signed kernel has failed to boot and an unsigned kernel happens to reside in the `/boot` directory. This is an obvious security hole, allowing (at minimum) a disk drive to present an unsigned kernel that has ever been legitimately written to the `/boot` directory. Further work would require patching GRUB so as not to allow this behavior. Other distributions may ship versions of GRUB that do not have this behavior; we suspect the Ubuntu decision was made on the grounds of general usability due to its particular target audience, to avoid breaking the boot process in case Secure Boot was applied incorrectly.

References

[1] Arstechnica, *How 'omnipotent' hackers tied to the NSA hid for 14 years- and were found at last*, [Online; accessed 11-May-2015]. [Online]. Available: `http://arstechnica.com/security/2015/02/how-omnipotent-hackers-tied-to-the-nsa-hid-for-14-years-and-were-found-at-last/`.

[2] Kaspersky Lab, "Equation group: Questions and answers," Kaspersky Lab, Tech. Rep., 2015.

[3] J. Domburg, *Hard disk hacking.* [Online]. Available: `http://spritesmods.com/?art=hddhack`.

[4] J. Zaddach, A. Kurmus, D. Balzarotti, E. O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Koltsidas, "Implementation and implications of a stealth hard-drive backdoor," in *ACSAC 2013, 29th Annual Computer Security Applications Conference, December 9-13, 2013, New Orleans, Louisiana, USA*, New Orleans, UNITED STATES, Dec. 2013. DOI: `http://dx.doi.org/10.1145/2523649.2523661`. [Online]. Available: `http://www.eurecom.fr/publication/4131`.

[5] "IEEE standard for cryptographic protection of data on block-oriented storage devices," *IEEE Std 1619-2007*, pp. c1–32, Apr. 2008. DOI: `10.1109/IEEESTD.2008.4493450`.

[6] *Public comments on the XTS-AES mode*, Sep. 2008. [Online]. Available: `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/XTS/collected_XTS_comments.pdf`.

[7] C. Fruhwirth, *LUKS on-disk format specification*, Oct. 2011. [Online]. Available: `https://gitlab.com/cryptsetup/cryptsetup/wikis/LUKS-standard/on-disk-format.pdf`.

[8] *Unified extensible firmware interface specification*, version 2.5, Apr. 2015. [Online]. Available: `http://www.uefi.org/sites/default/files/resources/UEFI%202_5.pdf`.

[9] Wikipedia, *Unified extensible firmware interface — Wikipedia, the free encyclopedia*, [Online; accessed 10-May-2015], 2004. [Online]. Available: `http://en.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface#Secure_boot`.

[10] *Secure boot signing tools.* [Online]. Available: `https://launchpad.net/sbtools/`.

[11] J. Kerr, *Sbkeysync & maintaining uefi key databases*, Aug. 2012. [Online]. Available: `http://jk.ozlabs.org/docs/sbkeysync-maintaing-uefi-key-databases/`.

A Code used and testing protocol

All code provided here assumes:

- The target machine begins in 'setup mode' with respect to Secure Boot.
- The target physical disk is located at `/dev/sda` and the auxiliary key storage disk is located at `/dev/sdb`. Both disks are overwritten in their entirety during the install procedure.
- No mistakes are made during the manual prompts.
- `ubiquity` is launched with the `-b` flag, which prevents bootloader installation.
- The scripts themselves are loaded in `/usr/local/etc on the live CD filesystem, and are run as root`.

An example qemu invocation might be:

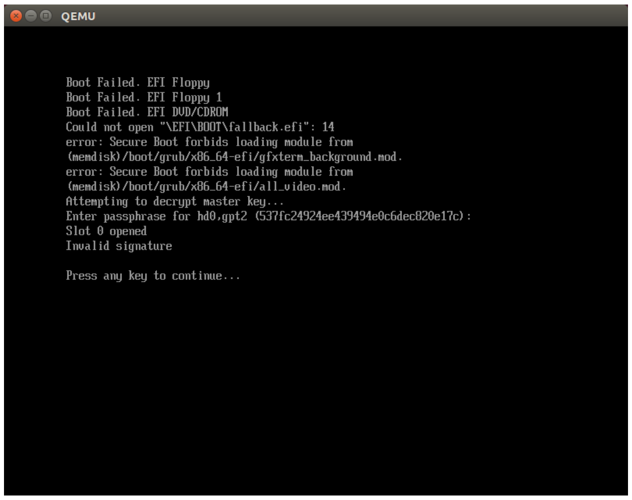
```
sudo qemu-system-x86_64 -enable-kvm -m 2048 -hda test.img -hdb extern_storage.img
-pflash OVMF.fd -cdrom /path/to/ubuntu-custom.iso -show-cursor
```

assuming that the relevant disk and firmware images are in the current directory.

The overall installation procedure from a user perspective:

1. Load the appropriately modified Ubuntu live environment. Open a terminal.
2. Run `sudo /usr/local/etc/preinstall.sh`. Follow all prompts.
3. Run the Ubiquity installer with the `-b` flag. For partitioning, choose "something else". Set the mount point for the `/dev/mapper/cryptoroot` virtual device as `/`. Follow all prompts.
4. When Ubiquity finishes, choose "continue testing".
5. Run `sudo /usr/local/etc/postinstall.sh`. Follow all prompts.
6. Restart the computer. Marvel at the secureness of the boot path.

We verified that this procedure worked. We also verified that introducing bit errors into the plaintext of an otherwise signed kernel would cause GRUB to refuse to boot the kernel (due to 'invalid signature').



```
Boot Failed. EFI Floppy
Boot Failed. EFI Floppy 1
Boot Failed. EFI DVD/CDROM
Could not open "\EFI\BOOT\fallback.efi": 14
error: Secure Boot forbids loading module from
(memdisk)/boot/grub/x86_64-efi/gfxterm_background.mod.
error: Secure Boot forbids loading module from
(memdisk)/boot/grub/x86_64-efi/all_video.mod.
Attempting to decrypt master key...
Enter passphrase for hd0,gpt2 (537fc24924ee439494e0c6dec820e17c):
Slot 0 opened
Invalid signature

Press any key to continue...
```

Figure 7: What happens when your kernel is bit-flipped.

As of May 2015, the modified install disk we created may be found at `http://link.csail.mit.edu/scratch/ubuntu-custom.iso`.

A.1 /usr/local/etc/preinstall.sh

```
#!/bin/bash

parted /dev/sda mklabel gpt
parted /dev/sda mkpart ESP fat32 1MiB 513MiB
parted /dev/sda set 1 boot on
parted /dev/sda mkpart primary 513MiB 100%


# format the EFI system partition
mkfs.vfat -F32 /dev/sda1

# wipe the primary container
cryptsetup open --type plain /dev/sda2 container
dd if=/dev/zero of=/dev/mapper/container
cryptsetup close container
```

```

# setup LUKS
# on current arch this sets up aes-xts-plain64 just fine
cryptsetup -y -v luksFormat /dev/sda2
cryptsetup open /dev/sda2 cryptoroot
mkfs -t ext4 /dev/mapper/cryptoroot
mount -t ext4 /dev/mapper/cryptoroot /mnt
```

A.2 /usr/local/etc/postinstall.sh

```
#!/bin/bash
parted /dev/sdb mklabel gpt
parted -a optimal /dev/sdb mkpart primary 0% 100%
mkfs.ext4 /dev/sdb1

mount /dev/sdb1 /mnt

cd /tmp

openssl genrsa -out test-key.rsa 2048
openssl req -new -x509 -sha256 -subj '/CN=test-key' -key test-key.rsa -out test-
    cert.pem
openssl x509 -in test-cert.pem -inform PEM -out test-cert.der -outform DER

guid=$(uuidgen)

sbsiglist --owner $guid --type x509 --output test-cert.der.siglist test-cert.der

for n in PK KEK db
do
   sbvarsign --key test-key.rsa --cert test-cert.pem --output test-cert.der.
       siglist.$n.signed $n test-cert.der.siglist
done

sudo mkdir -p /mnt/secureboot/keys/{PK,KEK,db,dbx,private}
sudo cp *.PK.signed /mnt/secureboot/keys/PK/
sudo cp *.KEK.signed /mnt/secureboot/keys/KEK/
sudo cp *.db.signed /mnt/secureboot/keys/db/
sudo cp test* /mnt/secureboot/keys/private

sbkeysync --verbose --pk --keystore /mnt/secureboot/keys --no-default-keystores

mount /dev/mapper/cryptoroot /target
chroot /target bash /usr/local/etc/chroot-postinstall.sh

sbsign --key test-key.rsa --cert test-cert.pem --output /tmp/grubx64.efi /target/
    tmp/grubx64.efi
sbsign --key test-key.rsa --cert test-cert.pem --output /tmp/bootx64.efi /target/
    usr/lib/shim/shim.efi
sbsign --key test-key.rsa --cert test-cert.pem --output /target/boot/vmlinuz-`
    uname -r`.efi.signed /cdrom/casper/vmlinuz.efi

mkdir /efi
mount /dev/sda1 /efi
mkdir -p /efi/efi/boot
cp /tmp/grubx64.efi /efi/efi/boot/grubx64.efi
cp /tmp/bootx64.efi /efi/efi/boot/bootx64.efi
```

A.3 /usr/local/etc/chroot-postinstall.sh

```
#!/bin/bash
mount /sys
mount /dev
mount /proc
mount -t tmpfs tmpfs /tmp

mount /dev/sr0 /cdrom

echo "cryptoroot /dev/sda2 none luks" > /etc/crypttab

dpkg-reconfigure cryptsetup

dpkg -r grub-gfxpayload-lists grub-pc
dpkg -i /cdrom/pool/main/e/efibootmgr/efibootmgr* /cdrom/pool/main/g/grub2/grub-efi-amd64* /cdrom/pool/main/s/shim/shim*

echo "GRUB_TERMINAL=console" >> /etc/default/grub
echo "GRUB_CMDLINE_LINUX=\"cryptdevice:/dev/sda2:cryptoroot\"" >> /etc/default/grub

cd /tmp
grub-mkconfig -o grub.cfg

grub-mkstandalone -d /usr/lib/grub/x86_64-efi/ -O x86_64-efi -o grubx64.efi "boot/grub/grub.cfg=/tmp/grub.cfg" --modules="acpi␣boot␣cat␣configfile␣cryptodisk␣datetime␣echo␣elf␣ext2␣fshelp␣gcry_rijndael␣gcry_sha1␣gettext␣gzio␣halt␣help␣linuxefi␣linux␣loadenv␣ls␣lsefi␣luks␣lvm␣msdospart␣multiboot␣normal␣part_gpt␣probe␣read␣reboot␣search␣sleep␣terminal␣test"
```