STARK
Tamperproof Authentication to Resist Keylogging

Tilo Müller, Hans Spath, Richard Mäckl, and Felix C. Freiling
Department of Computer Science
Friedrich-Alexander-University, Erlangen, Germany

Abstract. The weakest link in software-based full disk encryption is the authentication procedure. Since the master boot record must be present unencrypted in order to launch the decryption of remaining system parts, it can easily be manipulated and infiltrated by bootkits that perform keystroke logging; consequently password-based authentication schemes become attackable. The current technological response, as enforced by BitLocker, verifies the integrity of the boot process by use of the trusted platform module. But, as we show, this countermeasure is insufficient in practice. We present STARK, the first tamperproof authentication scheme that *mutually authenticates* the computer and the user in order to resist keylogging during boot. To achieve this, STARK combines two ideas in a novel way: (1) STARK implements *trust bootstrapping* from a secure token (a USB flash drive) to the whole PC. (2) In STARK, users can securely verify the authenticity of the PC before entering their password by using *one-time boot prompts*, that are updated upon successful boot.

Keywords: Disk Encryption, Evil Maid Attacks, Authentication, TPM

1 Introduction

Full disk encryption (FDE) protects sensitive data against unauthorized access in the event that a device is physically lost or stolen. The risk of data exposure is reduced by rendering disks unreadable to unauthorized users through encryption technologies like AES [11]. Unlike file encryption, FDE encrypts whole disks automatically on operating system level, without the need to take action for single files. Therefore, FDE is both more secure and more user-friendly than file encryption, and consequently, it can be considered best practice for securing sensitive data against accidental loss and theft.

As stated in a SECUDE survey from 2012 [36], full disk encryption is the most popular strategy for data protection in the majority of U.S. enterprises today. Also, the U.S. government recommends agencies to encrypt data on mobile devices to compensate the lack of physical security outside their agency location [19]. In Ponemon's annual study about the *U.S. Cost of a Data Breach* from 2011 [32], it is stated that "breaches involving lost or stolen laptop computers or other mobile data-bearing devices remain a consistent and expensive threat". This threat must be counteracted by full disk encryption systems like BitLocker for Windows, FileVault for Mac, dm-crypt for Linux, or the cross-platform utility TrueCrypt.

1.1 Threat Model

FDE cannot protect sensitive data when a user logs into the system and leaves it unattended. Likewise, FDE does generally not protect against system subversion through malware. In the case of malware infiltration, adversaries can access data remotely over a network connection with user privileges after the user logged in. Hence, hard disk encryption is only intended for scenarios where an adversary gains *physical access* to a target.

Given physical access, two types of attacks can be distinguished: opportunistic and targeted attacks. In *opportunistic* attacks the adversary steals a computer and immediately tries to retrieve the data. This is the attack scenario which is withstood by all widespread FDE solutions, including BitLocker and TrueCrypt. If an attacker simply grabs the computer and runs, data decryption without having the key or password is a futile task with today's crypto primitives like AES. However, a more careful adversary can carry out a *targeted* attack on the key, password, or access control management. Besides *cold boot attacks* [13, 8] and *DMA attacks* over FireWire [5, 30], PCIe [7, 9] or Thunderbolt [6, 33], keylogging attacks via *bootkits* [24, 25] are a notable threat.

Software-based FDE needs to modify the master boot record (MBR) of a hard drive in order to present pre-boot environments for user authentication. Commonly, pre-boot screens ask users for credentials in form of a secret password or passphrase, but they can also ask for credentials such as smart cards and tokens. Only after a user is authenticated, the operating system is decrypted and prepared to take over system control.

The MBR of an encrypted hard drive can easily be manipulated because it is necessarily left unencrypted for bootstrapping since CPUs can interpret only unencrypted instructions. As a consequence, bootkits can always be placed in the MBR to subvert the original bootloader with software-based keylogging. Such attacks are also referred to as *evil maid attacks* [16] and typically require access to the target machine twice: Let the victim be a traveling salesman who left his laptop in a hotel room and goes out for dinner. An "evil maid" can gain physical access to her target system unsuspiciously now. She replaces the original MBR with a malicious bootloader that performs keylogging and, later on, the unaware salesman boots up his machine and enters his password. On the next event, the evil maid can access the laptop a second time and reads out the logged passphrase.[1]

As shown by a recent study [29] on the security of *hardware*-based FDE (so-called self-encrypting drives, SEDs), evil maid attacks are generally *not* defeated by SEDs although these drives encrypt the MBR [14]. The reason is that evil maids can alternatively replace the entire disk drive, plug in a tiny bootable USB drive, flash the BIOS image [35, 18] or UEFI image [26, 3], or even replace the machine with an identical model. Overall, targeted attacks with repeated physical access (such as evil maid attacks) constitute the threat model of our paper.

[1] Following this cover story, we refer to female attackers and male victims throughout this paper.

1.2 Bootkit Attacks

The original evil maid attack was implemented against TrueCrypt in 2009 [16]. Earlier that year, another bootkit, called the *Stoned Bootkit* [31], had circumvented TrueCrypt as well. Until today, TrueCrypt is vulnerable to these attacks and the program's authors do not plan future improvements. To the contrary, they argue that bootkits "require the attacker to have [...] physical access to the computer, and the attacker needs you to use the computer after such an access. However, if any of these conditions is met, it is actually impossible to secure the computer". [39]

Microsoft's BitLocker, on the other hand, defeats software keyloggers up to a certain degree as it assures the integrity of the boot process by means of the trusted platform module (TPM). TPMs are used to build trusted checksums over sensitive boot parameters, such as firmware code, the BIOS, and the bootloader [40]. BitLocker's decryption key can only be derived if these checksums are in line with the reference configuration from system setup. Otherwise, users cannot decrypt their data and hopefully become suspicious that somebody manipulated their machine.

However, at the end of 2009, even BitLocker was successfully compromised by bootkit attacks. Türpe et al. [42] practically performed *tamper and revert* attacks that (1) tamper with the bootloader for introducing keylogging functionality, (2) let the victim enter his password into a forged text-mode prompt, (3) revert to the original bootloader, and (4) reboot. The victim may wonder about the reboot, but most likely he will enter his password again and proceed as usual – unaware of the fact that his password was already logged.

We reproduced both, the attack against TrueCrypt and the attack against BitLocker, with our own malicious bootloaders. Both attacks are still effective today.

1.3 Related Work: Attempts for Countermeasures

As shown by tamper and revert attacks, TPMs alone are *not* suitable to guarantee a trusted password prompt. Consequently, other countermeasures have been taken into consideration in recent years.

The first countermeasures were *external bootloaders* such as the Anti-Bootkit Project [2] from 2010. But even if the integrity of external USB bootloaders can be assured, because they are never left unattended, this measure is insufficient for several reasons: First, it remains unclear which USB port is preferred for booting. An attacker could plug in her own, tiny USB flash drive at the back of the machine. Second, BIOS passwords can often be reset by removing the onboard battery. This allows an attacker to reconfigure the boot sequence at her convenience, e.g., to boot from MBR. Third, the BIOS or UEFI might be manipulated and display a fake password prompt. And fourth, the entire target machine could simply be replaced with a manipulated model.

Another countermeasure named Anti Evil Maid [17] was introduced in 2011. This project mentions a *mutual authentication* scheme in the context of hard

disk encryption for the first time: Only if the boot process behaves with integrity, a secret (user-defined) message can be unsealed by means of the trusted platform module. This message must be shown to the user before he enters his password. If the secret message cannot be shown, the user is strongly advised not to enter his password because the boot process is likely to be compromised.

1.4 Possible Attacks against Anti Evil Maid

Considering travelers who boot their laptops at public places like airports, and in conference and meeting halls, it is unlikely that the authentication message from Anti Evil Maid remains confidential for a long time. It will be present on surveillance cameras, can be spied upon by shoulder surfing, and can perfectly be reconstructed as it is simple ASCII text.

Even worse, the authentication scheme of Anti Evil Maid is not secure if we take the confidentiality of the authentication message for granted. It is also not secure if we additionally assume that the sealed authentication message is placed on an external USB drive. Anti Evil Maid only raises the number of required physical accesses from two to three, because the attack by Türpe et al. [42] against BitLocker can be extended as follows:

1. On the first physical access, an evil maid manipulates the bootloader in a way that it copies USB drives, e.g., into the very last sectors of the hard drive or into another USB drive. Then she lets the victim plug in his USB drive and, after it has been cloned, the bootloader automatically reverts to its original state and reboots. The reboot occurs quickly after the copy procedure, and thus an unaware user may not even recognize suspicious behavior.
2. On the second physical access, the evil maid boots up her target with the recently cloned thumb drive and notes down the appearing authentication message. She builds her own bootloader with this message and overwrites the original bootloader a second time. The remaining attack equals the procedure of tamper and revert: The evil maid lets her victim enter his password and after it has been logged, the bootloader automatically reverts to its original state and reboots again. This time, the victim may wonder about the reboot, but most likely he will re-enter his password and proceed as usual.
3. On the third physical access, the evil maid can read out the logged password and so she gains full access to the data.

If a second password, e.g., in form of a PIN for the TPM, is used instead of external thumb drives, as alternatively proposed [17], the attack works analogously with a second password logger instead of the USB drive cloner. If images instead of simple text messages are used, as alternatively proposed [17], too, the attack works similarly as well, because appearing images can be photographed or digitally retrieved through cold boot attacks.

1.5 Contributions

We present the design and implementation of Stark, an authentication protocol that *fully* defends against targeted attacks with repeated physical access for the

first time. The goal of STARK is to reveal boot process manipulations before the user enters his password. To this end, we introduce a *mutual authentication scheme* that proves the integrity of a computer to its users. Similar to Anti Evil Maid, our authentication scheme is based on sealed authentication messages and external USB drives, but beyond that it introduces the concept of a *one-time password prompt*.

Roughly speaking, the authentication sequence of STARK works as follows:

- Upon boot, a user-defined authentication message is displayed. The message is sealed by the TPM and can only be unsealed if the boot process behaves with integrity. If the secret message is not displayed, the user must not enter his password.
- To prevent replay attacks as in Anti Evil Maid, we introduce *monces*: authentication messages that are used only once. Each time an authentication message has been displayed and verified, the user must define a new one. All monces are sealed by means of the TPM and stored on external USB drives. Upon boot, a consumed monce is always overwritten by a new one.
- The USB drive must be handled like a physical key and never be left unattended. The trustworthiness of our authentication scheme depends on the trust we have in the thumb drive (*trust bootstrapping*).
- Additionally, we store a sealed token value on the USB drive, to bind the drive necessarily to the decryption process (*two-factor authentication*).

In short, the trustworthiness of a machine state is authenticated to the user by displaying ever-changing authentication messages. Attacks against those messages must fail because an attacker can only retrieve messages that have already been consumed. To summarize, our contribution is threefold:

1. With STARK, we present a mutual boot authentication protocol between users and computers that is more secure than existing solutions (cf. Sect. 2).
2. We describe a practical implementation of STARK, for which we integrated the protocol into the bootloader of the FDE solution TRESOR [28] (cf. Sect. 3).
3. We give a formal argument for the security of STARK (cf. Appendix A).

Our implementation is open source (under GNU GPL v2) and together with other information available at `http://www1.cs.fau.de/stark/`.

2 STARK Protocol and Design Choices

We now describe the actual STARK protocol. STARK is a protocol for mutual authentication between users and computers. So there are two parties that follow the STARK protocol: The user $\mathcal{U}$ and the computer $\mathcal{C}$. Computer $\mathcal{C}$ must contain a TPM, because STARK requires its sealing capabilities to attest to $\mathcal{U}$ that it behaves with integrity. On the other hand, to attest to $\mathcal{C}$ that $\mathcal{U}$ is who he claims to be, traditional passwords are used.

2.1 The STARK Protocol

We formulate STARK in the standard notation of authentication protocols. STARK requires a bootstrapping phase, called session 0. Authentication sessions are numbered consecutively starting with session 1.

Bootstrapping phase. In the bootstrapping phase (see Fig. 1), $\mathcal{U}$ exchanges password p and the initial monce m_0 with $\mathcal{C}$ and receives $sealed_{\mathcal{C}}(m_0, t)$ from $\mathcal{C}$; t is a token value that binds the USB drive to the authentication process. Computer $\mathcal{C}$ permanently stores a hash sum $h(p, t)$ over p and the token value. After the setup phase, $\mathcal{U}$ and $\mathcal{C}$ can engage in an infinite sequence of authentication sessions.

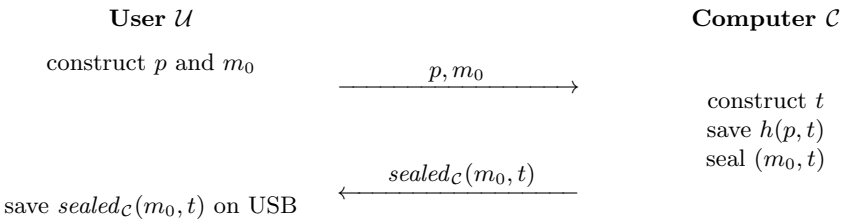

Fig. 1: Session 0 (bootstrapping phase).

Authentication sessions. In authentication session i (see Fig. 2), $\mathcal{U}$ plugs in his USB drive and boots up $\mathcal{C}$. Computer $\mathcal{C}$ reads out $sealed_{\mathcal{C}}(m_{i-1}, t)$, unseals it, and displays m_{i-1} to $\mathcal{U}$. If $\mathcal{U}$ does not recognize m_{i-1}, $\mathcal{U}$ must abort the protocol and consider $\mathcal{C}$ as compromised. Otherwise, if $\mathcal{U}$ recognizes m_{i-1}, $\mathcal{C}$ is authenticated and $\mathcal{U}$ can safely enter his password.

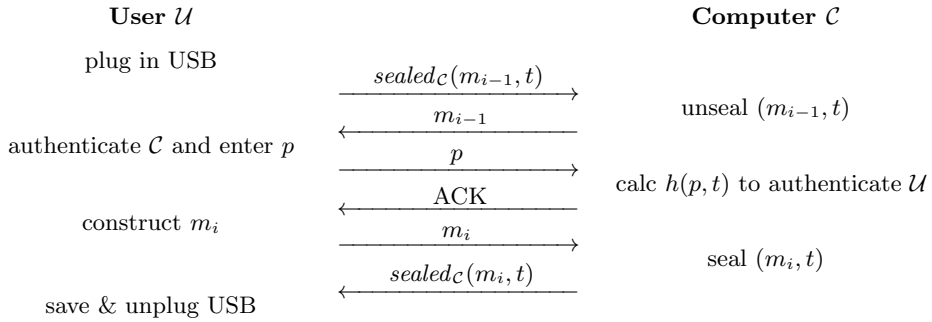

Fig. 2: Session i, $i \geq 1$ (authentication sessions).

Afterwards $\mathcal{C}$ computes $h(p,t)$ and compares it to the permanently saved value. If both values do *not* match, $\mathcal{U}$ is asked to enter his password again. Otherwise, $\mathcal{U}$ is now authenticated to $\mathcal{C}$, i.e., both parties are *mutually* authenticated. Only then $\mathcal{U}$ and $\mathcal{C}$ are safe to exchange a new monce m_i that will be used in session $i+1$. User $\mathcal{U}$ enters m_i, $\mathcal{C}$ seals it and then it is stored on the USB thumb drive. Finally, $\mathcal{U}$ can unplug the thumb drive and $\mathcal{C}$ passes control on to the operating system.

2.2 Security Argument

The central idea behind the security of Stark is that monces disallow replay attacks on the authentication message displayed to the user. To prove the security of the scheme, it is however vital that the user installs a new monce after every successful login without interference by the adversary. In traditional cryptographic protocols, replay attacks are oftentimes defeated by *nonces*, i.e., pseudo-random numbers that are used once. However, compared with traditional authentication schemes, the difficulty in the context of FDE is that one of the communication endpoints is *human*. Most humans have limited computational capabilities and therefore, authentication schemes requiring public/private key computations on both sides must be excluded and we introduce the concept of *monces*, i.e., messages that are used once. Basically, we replace the single authentication messages known from Anti Evil Maid with a series of continuously alternating messages.

In short, we use the sealing capabilities of the TPM to authenticate the computer towards the user. The user then authenticates towards the computer with traditional passwords as well as with a USB token. So upon each boot, the user gives two inputs: his password and a new authentication message m_i. Consequently evil maids can only catch outdated messages. We give a formal correctness argument for Stark in form of an inductive proof in Appendix A.

2.3 Design Components of Stark

We now describe the design choices that influenced Stark. There were two questions we had to answer when designing Stark. The first question was: What is the best way to achieve mutual authentication? Since users authenticate using a password, the problem of mutual authentication lies in having the computer prove its integrity first. We do this using a novel combination of the trusted platform module and one-time messages.

The second question was: How can we best protect the necessary authentication messages? Here we use a USB flash drive to hold the sealed monces. Another design decision, not directly related to the evil maid scenario, regards the use of token values, that are placed on the USB drive to protect users who choose weak passwords (two-factor authentication).

Mutual Authentication by means of the Trusted Platform Module. In practice all implementations to date that are based upon TPMs fail to attest the trustworthiness of boot parameters in a way that cannot be tampered. An example for this are attacks by Türpe et al. against BitLocker as described above. Even though BitLocker builds upon TPMs, users can easily be tricked into bogus password prompts because BitLocker looks exactly the same on every machine. An attacker can reproduce the original BitLocker prompt perfectly in her own bootloaders and victims have no chance to distinguish malicious from benign behavior. However, TPMs are neither insecure nor useless taken by themselves, they must just carefully be integrated into protocols. In STARK, the TPM is used to protect one-time messages and bind them to the computer at the same time.

Bootstrapping Trust from USB Flash Drives. We store sealed monces on an external USB thumb drive. If the monces were stored on the local hard drive, STARK would be insecure to attackers who simply boot the machine and note down the appearing monce, because such attackers could build their own bootloaders with the exactly same message. If only the laptop *or* the USB drive is missing, monces remain confidential because they can only be unsealed if both entities act together. Practically, that means we can leave PCs unattended when we assure the trustworthiness of USB drives. In this sense, STARK is a protocol that performs *trust bootstrapping* from trusted USB drives.

Two-Factor Authentication with Passwords and Token Values. Besides the machine state and the user password, the FDE decryption key is based on a token value stored on the USB drive. Similar to monces, the token is sealed by means of the TPM and can only be unsealed if the machine is in line with its reference configuration. While all components introduced above defeat bootkit attacks, token values prevent dictionary attacks against weak passwords. Without a token value, adversaries would be able to boot their targets and enter passwords one after another. Tokens are also a protection in scenarios where the user disclosed his password but did not lose the thumb drive. This can be the case due to social engineering [38], skimming and shoulder surfing [23, 4], reusing the password in a different system [15], or writing it down. Token values prevent such attacks because they bind USB drives compulsorily to the decryption process.

3 POTTS: A Linux-based Implementation of STARK

We now present POTTS, our practical implementation of the authentication protocol given above.[2] POTTS ***p**rotects **o**pportunistic and **t**argeted **t**hreat **s**cenarios* through the integration of TRESOR and STARK.

[2] Note that Pepper Potts is the personal secretary of Tony Stark [10].

3.1 Integrating STARK with TRESOR

When an attacker gains physical access to a machine while it is *running* or in *standby mode*, it is not protected by STARK but must be protected by separate mechanisms. Therefore we integrated STARK into TRESOR [28] because TRESOR already defeats another kind of physical access attacks, namely cold boot attacks [13].

Cold boot attacks exploit the *remanence effect* of DRAM [12], meaning that RAM contents fade away gradually over time. Due to the remanence effect, encryption keys can be restored from memory through rebooting a system with malicious USB drives, or by replugging RAM chips physically into another machine. That makes cold boot attacks rather generic and therefore they constitute a threat for *all* software-based FDE technologies to date, including BitLocker and TrueCrypt.

TRESOR, as a countermeasure, runs encryption securely outside RAM. To this end, TRESOR avoids RAM usage entirely and runs the key management, as well as the AES algorithm, solely on the microprocessor. Some processor registers (in detail, these are the debug registers) are permanently used as cryptographic key storage. TRESOR is implemented as a Linux kernel patch because side effects like context switching, which leak information into RAM, cannot be controlled in user mode.

3.2 STARK Protocol Extensions

In the presentation of STARK above we reduced the protocol to the communication of authentication messages between $\mathcal{U}$ and $\mathcal{C}$. However, we ignored some real-world aspects, like error recovery and usability, to make the protocol more amendable for its security analysis (cf. Appendix A). We now make up for those aspects and present protocol extensions of STARK within POTTS.

Data Rescue and Recovery Mechanism. To recover from protocol failures, from hardware configuration changes, and from system compromise, we use two different keys in POTTS: a *key encryption key* (KEK) and a *data encryption key* (DEK). KEK k is composed of token value t and a checksum over password p. KEK k decrypts DEK d which is encrypted using AES and therefore is safe to be stored on hard disk. DEK d is used to decrypt user data.

While k may change frequently, e.g., through password changes, d is designed to be constant. So d prevents cumbersome re-encryption of the disk in the case of password changes since only d must newly be encrypted. Furthermore, d allows for data recovery in the case of password loss, and in the case that USB drives are lost or TPM hardware failures occur. For data recovery, users must store d in plaintext at a physically secure place, e.g., in a vault at home. This allows for migration of the hard disk into other machines and can be consulted in the case of hardware configuration changes that lead to different TPM states.

Not all protocol inconsistencies must necessarily point to a system compromise but can arise from technical problems, too. However, this is hard to differentiate

in practice, and thus we advise users to act carefully: In practice, data can be recovered with d by mounting potentially compromised HDDs on a second, trusted computer as an external device. After data rescue, the HDD in question must be formatted, including the master boot record, and the PC must be reset to factory settings (including the BIOS or UEFI image). Only then, STARK's bootstrapping phase can be rerun securely to set up a new system. Overall, STARK enables users to *identify* a potential system compromise; we do not strictly regulate which actions to take if such a compromise is detected in POTTS.

Key Derivation Function. In the previous section we stated that k is derived from a checksum over the user password p and the token value t. We now specify the key derivation function used in POTTS in detail. We do not use a single SHA-256 checksum over p but run the *password-based key derivation function 2* (PBKDF2) from RSA laboratories [20], which is recommended by NIST [41].

PBKDF2 applies a cryptographic hash to the password along with a salt and repeats the procedure several times to derive the final key (*key stretching*). The salt as well as the additional iterations make brute force attacks on the password difficult. Salt values reduce the risk of precomputed tables (*rainbow tables*) that allow attackers to look up hash values of frequently used passwords. Additionally, several iterations slow down the brute force approach.

PBKDF2 library calls are usually parameterized with *algorithm*, *password*, *salt*, *iterations*, and *keylen*. In POTTS we have chosen the variant

$$k = pbkdf2(\text{HMAC-SHA-256}, p, t, 4096, 256)$$

to derive KEK k from password p. We use the token value t as salt, perform 4096 iterations of SHA-256, and get a final keysize of 256 bits. The decrypted DEK variant d is then derived from its encrypted variant D via

$$d = decrypt(\text{AES-256}, k, D)$$

meaning that the DEK is encrypted with k and AES-256. Decrypted DEK d is a 256-bit value and used to encrypt the disk. In the case that AES-128 or AES-192 is used, superfluous key bits are just ignored.

Usability vs. Security. We now discuss some obvious limitations concerning the tradeoff between security and usability. A reasonable argument against the practical applicability of STARK might be the additional overhead we demand from users by regularly defining new monces. To be able to generate and remember different monces quickly, we advise users to write sentences about their current activities, the places they have been during the day, or their personal sentiments.

However, to increase the usability (at expense of security) in POTTS, we allow users to skip the creation of new monces and to continue with the current monce by pressing F12. Skipping new monces can be harmless in many practical scenarios, because the actual threat situation varies heavily with the physical environment. Monces that have been displayed on a business trip should always

be exchanged, of course, but monces that were displayed at home can mostly be considered as confidential.

If a user skips monces permanently, the authentication scheme of POTTS falls back to that of Anti Evil Maid. But skipping monces carefully, users have a flexible authentication scheme that is both convenient in unthreatened situations and secure in threatened situations. That is, POTTS can be adapted to the user's environment on a daily basis.

The POTTS Protocol. POTTS uses a setup phase in which $\mathcal{U}$ enters secret password p and authentication message m_0. $\mathcal{C}$ generates the 256-bit token t and the 256-bit data encryption key d. $\mathcal{C}$ then seals m_0 and t and gives the sealed tuple $sealed_{\mathcal{C}}(m_0, t)$ back to $\mathcal{U}$ who stores it on USB drive. DEK d is encrypted by means of KEK k, as described above, and can safely be stored inside a crypto footer of the disk drive. The disk drive is encrypted with TRESOR, i.e., with AES-128, -192, or -256.

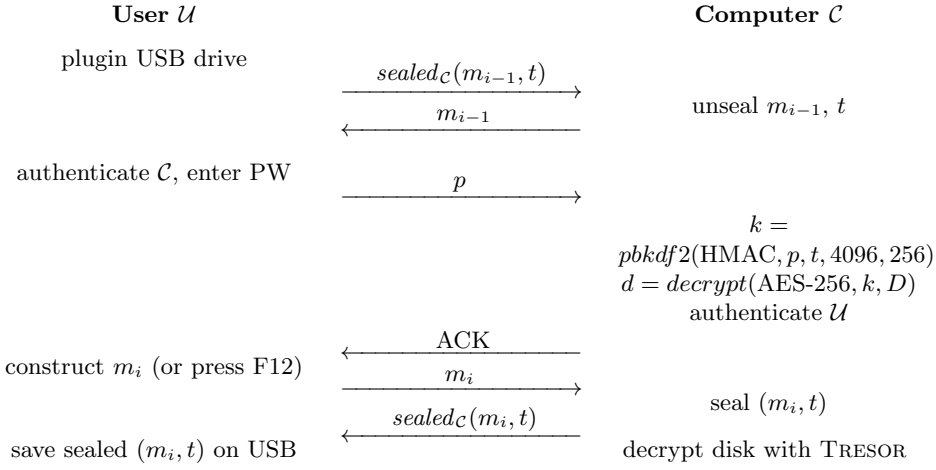

Fig. 3: POTTS authentication session i for $i \geq 1$.

After the setup phase, $\mathcal{U}$ and $\mathcal{C}$ can engage in an infinite sequence of mutual authentication sessions. The extended protocol of POTTS for session i, $i \geq 1$ is given in Fig. 3. $\mathcal{C}$ derives KEK k via PBKDF2 from p and t, so that d can be decrypted. With d, a block of the hard disk is decrypted exemplarily in order to verify the correctness of p. If p is wrong, the user is asked to enter his password again.

3.3 Potts Implementation Details

The integration of Stark into Tresor was clearly motivated by the fact that we wanted to build an FDE system which is more secure than common FDE solutions in use today. Cold boot attacks do not fall into the defined threat model of this paper, but combining the concepts of Stark with that of Tresor seemed reasonable. We now give some more information about the implementation of Potts. The full details are contained in our open source distribution available online.

Potts is based on *two* Linux kernels: (1) a minimized kernel for the Stark authentication which is loaded as first stage, and (2) the Tresor patched kernel which is loaded as second stage. The reason behind this design choice was that TPM toolchains like TrouSerS [1] are available for user mode only, and that talking to the TPM in a pre-boot environment turned out to be difficult. In a nutshell, the boot process of Potts follows the following sequence:

- Users plug in a bootable USB drive that contains *TrustedGRUB*. TrustedGRUB is an extended version of the known bootloader GRUB with TPM support. This makes a secure bootstrap architecture possible because TrustedGRUB measures the whole boot process and makes it verifyable for system integrity checks.
- From TrustedGRUB we boot into the first Linux system. The kernel and the user mode initialization of the first system are optimized for speed and it takes less than 3 seconds to boot it.
- In the first stage we display an *ncurses*-based interface that encapsulates the Stark protocol in a user-friendly way. For example, the interface displays the recently defined monce, asks for the password, and asks for a new monce (if F12 is not pressed). Internally, the Stark protocol is implemented via *tpm_sealdata* and *tpm_unsealdata* from the TPM toolchain. For the key derivation function PBKDF2 we make use of the OpenSSL library.
- With the recently derived key, the second kernel, which is stored on hard drive, is decrypted. After the authentication, we boot into this second kernel via *kexec*, a Linux command that replaces the currently running OS with a new OS without actually rebooting the system.
- After the second stage booted into full (Tresor-based) Linux, end-users can use their systems as usual. Internally we pass on the decryption key to the Tresor system over the debug registers of CPU0. Incidentally, *kexec* does not reset the debug registers of CPU0 while booting into a new kernel. All residues of the key and the password are wiped out from RAM between the first and the second stage.

Besides the simplicity of the implementation, the design choice provides us with a flexible system for future enhancements. Since passwords are not entered in pre-boot environments but in full Linux systems, we can easily add functionality to bootstrap trust from smartphones (cf. Sect. 4.2), for example.

4 Limitations and Outlook

To conclude, we give an outlook on current limitations (Sect. 4.1) and possible future enhancements (Sect. 4.2).

4.1 Limitations: Hardware-based Attacks

Stark defeats traditional evil maid attacks that require physical access to the machine. In practice, however, the borderline from software-only attacks to more complex, hardware-based attacks is blurred. For example, hardware-based attacks can install hidden USB or PS/2 keyloggers, even to laptops [22]. Such keyloggers are generally hard to defeat in software. Academic attempts for detecting hardware keyloggers in software exist [27], but they are hardware-specific and cannot be generalized well. Another countermeasure against hardware keylogging, which is often used by online banking software, is the use of on-screen keyboards or randomized keyboard mappings. However, such countermeasures can in turn be attacked through VGA and DVI loggers [21]. So hardware-based attacks cannot be counteracted well by software solutions. In the abstract model of Stark, attackers can arbitrarily manipulate the software configuration, including the BIOS and the MBR, but they cannot manipulate hardware that is not measured by the TPM.

4.2 Outlook: Taking Advantage of New Hardware

An interesting question is whether the upcoming technology *UEFI secure boot* [43] is an alternative to Stark on future PCs. Secure boot will be supported by Windows 8 for the first time, meaning that the Windows 8 bootloader will be signed by Microsoft, and the underlying UEFI can refuse unsigned bootloaders [37]. While a TPM is a *passive* module and neither halts the computer, nor warns the user when the current configuration differs from its reference configuration [34], secure boot extends the trusted computing architecture by active components. However, UEFI images could still be replaced or manipulated with physical access and, consequently, users can still be tricked into bogus password prompts. Only Stark pursues the idea of authenticating the machine state to the user first. While secure boot stops malicious bootloaders from getting loaded, it does not prove the system state to users. Hence, Stark should be combined with secure boot in the future, but it cannot be fully replaced by secure boot.

Besides that, we plan to extend the idea of *trust bootstrapping*. Currently, we use passive USB devices as confidential storage for sealed monces. For the future, we consider more intelligent devices as confidential, e.g., Android-based smartphones which can always be carried along like physical keys, too, in order to bootstrap trust from them. Such devices have the advantage that they come with computing capabilities that allow for more advanced, RSA-based authentication schemes and can eliminate the need for monces.

References

1. TrouSerS: The open-source TCG Software Stack. http://trousers.sourceforge.net/.
2. Ebfe's Anti-Bootkit Project. http://ebfes.wordpress.com/tag/bootloader/, 2010.
3. Andreas Galauner. EFI Rootkits: Pwning your OS before it's even running. Tech. rep., dexlabs.org, 2012. SIGINT 2012.
4. Asonov, D., and Agrawal, R. Keyboard Acoustic Emanations. Tech. rep., IBM Almaden Research Center, San Jose, CA, 2004. IEEE Symposium on Security and Privacy, IEEE Computer Society.
5. Böck, B. *Firewire-based Physical Security Attacks on Windows 7, EFS and BitLocker*. Secure Business Austria Research Lab, Aug. 2009.
6. Break & Enter. Adventures with Daisy in Thunderbolt-DMA-Land: Hacking Macs through the Thunderbolt interface, Feb. 2012.
7. Brian D. Carrier and Eugene H. Spafford. Getting Physical with the Digital Investigation Process. *IJDE 2*, 2 (2003).
8. Carbone and Bean and Salois. An in-depth analysis of the cold boot attack. Tech. rep., DRDC Valcartier, Defence Research and Development, Canada, Jan. 2011. Technical Memorandum.
9. Christophe Devine and Guillaume Vissian. Compromission physique par le bus PCI. In *Proceedings of SSTIC '09* (June 2009), Thales Security Systems.
10. Favreau, J. Iron Man (movie). Paramount Pictures, 2008.
11. FIPS. Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, NIST, Nov. 2001.
12. Gutmann, P. Data Remanence in Semiconductor Devices. In *Proceedings of the 10th USENIX Security Symposium* (Washington, D.C., Aug. 2001), USENIX Association.
13. Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W. Lest We Remember: Cold Boot Attacks on Encryptions Keys. In *Proceedings of the 17th USENIX Security Symposium* (San Jose, CA, Aug. 2008), Princeton University, USENIX Association, pp. 45–60.
14. Intel Corporation. *Solid-State Drive 520 Series*, 2012. http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-520-series.html.
15. Ives, B., Walsh, K. R., and Schneider, H. The domino effect of password reuse. In *Communications of the ACM* (Apr. 2004), vol. 47/4.
16. Joanna Rutkowska. Evil Maid goes after TrueCrypt, Oct. 2009. The Invisible Things Lab.
17. Joanna Rutkowska. Anti Evil Maid, Sept. 2011. The Invisible Things Lab.
18. John Heasman. Implementing and Detecting an ACPI BIOS Rootkit. Tech. rep., NGS Consulting, 2006. BlackHat Briefings, Europe.
19. Johnson, C. *Protection of Sensitive Agency Information*. U.S. Executive Office of the President, Washington, D.C. 20503, June 2006.
20. Kaliski. PKCS #5: Password-Based Cryptography Specification. In *Request for Comments (RFC): 2898*, Internet Engineering Task Force, Ed., vol. 2.0. RSA Laboratories, 2000.
21. KeeLog. Video Ghost. http://www.keelog.com/hardware_video_logger.html, 2012.
22. KeyGhost Ltd. PCI / Mini-PCI Hardware Keylogger. http://www.keyghost.com/PCI-MPCI-Keylogger.htm, 2006.
23. Kuhn, M. G. Optical Time-Domain Eavesdropping Risks of CRT Displays. Tech. rep., University of Cambridge, Computer Laboratory, Berkeley, California, May 2002. Proceedings 2002 IEEE Symposium on Security and Privacy (SSP 02).

24. Kumar, N., and Kumar, V. VBootKit 2.0 - Attacking Windows 7 via Boot Sectors. In *Hack In The Box Conference (HITBSecConf)* (Dubai, Apr. 2009).
25. Li, X., Wen, Y., Huang, M., and Liu, Q. An Overview of Bootkit Attacking Approaches. In *Seventh International Conference on Mobile Ad-hoc and Sensor Networks (MSN'11)* (2011), IEEE Computer Society, pp. 428–431.
26. Loukas K. DE MYSTERIIS DOM JOBSIVS – Mac EFI Rootkits. Tech. rep., assurance, 2012. Black Hat Conference Proceedings, USA.
27. Mihailowitsch, F. Detecting Hardware Keyloggers. In *HITB SecConf* (Kuala Lumpur, Malaysia, Oct. 2010), cirosec GmbH. Hack In The Box.
28. Müller, T., Freiling, F., and Dewald, A. TRESOR Runs Encryption Securely Outside RAM. In *20th USENIX Security Symposium* (San Francisco, California, Aug. 2011), University of Erlangen-Nuremberg, USENIX Association.
29. Müller, T., Latzo, T., and Freiling, F. Hardware-based Full Disk Encryption (In)Security Survey. Tech. rep., Friedrich-Alexander University of Erlangen-Nuremberg, Sept. 2012. Technical Report.
30. Panholzer, P. Physical Security Attacks on Windows Vista. Tech. rep., SEC Consult Vulnerability Lab, Vienna, May 2008.
31. Peter Kleissner. Stoned Bootkit, July 2009. Black Hat, USA.
32. Ponemon Institute, LLC. *2010 Annual Study: U.S. Cost of a Data Breach.* Symantec, Mar. 2011.
33. Robert David Graham. Thunderbolt: Introducing a new way to hack Macs, Feb. 2011. Errata Security.
34. Rutkowska, J., Tereshkin, A., and Wojtczuk, R. Thoughts about Trusted Computing. In *EUSecWest 2009* (May 2009), The Invisible Things Lab.
35. Sacco, A. L., and Ortega, A. A. Persistent BIOS Infection: The early bird catches the worm. In *Proceedings of the Annual CanSecWest Applied Security Conference* (Vancouver, British Columbia, Canada, 2009), Core Security Technologies.
36. SECUDE. *US Full Disk Encryption 2011 Survey.* Research SECUDE AG, 2012.
37. Software Freedom Law Center. Microsoft confirms UEFI fears, locks down ARM devices. Tech. rep., Jan. 2012.
38. Thornburgh, T. Social engineering: the Dark Art. Tech. rep., New York, NY, USA, 2004. Proceedings of the 1st Annual Conference on Information Security Curriculum Development (InfoSecCD 04).
39. TrueCrypt Foundation. TrueCrypt: Free Open-Source On-The-Fly Disk Encryption Software for Windows 7/Vista/XP, Mac OS X and Linux. http://www.truecrypt.org/, 2012.
40. Trusted Computing Group, Incorporated. TPM Main Specification. Tech. Rep. Specification Version 1.2, rev. 116, TCG Published, Mar. 2011.
41. Turan, M., Barker, E., Burr, W., and Chen, L. Special Publication 800-132: Recommendation for Password-Based Key Derivation. Tech. rep., NIST, Computer Security Division, Information Technology Laboratory, Dec. 2010.
42. Türpe, S., Poller, A., Steffan, J., Stotz, J.-P., and Trukenmüller, J. Attacking the BitLocker Boot Process. In *Trusted Computing Second International Conference TRUST* (Oxford, UK, Apr. 2009), L. Chen, C. J. Mitchell, and A. Martin, Eds., vol. 5471, Fraunhofer Institute for Secure Information Technology (SIT), Springer, pp. 183–196.
43. Unified EFI, Inc. *Unified Extensible Firmware Interface Specification*, Ver. 2.3.1, Errata B ed., Apr. 2012.

A Formal Security Argument

In this section, we formalize STARK as a security protocol within an abstract system model. To this end, we add a third player: attacker $\mathcal{A}$ who wants to break the authentication scheme between $\mathcal{U}$ and $\mathcal{C}$. Parties $\mathcal{U}$ and $\mathcal{C}$ follow the protocol given above and additional rules given below (Sect. A.1, Sect. A.2). Attacker $\mathcal{A}$ can act arbitrarily under the restrictions described below (Sect. A.3).

In the formal model of STARK, parties communicate by exchanging messages in an abstract sense. This can correspond to typing text on the keyboard, displaying a message on the screen, or attaching a USB thumb drive. Due to the immediate geographical vicinity of all parties, we assume that message exchange is reliable and synchronous, meaning that if a party sends message m, the other party receives m within a short delay.

The senders of individual messages cannot be authenticated, i.e., the receiver of a message does not necessarily know who sent the message. For example, $\mathcal{C}$ cannot distinguish text typed by $\mathcal{A}$ or $\mathcal{U}$. Similarly, a message prompt shown to $\mathcal{U}$ may come from $\mathcal{C}$ (using the original boot loader) or from $\mathcal{A}$ (using a forged boot loader).

A.1 User Model

We now give the rules that $\mathcal{U}$ has to conform to. $\mathcal{U}$ corresponds to a human user sitting in front of the keyboard and wishes to authenticate securely to $\mathcal{C}$. The security of STARK relies on the following rules for $\mathcal{U}$:

- *Completeness*: $\mathcal{U}$ must complete the protocol and define a new m_i as soon as he started session i and consumed m_{i-1}.
- *Singularity*: Every monce m_i chosen by $\mathcal{U}$ must be unpredictable and used only once.
- *Coherence*: $\mathcal{U}$ must not leave the computer during the authentication phase but execute all steps consecutively.
- *Correctness*: If $\mathcal{U}$ cannot authenticate $\mathcal{C}$, $\mathcal{U}$ must abort the protocol entirely and never engage in the protocol with the same participants again.

If any of these rules are violated, the security of STARK cannot be proven, and disk encryption may become insecure. In order to fulfill completeness, e.g., after $\mathcal{C}$ crashed due to a power cut, $\mathcal{U}$ is allowed to restart STARK at any point and to consume m_{i-1} a second time if our assumption about coherence is not violated.

A.2 Computer Model

We now define the possibilities for $\mathcal{C}$. Unlike $\mathcal{U}$, $\mathcal{C}$ is an electronic system that has powerful computation capabilities, including crucial cryptographic primitives. STARK relies on the following properties of $\mathcal{C}$:

- *Integrity*: $\mathcal{C}$ must not be compromised initially during setup phase.

– *Crypto Competence*: We assume that $\mathcal{C}$ can *seal* information in the following sense: Given a precise software configuration c of $\mathcal{C}$, $\mathcal{C}$ can encrypt and sign a message m such that $\mathcal{C}$ itself can decrypt it only when being in c again. It is unfeasible to seal or unseal information for any party besides $\mathcal{C}$ as well as for $\mathcal{C}$ itself if it is not in c.
– *Reliability*: $\mathcal{C}$ waits indefinitely for message m_i and password p to be entered without shutting down. Reading from and writing to a USB thumb drive does not fail, neither do other electronic operations fail.

Intuitively, the crypto competence of $\mathcal{C}$ corresponds to the sealing capabilities of trusted platform modules, meaning that $\mathcal{C}$ is required to have a TPM. Most notably, configuration c encompasses the BIOS settings and the master boot record. The reliability of $\mathcal{C}$ is an interesting property, both in practice as in theory, because it is generally hard to differentiate between malicious and faulty behaviors. This makes secure recovery mechanisms in the case of accidental data corruption difficult (see Sect. 3.2).

A.3 Attacker Model

Attacker $\mathcal{A}$ corresponds to an evil maid who may additionally be equipped with electronic devices. So $\mathcal{A}$ can perform both human actions as well as computationally complex calculations. Overall, $\mathcal{A}$ acts arbitrarily under the following restrictions:

– Attacker $\mathcal{A}$ can let $\mathcal{C}$ unseal any data for her (when she plugs in a USB drive and boots $\mathcal{C}$), but she cannot break cryptography, i.e., she cannot derive the password from a hash, and she cannot seal data herself.
– Attacker $\mathcal{A}$ can inject or replay arbitrary messages before and after the authentication process of Stark, and she can start an authentication process herself. But she cannot send messages to $\mathcal{U}$ or $\mathcal{C}$ within complete protocol parts, i.e., she is not allowed to interrupt the authentication process of $\mathcal{U}$. This corresponds to the event that $\mathcal{A}$ interferes with $\mathcal{U}$ while he boots up $\mathcal{C}$, which we exclude.
– Attacker $\mathcal{A}$ cannot get physical access to the USB drive. Nevertheless, she can make a one-to-one copy of it after manipulating the MBR (as sketched in the attacks from Sect. 1.4); so she may know all monces up to m_{i-2}.

A.4 Security Argument

The security argument is by induction. We argue that over an infinite sequence of sessions, the following invariant is maintained at the beginning of every session i (the security of Stark follows from item 3 of the invariant):

1. User $\mathcal{U}$ knows p and m_{i-1} and owns $\mathit{sealed}_{\mathcal{C}}(m_{i-1})$ on a USB drive.
2. Attacker $\mathcal{A}$ does not know m_{i-1}.
3. Attacker $\mathcal{A}$ does not know p.

Note that token t does not play a role in our security model, because t does not add security to the evil maid scenario, but in other scenarios as described above.

We now prove that STARK maintains the invariant by induction over the number of sessions. The base case (session 0) is straightforward and follows from the setup phase: $\mathcal{U}$ knows m_0 and has it sealed by $\mathcal{C}$, and the attacker $\mathcal{A}$ can neither have the password p nor the monce m_0 because of the initial integrity of $\mathcal{C}$.

For the induction step, assume that the invariant holds at the beginning of session i. There are two possibilities: Either $\mathcal{A}$ manipulates $\mathcal{C}$ and tries to inject messages, or $\mathcal{U}$ starts the protocol with regular behavior of $\mathcal{C}$, possibly after a reboot, and possibly after his USB drive was cloned. We discuss both cases in turn:

- Case 1 ($\mathcal{A}$ manipulates $\mathcal{C}$ and tries to inject messages): Since $\mathcal{A}$ does not know m_{i-1} (from invariant) and m_{i-1} cannot be guessed (from singularity of monces), any value injected by $\mathcal{A}$ towards $\mathcal{C}$ results in a wrong monce displayed to $\mathcal{U}$. $\mathcal{C}$ cannot unseal $\mathit{sealed}_{\mathcal{C}}(m_{i-1})$ because it is not in line with its reference configuration (from crypto competence of $\mathcal{C}$). Consequently, the protocol will be aborted and never restarted with the engaged parties (from correctness of user behavior).
- Case 2 ($\mathcal{U}$ starts the protocol with regular behavior of $\mathcal{C}$, possibly after a reboot): $\mathcal{A}$ may know $\mathit{sealed}_{\mathcal{C}}(m_{i-1})$, and hence, $\mathcal{A}$ can learn m_{i-1} from $\mathcal{C}$. However, $\mathcal{A}$ cannot inject any messages because $\mathcal{U}$ started the protocol (from coherence of user behavior). $\mathcal{U}$ successfully authenticates towards $\mathcal{C}$ using his password p, and completes the protocol by defining m_i (from completeness of user behavior), such that $\mathit{sealed}_{\mathcal{C}}(m_i)$ is stored on a USB drive (from reliability of $\mathcal{C}$).

In both cases, session i completes and session $i+1$ commences. Before the new session starts, observe the following points:

1. From session i, $\mathcal{U}$ knows m_i and has a sealed version of it. Of course, $\mathcal{U}$ still knows p.
2. Attacker $\mathcal{A}$ does not know m_i. Indeed, $\mathcal{A}$ may know m_{i-1}, but $\mathcal{A}$ cannot exploit it to fool $\mathcal{U}$ because $\mathcal{U}$ already defined m_i.
3. Attacker $\mathcal{A}$ does not know password p (from invariant).

Overall the invariant still holds at the beginning of session $i+1$, concluding the proof. Crucial for our argument is that *complete protocol phases* restrict the behavior of adversary $\mathcal{A}$. As explained above, we assume that $\mathcal{U}$ completes session i, meaning that when $\mathcal{U}$ executes the first step of a session, $\mathcal{U}$ does not abort but runs the protocol to completion. Without the completeness property, STARK would be insecure.

Acknowledgments. We would like to thank Stefan Vömel and Johannes Bauer for helpful comments on prior versions of this paper.