Gracewipe: Secure and Verifiable Deletion under Coercion

Lianying Zhao and Mohammad Mannan
Concordia Institute for Information Systems Engineering
Concordia University, Montreal, Canada
{z_lianyi, mmannan}@ciise.concordia.ca

***Abstract*—For users in possession of password-protected encrypted data in persistent storage (i.e., "data at rest"), an obvious problem is that the password may be extracted by an adversary through dictionary attacks or by coercing the user. Techniques such as multi-level hidden volumes with plausible deniability, or software/hardware-based full disk encryption (FDE) cannot adequately address such an attacker. For these threats, making data verifiably inaccessible in a quick fashion may be the preferred choice, specifically for users such as government/corporate agents, journalists, and human rights activists with highly confidential secrets, when caught and interrogated in a hostile territory. Using secure storage on a Trusted Platform Module (TPM) and modern CPU's trusted execution mode (e.g., Intel TXT), we design *Gracewipe* to enable secure and verifiable deletion of encryption keys through a special deletion password. An attacker cannot distinguish between a deletion and real password. He can guess the real password to unlock the target encryption key only through the valid Gracewipe environment; guessing the deletion password will trigger deletion of the real key. When coerced, a user can fake compliance, and enter the deletion password; and then the user can prove to the attacker that Gracewipe has been executed and the real key is no longer available (through a TPM quote), hoping that a reasonable adversary then will find no reason to keep holding the victim, and may even release her. We implement two prototypes of Gracewipe: software-based FDE system with plausible deniability (using TrueCrypt with hidden volume), and hardware-based FDE (using a Seagate self-encrypting drive (SED)). Our choice of booting Windows at the end of a Gracewipe session (for the possibility of immediate adoption), poses some unique challenges. Through the design and prototypes of Gracewipe, we hope to raise awareness of a special but critical use-case of FDE systems.**

I. Introduction and Motivation

Plausibly deniable encryption (PDE) schemes for file storage were proposed more than a decade ago; see Anderson et al. [5] for the first academic proposal (1998). In terms of real-world PDE usage, TrueCrypt [54] is possibly the most-widely used tool, available since 2004. Several other systems also have been proposed and implemented. All these solutions share an inherent limitation: an attacker can detect the existence of such systems (see e.g., TCHunt [3]). A user may provide *reasonable* explanation for the existence of such tools or random-looking free space; e.g., claiming that TrueCrypt is used only as a full-disk encryption (FDE) tool, no hidden volumes exist; or, the random data is the after-effect of using tools that write random data to securely erase a file/disk. However, a coercive attacker may choose to detain and punish a suspect up until the true password for the hidden volume is revealed, or up to a time period as deemed necessary by the attacker. Such coercion is also known as *rubberhose cryptanalysis* [37], which is alleged to be used in some countries (e.g., Turkey [12]). The use of multiple hidden volumes or security levels (e.g., as in StegFS [32]), may also be of no use if the adversary is patient. Another avenue for the attacker is to derive candidate keys from a password dictionary, and keep trying those keys, i.e., a classic offline dictionary attack. If the attacker possesses some knowledge about the plaintext, e.g., the hidden volume contains a Windows installation, such guessing attacks may (easily) succeed against most user-chosen passwords.

Another option for the victim is to provably destroy/erase data when being coerced. Note that such coercive situations mandate a very quick response time from tools used for erasure irrespective of media type (e.g., magnetic or flash); i.e., tools such as ATA secure erase, and DBAN [15] that rely on data overwriting are not acceptable solutions (cf. [21]). Otherwise, the attacker can simply terminate the tool being used by turning off power, or make a backup copy of the target data first. The need for rapid destruction was recognized by government agencies decades ago; see Slusarczuk et al. [48]. For a quick deletion, cryptographic approaches appear to be an appropriate solution, as introduced by Boneh and Lipton [8] (see also [13, 39]). Such techniques have also been implemented by several storage vendors in solid-state/magnetic disk drives that are commonly termed as self-encrypting drives (SEDs); see, e.g., Seagate [46], HGST/Western Digital [23] (cf. ISO/IEC WD 27040 [27]). SEDs allow overwriting of the data encryption key via an API call. Currently, as we are aware of, no solutions offer pre-OS secure erase that withstand coercive threats (i.e., with undetectable deletion trigger). Even if such a tool is designed, still several issues remain: verifiable deletion is not possible with SEDs alone (i.e., how to ensure that the secure erase API has been executed); and implicit deletion of the key without the adversary being notified is not possible, as required in deletion under coercion (i.e., calls to the deletion API can be monitored via SATA/IDE interface). We use SEDs as part of our solution without directly depending on their key deletion API.

In this paper, we discuss the design and implementation

NDSS '15, 8-11 February 2015, San Diego, CA, USA
 ISBN 1-891562-38-X
http://dx.doi.org/10.14722/ndss.2015.23258

of *Gracewipe*, a solution implemented on top of TrueCrypt[1] and SEDs that can make the encrypted data permanently inaccessible without exposing the victim. When coerced to reveal her hidden volume encryption password, the victim will use a special pre-registered password that will irrecoverably erase the hidden volume key. The coercer cannot distinguish the deletion password from a regular password used to unlock the hidden volume key. After deletion, the victim can also prove to the coercer that Gracewipe has been executed, and the key cannot be recovered anymore. A trusted hardware chip such as the Trusted Platform Module (TPM) alone cannot realize Gracewipe, as current TPMs are passive (i.e., run commands as sent by the CPU), and are unable to execute external custom logic. To implement Gracewipe, we use TPM along with Intel trusted execution technology (TXT), a special secure mode available in several consumer-grade Intel CPU versions (similar to AMD SVM).

The basic logic in Gracewipe for a PDE-enabled FDE system (e.g., TrueCrypt) can be summarized as follows. A user selects three passwords during the setup procedure: (i) Password PH that unlocks only the hidden volume key; (ii) Password PN that unlocks only the decoy volume key; and (iii) Password PD that unlocks the decoy volume key and overwrites the hidden volume key (multiple PDs may also be used; cf. [11]). These passwords and volume keys are stored as TPM-protected secrets that cannot be retrieved without defeating TPM security. Depending on the scenario, the user will provide the appropriate password. The attacker can coerce the victim to reveal all three passwords, but she must rely on the victim to identify which one will unlock the hidden volume. Random guessing will allow a success probability of $1/3$ (if all three passwords are extracted from the victim), but with the same probability, the data may be irrecoverably destroyed.[2] Overwrite of the hidden volume key occurs within the hardware chip, an event we assume to be unobservable to the attacker. Now, the attacker does not enjoy the flexibility of password guessing without risking the data being destroyed. For a regular FDE system (e.g., SEDs), the decoy volume is not needed, and as such, Gracewipe will require only two passwords (PH and PD).

The relatively simple design of Gracewipe however faced several challenges when implemented with real-world systems such as TrueCrypt and SEDs. As Gracewipe works in the pre-OS stage, no ready-made TPM interfacing support is available (especially for some complex TPM operations, see Section IV). We have to construct TPM protocol messages on our own. Also, as we combine multiple existing tools in a way they were not designed for, e.g., TrueCrypt master boot record (MBR) and Windows volume boot record (VBR; see Section III), additional effort is necessary to satisfy each component with the environment it expects. Furthermore, we primarily base Gracewipe on TrueCrypt because it is open sourced. Auditability is essential to security applications, and most other FDE solutions as we found are proprietary software/firmware and thus verifying their design and implementation becomes difficult for users. For this reason, we must be able to load Windows after exiting TXT (as TrueCrypt FDE is only available in Windows), which requires invocation of real-mode BIOS interrupts. It turned out to be a major challenge for Gracewipe. For the SED-based solution, we also choose to boot a Windows installation from the SED disk, as Windows is still more commonly used by everyday computer users. However, our Windows-based prototypes require disabling DMA, and thus suffer serious performance penalty (the system remains *usable* nonetheless, for non-disk-bound tasks); the root cause appears to be Intel TXT's incompatibility with real-mode (switching from protected to real-mode is required by Windows boot). In retrospect, booting a Linux-based OS after Gracewipe would have been easier to implement (as we could modify Linux as needed), but that would have less utility than our current Windows-targeted implementations.

Note that, in Gracewipe, the victim actively participates in destroying the hidden/confidential data, and thus may still be punished, e.g., put into jail for a significant period of time (e.g., [52]; see also cryptolaw.org for a survey on related laws in different jurisdictions). Gracewipe is expected to be used in situations where the exposure of hidden data is no way a preferable option. We assume a coercive adversary, who may release the victim when there is no chance of recovering the target data. Complexities of designing technical solutions for data hiding (including deniable encryption and verifiable destruction) are discussed in a blog post by Rescorla [40].

Authentication schemes under duress have been explored in recent proposals, e.g., [20, 6]. Such techniques may be integrated with Gracewipe, but they alone cannot achieve its goals, e.g., being able to delete keys under duress.

Contributions.

1) We propose Gracewipe, a secure data deletion mechanism to be used in coercive situations, when protecting the hidden/confidential data is of utmost importance. To the best of our knowledge, this is the first proposal to enable the following features together: triggering the hidden key deletion process in a way that is indistinguishable from unlocking the hidden data; verification of the deletion process; preventing offline guessing of passwords used for data confidentiality; restricting password guessing only to an unmodified Gracewipe environment; and tying password guessing with the risk of key deletion.
2) We implement Gracewipe with a PDE-mode TrueCrypt installation, and with an SED disk. Our implementation relies on secure storage as provided by TPM chips, and the trusted execution mode of modern Intel/AMD CPUs; such capabilities are widely available even in consumer-grade systems.
3) From our implementation experience with TrueCrypt and SED, apparently the design of Gracewipe is generic enough that it can be easily adapted for other existing software and hardware based FDE/PDE schemes.
4) Apart from secure deletion, our pre-OS trusted execution environment may enable other security-related checks, e.g., verifying OS integrity as in Windows secure boot, but through an auditable, open-source alternative.

[1] Recently, the anonymous TrueCrypt developers announced a sudden termination of their project. However, other groups have already been formed to continue future development, e.g., TCnext (http://truecrypt.ch) and CipherShed (https://ciphershed.org); and the TrueCrypt audit project (http://istruecryptauditedyet.com) is also being actively pursued. Furthermore, other open-sourced TrueCrypt replacements, e.g., VeraCrypt (https://veracrypt.codeplex.com) can also be adapted for Gracewipe.

[2] An attacker may optimize the guessing probability, which can be restricted by using multiple deletion passwords; see Clark and Hengartner [11].

II. Background, Goals, and Threat Model

A. Background

Gracewipe leverages several existing tools and mechanisms. Below, we provide a brief overview of them, to help understand Gracewipe's design and implementation.

Multiboot. The multiboot specification [18] is an open standard for multistage/coexistent booting of different operating systems or virtual machine monitors (VMMs); it has been implemented in several tools, e.g., GRUB,[3] kexec tools,[4] and tboot [24]. It enforces deterministic machine state and standardized parameter passing so that each stage (e.g., bootloader) knows what to expect from the previous stage and what to prepare for the next stage.

Chainloading. As Windows does not support the multiboot specification, it is *chainloaded*[5] by Gracewipe. Chainloading involves loading an OS/VMM as if it is being loaded at system boot-up (which may be actually from another running OS/VMM). The target image is loaded at a fixed memory address in real-mode (usually at 0x0000:0x7C00). The system jumps to the first instruction of the image without parsing its structure (except for the recognition of an MBR). At this time, machine state is like after a system reset, e.g., real-mode, initialized I/O, default global/interrupt descriptor table (GDT/IDT). We use GRUB as the bootloader for Gracewipe, as GRUB supports both multiboot and chainloading.

Tboot. Tboot [24] is an open-source project by Intel that uses the trusted execution technology (TXT) to perform a measured late-launch of an OS kernel (currently only Linux) or VMM (e.g., Xen). It can reload the platform dynamically (with the instruction GETSEC[SENTER]) and chain the measurement (through the TPM *extend* operation) of the whole software stack for attestation, including ACM (Intel's proprietary module), tboot itself, and any other binaries defined in the launch policy. The measurement outcome is checked against pre-established known values, and if different, the booting process may be aborted. Thereafter, the run-time environment is guaranteed to be isolated by TXT, with external DMA access restricted by VT-d (MMIO). Tboot can load (multiboot) ELF image and Linux bzImage. Note that it must be preceded by GRUB as tboot cannot be chainloaded.

TrueCrypt. The TrueCrypt on-the-fly full-disk encryption (FDE) utility is possibly the most popular choice in its kind. It supports plausibly deniable encryption (PDE) in the form of a hidden volume, which appears as free space of another volume. In the regular mode, an encrypted volume is explicitly mounted through TrueCrypt, on demand, after the OS is already booted up. We use its PDE-FDE mode (available only in Windows), where the OS volume is also encrypted and the original Windows MBR is replaced with the TrueCrypt MBR, which prompts for a password and loads the next 40–60 sectors (termed TrueCrypt modules) to decrypt the system volume.

Self-Encrypting Drives (SEDs). SEDs offer hardware-based FDE as opposed to software-only FDE solutions. A major benefit of an SED is its on-device encryption engine, which always keeps disk data encrypted. A media encryption key (MEK) is created at provisioning time and used to encrypt all data on the drive. MEK never leaves an SED (similar to the SRK of a TPM), and is only accessible to the on-device encryption engine (i.e., not exposed to RAM/CPU). An authentication key (AK) derived from a user-chosen password is used to encrypt the MEK. Several storage manufacturers now offer SED-capable disks. Trusted Computing Group (TCG) also has its open standard named Opal/Opal2 [60] for SEDs. SEDs provide various features such as instant secure erase and multiple user management.

With regard to the user interface for password entry, SEDs are usually shipped with an ATA security compliant interface as in regular drives. When a drive is powered up, it is by default in a *locked* state, until the user enters the correct password to switch it over to an *unlocked* state. The drive falls back to locked state at power loss. Unlocking involves using AK to decrypt MEK and, thus enabling decryption of disk data.

B. Goals and terminology

In this section, we specify Gracewipe's goals, and explain how they can be achieved. We also define the terminology as used in the remainder of the paper.

Goals. (1) When under duress, the user should be able to initiate key deletion in a way indistinguishable to the adversary. The adversary is aware of Gracewipe, and knows the possibility of key deletion, but is unable to prevent such deletion, if he wants to try retrieving the suspected hidden data. (2) In the case of emergency data deletion (e.g., noticing that the adversary is close-by), the user may also want to erase her data quickly. (3) In both cases, when the deletion finishes, the adversary must be convinced that the hidden data has become inaccessible and no data/key recovery is possible, even with (forced) user cooperation. (4) The adversary must be unable to retrieve TPM-stored volume encryption keys by password guessing, without risking key deletion; i.e., the adversary can attempt password guessing only through the Gracewipe interface. Direct offline attacks on volume keys must also be computationally infeasible.

Terminology and notation. We primarily target two types of storage encryption systems: software-based FDE with support for plausible deniability (termed as *PDE-FDE*) and hardware-based FDE. For a PDE-FDE system (e.g., TrueCrypt under Windows), a *decoy system* refers to the one appearing to be the protected system. The user should maintain certain frequency of using it for the purpose of deception. A *hidden system* is the actual protected system, the existence of which *may* be deniable and can only be accessed when the correct password is provided. The user should avoid leaking any trace of its use (as in TrueCrypt; cf. [14]). KN is the key needed to decrypt the decoy system, and PN is the password for retrieving KN. Similarly, KH is the key needed to decrypt the hidden system and PH is the password for retrieving KH. In addition, PD is the password to perform the secure deletion of KH; note that there might be multiple PDs (cf. panic password [11]), but in our current implementation, we only support one. KN and KH are stored/sealed in TPM NVRAM, which can be retrieved using the corresponding password, only within the verified Gracewipe environment. For a regular FDE system (e.g., SEDs), no decoy systems are needed, and thus no need to set up KN and PN. We use hidden/protected/confidential data interchangeably in this document.

[3] http://www.gnu.org/software/grub/

[4] https://www.kernel.org/pub/linux/utils/kernel/kexec/

[5] https://www.gnu.org/software/grub/manual/html_node/Chain_002dloading.html

Overview of how Gracewipe goals are achieved. For goal (1), we introduce PD that retrieves KN but at the same time deletes KH from TPM. Thus, if either the user/adversary enters a PD, the hidden data will become inaccessible and unrecoverable (due to the deletion of KH). PN, PH and PDs should be indistinguishable, e.g., in terms of password composition. In a usual situation, the user can use either PH or PN to boot the corresponding system. If the user is under duress and forced to enter PH, she may input a PD instead, and Gracewipe will immediately delete KH (so that next time PH only outputs a null string). Under duress, she can reveal PN/PDs, but must refrain from exposing PH. The use of any PD at any time (emergency or otherwise), will delete KH the same way, and thus goal (2) can be achieved.

Goal (3) can be achieved by a chained trust model and deterministic output of Gracewipe. The trusted environment is established by running the deletion operation via DRTM, e.g., using Intel TXT through tboot [24]. We assume that Gracewipe's functionality is publicly known and its measurement (in the form of values in TPM PCRs) is available for the target environment, so that the adversary can match the content in PCRs with the known values, e.g., through a TPM quote operation. Gracewipe prints a hexadecimal representation of the quote value, and also stores it in TPM NVRAM for further verification. A confirmation message is also displayed after the deletion (e.g., "A deletion password has been entered and the hidden system is now permanently inaccessible!").

For goal (4), we use TPM's sealing feature, to force the adversary to use a genuine version of Gracewipe for password guessing. Sealing also stops the adversary from modifying Gracewipe in such a way that it does not trigger key deletion, even when a PD is used. We use long random keys (e.g., 128/256-bit AES keys) for actual data encryption to thwart offline attacks directly on the keys. A side-effect of goal (4) is that, if a Gracewipe-enabled device (e.g., a laptop) with sensitive data is lost or stolen, the attacker is still restricted to password guessing with the risk of key deletion.

C. Threat model and assumptions

Here we specify assumptions for Gracewipe, and list several unaddressed attacks.

1) We assume the adversary to be hostile and coercive, but rational otherwise (cf. [40]). He is diligent enough to verify the TPM quote when key deletion occurs, and then (*optimistically*) stop punishing the victim for password extraction, as the hidden password is of no use at this point. If the victim suspects severe retaliation from the adversary, she may choose to use the deletion password only if the protected data is extremely valuable, i.e., she is willing to accept the consequences of provable deletion.
2) The adversary knows well (or otherwise can easily find out) that TrueCrypt/SED disk is used, and probably there exists a hidden volume on the system. He is also aware of Gracewipe, and its use of different passwords for accessing decoy/hidden systems and key deletion. However, he cannot distinguish PDs from other passwords on a list that the victim is coerced to provide.
3) The adversary can have physical control of the machine and can clone the hard drive before trying any password. However, we assume that the adversary does not get the physical machine when the user is using the hidden system (i.e., KH is in RAM). Otherwise, he can use cold-boot attacks [22] to retrieve KH; such attacks are excluded in our threat model, but see also TRESOR [34].
4) The adversary may reset the TPM *owner* password with the *takeownership* command, or learn the original owner password from the victim; note that NVRAM indices (where we seal the keys) encrypted with separate passwords are not affected by resetting ownership, or the exposure of the owner password. With the owner password, the adversary can forge TXT launch policies and allow executing a modified Gracewipe instance. Any such attempts will fail to unlock the hidden key (KH), as KH is sealed with the genuine copy of Gracewipe. However, with the modifications, the attacker may try to convince the user to enter valid passwords (PH, PN or PD), which are then exposed to the attacker. We expect the victim not to reveal PH, whenever the machine is suspected to have been tampered with. We do not address the so called evil-maid attacks [43, 30], but Gracewipe can be extended with existing solutions against such attacks (e.g., [36]).
5) We exclude inadvertent leakage of secrets/passwords from human memory via side-channel attacks, e.g., the EEG-based *subliminal probing* [16]; see Bonaci et al. [7] for counter-measures. We also exclude *truth-serum* [62] induced attacks; effectiveness of such techniques is also strongly doubted (see, e.g., [45]).
6) Gracewipe facilitates secure key deletion, but relies on FDE-based schemes for data confidentiality. For our prototypes, we assume TrueCrypt and SED adequately protect user data and are free of backdoors. The SED-based Gracewipe relies on a proper implementation of cryto-primitives and FDE on SED devices, as Gracewipe only unlocks/erases the ATA security password from TPM NVRAM. For the SED version, the user must choose a real SED disk instead of a regular drive, even though Gracewipe only uses the ATA security protocol supported by most current hard drives. The SED disk must ensure that MEK is encrypted properly with the ATA security password and the user data is properly encrypted with MEK (but see Müller et al. [35]). In contrast to TrueCrypt's open source nature, unknown design/implementation vulnerabilities in a specific SED may invalidate Gracewipe guarantees.
7) We assume the size of *confidential/hidden data* is significant, i.e., not memorizable by the user, e.g., a list of all US citizens with top-secret clearances (reportedly, more than a million citizens[6]). After key deletion, the victim may be forced to reveal the nature of the hidden data, but she cannot disclose much.
8) We assume Intel TXT is trustworthy and cannot be compromised and thus ensures the calculated measurements can be trusted (hence only genuine Gracewipe unseals the keys); past attacks [64, 65] on TXT include exploiting the CPU's SMM (System Management Mode) to intercept TXT execution. Protections against such attacks include: Intel SMI transfer monitor (STM), and the newly pro-

[6] http://www.usatoday.com/story/news/2013/06/09/government-security-clearance/2406243/

posed (still unavailable) Intel software guard extensions (SGX). Additionally, we assume that hardware-based debuggers cannot compromise Intel TXT. We could not locate any documentation from Intel in this regard.[7] As documented [4], AMD's SVM disables hardware debug features of a CPU.

III. Gracewipe Design

In this section, we expand the basic design as outlined in Section I. We primarily discuss Gracewipe for an FDE solution with deniable hidden volume support (i.e., PDE-FDE), and we use TrueCrypt as a concrete example. We provide implementation details in Sections IV, V. The FDE-only version is simpler than the PDE-FDE design, e.g., no decoy volume and no chainloading are needed; for details of the FDE-only version, see Section VI. These two versions mostly use the same design components, differing mainly in the key unlocked by Gracewipe and the destination system that receives the key.

Overview and disk layout. Gracewipe inter-connects several components, including: BIOS, GRUB, tboot, TPM, *wiper* (provides Gracewipe's core functionality—see below under "Wiper"), TrueCrypt MBR (or SED/ATA interface), and Windows bootloader. See Fig. 1 for an overview of Gracewipe components, disk layout when TrueCrypt is used, and execution control flow. The hidden data is stored encrypted on a hard drive, as in a typical TrueCrypt hidden volume. We assume two physical volumes: one hosting the decoy system (regular TrueCrypt encrypted volume), and the other volume containing the hidden system (hidden TrueCrypt volume). KN and KH are technically TrueCrypt volume passwords for the two volumes respectively, but we generate them from a random source. Both are stored in TPM NVRAM, and are not typed/memorized explicitly by the user. In the deployment phase, they are generated in a secure way with good entropy and configured as TrueCrypt passwords. Each valid password (including any PD) will decrypt a corresponding key in TPM NVRAM for a specific purpose.

Wiper. The core part of Gracewipe's functionality includes bridging its components, unlocking appropriate TPM-stored keys, and deletion of the hidden volume key. We term this part as the *wiper*, which is implemented as a module securely loaded with tboot. It prompts for user password, and its behavior is determined by the entered password (or more precisely, by the data retrieved from TPM with that password). Namely, if the retrieved data contains only a regular key (KH/KN), the wiper passes it on to TrueCrypt, or if it appears otherwise (as designated by a deletion indicator) to have a control block for deletion, the wiper performs the deletion and passes the decoy key KN to TrueCrypt. We modified TrueCrypt to directly accept input from the wiper (i.e., the original TrueCrypt password prompt is bypassed), and boot one of the encrypted systems.

As the wiper must operate at an early stage of system boot and still provide support for relatively complex functionality, it must meet several design considerations, including:

1) It must be bootable by tboot, as we need tboot for the measured launch of the wiper. This can be achieved by conforming to required file formats (e.g., ELF) and header structures (e.g., multiboot version number).
2) It must load the TrueCrypt loader for usual operations, e.g., decrypt the correct volume and load Windows. This is mainly about parameter passing (e.g., TrueCrypt assumes register DL to contain the drive number).
3) It must access the TPM chip and perform several TPM operations including sealing/unsealing, quote generation, and NVRAM read/write. Note that at this point, there is no OS or trusted computing software stack (such as TrouSerS [2]) to facilitate TPM operations.
4) It must provide an expected machine state for the component that will be loaded after the wiper (e.g., Windows). Both TrueCrypt and Windows assume a clean boot from BIOS; however, Windows supports only strict chainloading, failure of which causes several troubles including system crash (see Section V).

Execution steps. Gracewipe's execution flow is outlined in Fig. 1. It involves the following steps: (1) The system BIOS loads GRUB, which then loads tboot binary as the kernel, together with other modules including the wiper, ACM SINIT module and the policy list (see Section IV-C). (2) Tboot checks for required support on the platform; if succeeded, tboot starts the MLE by calling GETSEC[SENTER]. (3) All measurements are calculated and matched with the values stored in TPM. If the matching is successful, the wiper is loaded in the same context as tboot; otherwise, execution is halted. (4) The wiper prompts the user for password, and uses the entered password to decrypt locations where we store KH/KN one by one. If none is decrypted, it halts the system; otherwise, the wiper copies the decrypted key (i.e., TrueCrypt password) to a memory location to be retrieved later by TrueCrypt. (5) If one of the PDs is entered (indicated by the decrypted data), the wiper immediately erases KH from TPM, and performs a quote to display the attestation string on the screen. It either halts the system or continues loading the decoy system according to user choice. (6) The wiper switches the system back to real-mode, reinitializes it by mimicking what is done by BIOS at boot time, and replaces the handler of INT 13h. (7) TrueCrypt MBR is executed, which decompresses the subsequent sectors from the hard drive into system memory. TrueCrypt also inserts its filter to the handlers of INT 13h and 15h. The corresponding volume is decrypted on-the-fly, if the TrueCrypt password (as received from the wiper) is correct. Then the boot record on the decrypted partition is chainloaded, and Windows is booted.

Storing Gracewipe components. For booting the target system, Gracewipe's software components (GRUB, tboot, wiper, TrueCrypt MBR) can reside on any media, including any secondary storage. In our proof-of-concept system, we keep these components on a secondary USB storage. The target hard drive only contains TrueCrypt modules (except its MBR) and all encrypted partitions. All Gracewipe components can also be placed on the target hard drive alone, with additional effort, including: (a) an extra partition with file system is needed to store tboot and its modules; (b) GRUB MBR will overwrite part of TrueCrypt, and thus either of them must be relocated; and (c) TrueCrypt MBR must be modified not to read the TrueCrypt modules from their predefined locations (i.e., sector 2 of the disk).

[7] See a related tboot discussion thread at (Aug. 2012): http://sourceforge.net/p/tboot/mailman/message/29747527/

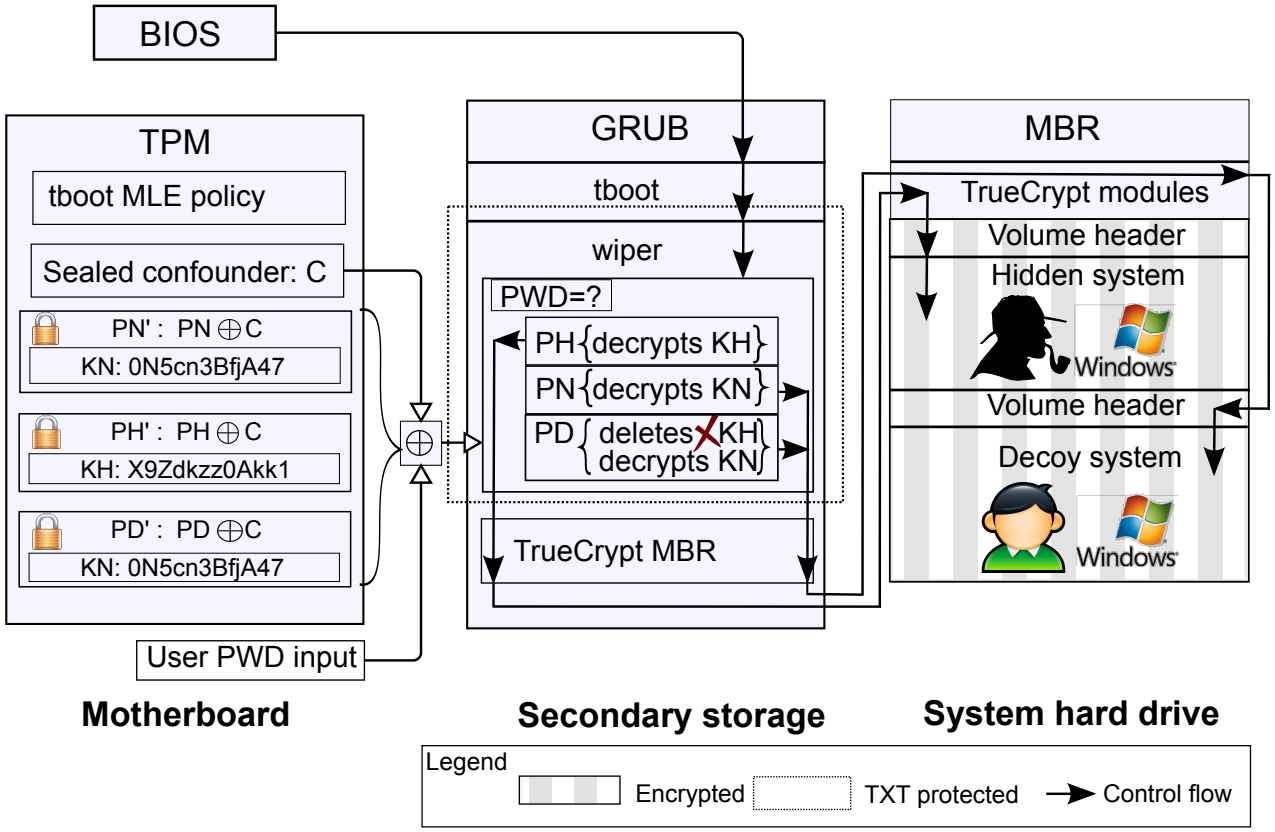

Fig. 1. Gracewipe with TrueCrypt (PDE-FDE version). PN = password for the decoy system; PH = password for the hidden system; PD = deletion password. PH unlocks KH, the TrueCrypt key for the hidden system; both PN and PD unlock KN, the TrueCrypt key for the decoy system. Additionally, PD also deletes KH from TPM NVRAM. Components included in measurement: tboot, wiper and TrueCrypt MBR. Note that passwords are hashed before XOR'd (see under "Sealing in NVRAM" in Section III.

Sealing in NVRAM. TPM specifications mandate mechanisms against guessing attacks on password-protected NVRAM data (e.g., only a few passwords may be tested in a row). However, such mechanisms are inadequate for Gracewipe as the adversary has physical control and can patiently keep testing passwords, and user-chosen passwords tend to be relatively weak. The implementation of such mechanisms is also vendor-specific (see Section VII). If the adversary would like to brute-force a specific index a few times until the chip is locked out and reset it with TPM_ResetLockValue, he may eventually succeed by automating the process.

To address this, we apply TPM's data sealing technique, so that if an altered software stack (i.e., anything other than Gracewipe) is run, the desired data will not be unsealed, and thus will remain inaccessible. Note that sealing does not disallow guessing from within the Gracewipe environment; however, as Gracewipe is active, each guess may unlock the hidden/decoy data, or trigger key deletion.

A 128-bit confounder C is generated at Gracewipe's initialization and sealed in an NVRAM index using PCR 17 (tboot's measurement) and PCR 18 (wiper and TrueCrypt MBR). Sealing ensures that C is never exposed outside the designated environment. Then each user-chosen password is hashed with SHA1 to 160 bits and truncated to 128 bits. The output is XOR'd with C and used as authdata secret to protect an NVRAM index storing the corresponding encryption key along with a deletion indicator; e.g., the authdata secret derived from PN is: $PN' = \text{SHA1}(PN) \oplus C$, where SHA1 output is truncated to the first 128 bits (PH' and PD' are derived in a similar manner). PN' is then used to protect KN concatenated with the deletion indicator, which is set only for the value protected by PD'. Now, without brute-forcing C, an attacker cannot launch a guessing attack on passwords; as C is a random 128-bit value, brute-forcing would be infeasible.

Note that, if we use passwords directly as authdata secrets (e.g., PN instead of PN'), even with sealing, passwords may still be guessed. Consider the following construction: use PN as the authdata secret to protect $KN \oplus C$ concatenated with the deletion indicator. Without the sealed secret C, of course KN cannot be retrieved. However, by checking the TPM's response (success/failure) to a guessed authdata secret value, the attacker can learn PN and other valid passwords, without going through Gracewipe. To avoid this attack, we use PN' as authdata secret and use KN as the protected data.

IV. Implementation with TrueCrypt

In this section, we discuss implementation details of Gracewipe with TrueCrypt under Windows. We also implemented Gracewipe with SED disks; see Section VI. Side effects as observed from our implementation choices are discussed in Sections V and VII. Gracewipe involves primarily the wiper and some minor changes in TrueCrypt. Approximately, the wiper has 400 lines of code in assembly, 700 lines in C and 1300 lines of reused code from tboot.

Machine configuration. For our prototype system, we used a primary test machine with an Intel Core i7-3770S processor (3.10 GHz) and Intel DQ77MK motherboard, 8GB RAM with 1TB Western Digital hard drive. We also used another machine of similar configuration for verification of the TXT issue when switched back to real mode (Section V). In addition, a few non-TXT-enabled machine were also used for miscellaneous purposes (e.g., TPM deadlock problem and TrueCrypt testing).

A. TPM I/O Access

As discussed in Section III, the wiper must access the TPM at an early stage of system boot, i.e., right after GRUB and tboot. Most applications access TPM through the TCG software stack [2], by calling a set of APIs provided by a library with a device driver, as part of the OS. However, at boot time, no such support is available (as in our case); thus, we must handle the communications between the TPM and wiper, and implement a subset of (complex) functionalities of the TCG software stack.

During our pre-boot wiper environment, there are three options for interfacing with a TPM. (1) Port-mapped I/O (PMIO): It is the legacy way to connect various peripherals with the processor, and is available in both real and protected modes. However, it is not recommended for TPM communications as the port assignment can be vendor-specific. (2) TCG BIOS: TPM functionalities may be accessed using a TCG-compliant BIOS [59], via INT 1Ah, and thus it works only in real mode. We do not use this method as Intel tboot/TXT requires protected-mode (more detailed in Section V). (3) Memory-mapped I/O (MMIO): In the TPM 1.2 specification, interfacing with TPM of different vendors has been standardized to use MMIO (legacy I/O is supported for backward compatibility). MMIO, as opposed to port-mapped I/O, uses the same address bus to address both memory and I/O devices. The devices are mapped to a range of memory and read/written as regular memory location (whereas PMIO requires special instructions such as IN and OUT). TPM by default is mapped to a region starting at 0xFED40000. This only works in protected mode, since it exceeds the boundary of memory access in real mode. As the wiper executes right after tboot releases control, where a full MMIO-based access to TPM is already initialized (in protected mode), we can reuse part of the TPM functions in tboot (e.g., status check).

B. Secure Storage in TPM

In addition to I/O interfacing, construction of packets (termed as message blobs) to communicate with TPM is another consideration as we are implementing our own protocol stack. Since tboot takes care of the measured launch, what we need in the wiper is mainly how to securely store/access the keys in TPM. Below, we briefly explain how to use TPM's secure storage and the way we store/access Gracewipe keys in TPM. In a few cases, we reverse-engineer TPM message blobs, due to the lack of adequate documentation.

TPM NVRAM memory is represented by an index number plus an offset. Indices must be allocated with certain protection, either through *localities* or with explicit authentication. Locality [55] defines a specific context in which an operation takes place. TPM NVRAM operations are locality-sensitive. For instance, an index defined as read locality 0 and write locality 4 only permits data to be read out at 0 and written in at 4. Explicit authentication requires a secret to protect the index. It can either be the TPM owner password or a separately defined secret; here, we term the former case as *owner access* and the latter as *authdata access*.

There are three authorization levels defined to communicate with TPM: non-authorized, single authorized, and dual authorized command messages. Depending on the context and property of the command, one or multiple authorization levels may be required. There are three authorization tags, indicating which level the caller wants to establish: TCG_TAG_RQU_COMMAND, TCG_TAG_RQU_AUTH1_COMMAND and TCG_TAG_RQU_AUTH2_COMMAND. They are named literally according to the number of authorization blocks in the message; we refer them by *non-auth tag*, *auth1 tag* and *auth2 tag*, respectively hereafter.

Combining the aspects above, there are three options for storing the keys. (1) Locality-protected indices: As locality provides only weak protection bound to the context, keys must be encrypted before being stored in TPM, and thus requires adding another layer of complexity. (2) Indices with owner access: TPM owner password is shared for various functions and cannot be dedicated to specific indices. More importantly, as stated in Section II-C, the owner password can be reset easily, as the adversary has physical access. Therefore, this method is not secure at all for our key storage. (3) Indices with authdata access: A password as authdata can be set to protect each index; this is the option we choose to use.

However, authdata access is less straightforward, as constructing authenticated message blobs for TPM is complicated. The TPM specification [56, 57, 58] provides enough documentation for operation procedures and data structures, but sometimes unexplained contextual information is needed for proper implementation of auth access commands (e.g., the usage of different types of authdata is not clearly stated, and often all zeros are used). To the best of our knowledge, in pre-OS projects such as TrustedGRUB and tboot, no realization for authdata accessed NVRAM interaction is available. TrustGRUB uses TCG BIOS calls in real-mode (it just needs PCR operations); tboot implements only TPM_NV_ReadValue()/TPM_NV_WriteValue() (with non-auth tag). We also explored the source code of the TCG software stack (TrouSerS) in Linux, which is written as a full-fledged implementation with several layers of abstraction and context-dependent branching. At the end, it was easier to reverse-engineer the packets (blobs) that we could observe through the debug mode of TrouSerS. We then checked the traffic to find inconsistencies that are not stated in the specification. Below, we discuss a particular undocumented example case:

How to execute the object specific authorization protocol (OSAP) to generate a shared secret, and use it to generate *encAuth* is mostly undocumented (calculation of encAuth is

specified as depending "on the context" [57]). It is vaguely mentioned in an example workflow of the specification. For instance, in the case of TPM_NV_DefineSpace command, we found that the NVRAM authdata (hashed) is used for encAuth, if the parameter *size* is non-zero; otherwise, a 20-byte value of all zeros is used (which implies releasing space). In both cases, the hashed owner auth is needed.

At the end, we implemented the following TPM functions in C: *tpm_nv_read_value_auth()*, *tpm_nv_write_value_auth()*, *tpm_nv_define_release()*, *tpm_loadkey2()*, *tpm_quote2()*; we reused most other functions from tboot.

C. Measured Launch with Tboot

The root of trust measurement in Gracewipe is based on and implemented with tboot. Here, we briefly describe important links in our chain of trust.

The correct measurements of Gracewipe binaries are stored in TPM NVRAM as policies. The policies are written to indices with owner write access and enforced at boot time. They must be generated in the configuration phase, when it is assumed that the environment is malware-free. In tboot terminology, there are two types of policies: MLE policy (Intel TXT policy) and custom policy (tboot verified launch policy). The MLE policy can take into account the software stack required to prepare for TXT execution (see Section IV-D). When all the modules in the MLE policy have been verified in TXT, tboot is re-entered and ready to perform its own measurement (custom policy). The tboot binary is included in the MLE policy and our wiper with embedded TrueCrypt MBR is defined in the custom policy.

Tboot must be loaded with GRUB or a similar multiboot-compliant loader. We add entries for Gracewipe binaries in GRUB's *menu.lst* file (stored on a USB disk, simulated as a floppy (fd0)). In tboot's command line parameters, we add *ap_wake_mwait=true* to deal with some TXT shutdown issues (see Section V-D).

D. Attestation with TPM Quote

For verifying the deletion of KH, the measurement of the executed code must be conveyed in a trusted way, e.g., via a TPM quote. A quote operation involves generating a signature on a requested set of PCRs, and a verifier-provided nonce with TPM's attestation identity key (AIK). As we assume that both the identity of a TPM and the measurements of Gracewipe software stack are verifiable with public information, the integrity of the deletion process can be guaranteed by verifying the quote result. Below, we discuss issues related to quote generation and verification in Gracewipe.

Human-initiated attestation. During the quote generation, the adversary's participation in selecting a nonce is required to prevent replay attacks. We consider two sources for the nonce: (1) A timestamp (hashed for the same output length) used in quoting may convince the verifier that the quote is fresh. Gracewipe displays the timestamp used, and the verifier can write down the short timestamp for later verification, and match it with a separate clock. (2) We also implement a more direct method to prompt the verifier for an arbitrary string and use the hashed string as the nonce; this requires active involvement of the verifier.

Generating the quote. As an advantage of the "late launch" with Intel TXT, we only need to take into account everything after the trusted execution starts, i.e., the authenticated code module (SINIT ACM) together with some static fields (PCR 17), and the measured launch environment (MLE, PCR 18). Therefore, the known good measurements are only determined by a limited number of Intel published ACMs (processor-specific), tboot and Gracewipe.

Note that we do not have to use either auth1 or auth2 tags, as no authentication is needed to generate a quote; so we just used the non-auth tag. However, loading the signing key requires an object-independent authorization protocol (OIAP) session with the storage root key (SRK) password. In the end, we write the 256-byte quote value into an NVRAM index and also display it on the screen.

Transferring the quote. With Gracewipe, the adversary must "offload" the quote from the victim's computer as he cannot perform an in-place verification. A straightforward way is to display the quote on the screen for the adversary. We considered using QR code so that he can scan it with a smartphone-like device. But we refrained from doing so because it requires Gracewipe to enter in graphics mode. Instead, we store the quote into a predefined NVRAM index (with no password protection). The adversary can access it at a later time from the victim's computer. He can boot into an OS to perform the verification on his own. We also display the quote on the screen for a diligent adversary for matching purposes.

E. Changes in TrueCrypt

To make TrueCrypt aware of Gracewipe, we make some changes in TrueCrypt. We keep such changes to a minimum for easier maintenance and deployment. Our changes are mostly in BootLoader.com.gz (BootMain.cpp, 9 lines added in assembly, 6 lines added in C), and a few minor changes in BootSector.asm. In BootSector.bin, the integrity of the rest of TrueCrypt modules (including the decompressor) is first verified. It calculates only the checksum (CRC32) due to the small footprint of MBR. The checksum is generated at install time and when we make any changes, we have to update the corresponding checksum in the configuration area (part of the 512-byte MBR). During development, we flipped the corresponding bit to disable it.

In BootLoader.com.gz (TrueCrypt modules), the modifications are mainly for receiving decrypted passwords (treated as keys in Gracewipe) from the wiper without user intervention. As an interchange between the wiper and TrueCrypt, we use a hard-coded memory location (0x0000:0x7E00, an address following the wiper, and not foreseeably used).

F. Wiper Initialization

Before Gracewipe can function properly, some initial setup must be performed in addition to initializing the platform (e.g., taking TPM ownership and installing TrueCrypt). Such initial setup involves both the host OS and Gracewipe.

Preparation in the host OS. A script that works with the TrueCrypt installer must automatically generate a strong key (i.e., random and of sufficient length) to replace the user-chosen password. This is done for both KN and KH. Then the user must copy (manually or with the help of the script) KN and KH to be used with Gracewipe. She must destroy her copies of the two keys after the setup phase.

Preparation in Gracewipe. Gracewipe comes with a single consolidated binary with two modes of operation: deployment and normal. Modes are determined by the value (zero/non-zero) in an unprotected NVRAM index; note that, reinitializing Gracewipe has no security impact (beyond DoS), but still a simple password can be set to avoid inadvertent reset. If the value is non-zero, normal mode is entered; otherwise, Gracewipe warns the user and enters the deployment mode.

In the deployment mode, the wiper first clears out the three NVRAM indices corresponding to PN, PH, and PD. Then it uses the random number generator in TPM to generate a 128-bit confounder C and seals it with the current environment measurements into an NVRAM index. The user is then prompted for the three passwords (PN, PH and PD) of her choice and the two keys (KN and KH) generated on the OS. The wiper XORs C with the hashed passwords, and uses the output to secure the three indices for storing KN and KH (see Section III). In the end, the wiper toggles the mode value for the next time to run in the normal mode.

V. Loading Windows after Tboot

In this section, we discuss some side effects caused by loading Windows after running tboot and possible solutions. Parts of them have been verified and can be applied to other similar systems with certain adaptation, while some may be too specific to our test machines.

A. Observations

Loading Windows after TXT (as opposed to Linux/Xen) causes issues in two aspects:

- At boot time, the system must be switched back to real mode (from tboot's protected mode), as Windows assumes that it starts at system reset (chainloading). Specifically, Windows invokes BIOS interrupts for the most part of its initialization. In contrast, Linux has both real/protected-mode entries, and is multiboot compliant.
- As Windows is closed-sourced, it cannot be adapted to be TXT-aware, as opposed to Linux (e.g., via a simple flag, CONFIG_INTEL_TXT).

As of writing, we are unaware of any projects or products that involve entering TXT and thereafter switching back to real-mode, not to mention invoking BIOS interrupts therein. According to Intel's documentation [25], we can infer that TXT is not intended for use in real-mode or preserving the functionality for real-mode. Our observation is when the MLE is launched successfully, and by the time our wiper chainloads the TrueCrypt boot record, no matter whether the system is still in TXT or has exited TXT, some I/O related BIOS calls do not function properly. The root cause appears to be DMA-related as observed from our two test machines (see Section V-E); however, we could not confirm this from TXT documentation.

One of the symptoms of BIOS issues is that a disk read operation with INT 13h returns an error code of 80h (which means timeout), as tested with BIOS version 56P or below (Intel DQ77MK motherboard). A system halt may also occur with BIOS version 60. The problem appears to be non-deterministic. As the rest of the binaries we need to load all rely on calling INT 13h (TrueCrypt MBR, TrueCrypt modules, BootMgr, and winload.exe), we could not proceed without restoring such BIOS functionality.

Also, tboot is not designed for chainloading, and as we observed, the presence of tboot as part of the loading chain makes the environment incompatible with Windows. In the rest of this section, we discuss our fixes to the above mentioned Windows-specific problems.

B. Rewriting BIOS Interrupt Handler for Disk Access

We reimplement parts of the BIOS interrupt handler that must be available until the point when Windows is fully initialized, more precisely, until the TrueCrypt device driver takes control.

For simplicity, we use the programmed input/output (PIO) mode of the commonly accepted ATA specifications. Without BIOS support, disks can still be accessed through writing to and reading from a group of special I/O ports in a designated sequence. The completion of an operation can be signalled by triggering an external interrupt, or by polling the status register. We avoid involving another layer of interrupt, and use the polling mechanism instead.

There are two access modes that should be supported in the INT 13h handler. CHS (Cylinder, Head and Sector [66]) is a legacy method of accessing hard disks. The caller must be aware of the geometry of the disk being accessed. Most boot-time programs (e.g., Windows VBR and GRUB stage1) use this mode for simplicity and backward compatibility. By default, INT 13h uses CHS (function codes starting with zero, e.g., 0x02 and 0x03). LBA (logical block addressing [66]) is a more recent, widely-used linear addressing scheme; the location of blocks of data is represented by a single integer regardless of the actual geometry of the disk. In reality, after the initial stage, most bootloaders switch to LBA, if it is supported by the device (which is common nowadays). To also simulate LBA, we need to add support for INT 13h Extensions (function codes starting with 4, e.g., 0x42 and 0x43).

We provide the required INT 13h functionality by translating the BIOS-formatted read/write and other requests into the representation of ATA PIO specifications in assembly. This covers eight functions of INT 13h with an approximate code size of 350 lines in assembly; the functions are: read drive parameters (0x08), read/write sectors from/to drive (0x02/0x03), check extensions present (0x41), extended read/write sectors from/to drive (0x42/0x43), extended read drive parameters (0x48) and read disk/DASD type (0x15). We observed that function code 0x15 is used by winload.exe as a shadowed BIOS call, which is surprising as BIOS interrupt handlers are invalidated in protected mode and Windows kernel should not use them (see more in Appendix A).

C. Memory Overlaps

A problem we frequently encountered is system crash, caused by memory access violation due to memory overlaps between tboot and Windows. Below, we discuss such access violations and how we fixed them.

Windows is loaded by TrueCrypt modules at memory address 0x0000:0x7C00, when the desired volume has been decrypted. However, Windows is totally unaware of tboot, and the memory layout used by Windows is also unavailable to us. We observed that when tboot is removed from Gracewipe, Windows can start up successfully; however, if tboot is included for a measured launch, Windows crashes while loading device

drivers in winload.exe. By checking the debug information (in safe mode), we could not identify a specific offending module. Then we employed a manual technique: from Windows boot manager we chainloaded GRUB4DOS [1], where we gradually shrunk a zeroized region using mem_set() and identified an overlapped memory region between tboot and Windows.

By default, tboot sets its starting address at 0x00800000, and reserves its space in the E820 table. However, Windows seems to use a region that overlaps part of the tboot binary, as observed via Windows boot debugger (a mode of windbg [33] for debugging Windows BootMgr): `Primary image base = 0x0086b000 Loaded module list = 0x00905b40`. This overlaps the range of 0x00800000–0x00AC7000, the first section reserved by tboot and causes access violations, leading to system crash.

Workaround. We first tried with the Windows boot configuration data (BCD [41]) file, which provides two sets of parameters that affect Windows memory allocation. (1) *AvoidLowMemory* restricts the use of memory below the specified value by the bootloader, but we could not verify the effectiveness of this parameter. (2) *BadMemoryList* marks a list of memory page frames (4K) as bad, and setting *BadMemoryAccess* to NO prevents access to bad pages. BadMemoryList entries are enforced by the Windows memory manager that resides in Ntoskrnl.exe. However, Ntoskrnl.exe is loaded midway in winload.exe [42] and thus the enforcement comes too late (as before Ntoskrnl.exe, other modules are already loaded). In fact, the region where tboot resides is accessed before the memory manager takes control.

We then tested an adhoc method with success. We assume that Windows does not use the range starting at 0x08000000 (i.e., 0x00800000 multiplied by 16), at least before it is fully initialized. In tboot's config.h file, we changed TBOOT_BASE_ADDR from 0x00800000 to 0x08000000 and TBOOT_START from 0x00804000 to 0x08004000. This shifts the whole tboot binary up to a much higher location in memory, and thus avoids the overlap.

D. Shutting down TXT

For secure deletion, we require TXT protection only for the Gracewipe environment. As Windows is TXT unaware, we must tear down the TXT session when leaving the wiper; this can be easily done by calling GETSEC[SEXIT]. However, simply executing this instruction before loading TrueCrypt MBR crashed the system. We briefly explain the reason and a simple solution to it.

In a multi-core environment, application processors (AP) are waken up on demand by the OS, when the bootstrap processor (BSP) finishes initialization. In our case, we simply bootstrap the system and run the wiper on the BSP, and without making use of (waking up) any APs by scheduling an OS/VMM, we switch the system back to its initial state. Thus, when trying to shutdown TXT at this point, while we have 7 APs in the Wait-for-SIPI (WFS) state, SEXIT causes a system reset; this is not so unexpected as mentioned in the tboot documentation [29].

To avoid dealing with the state of WFS, we make use of the MWAIT feature available in current CPU models [26]. GETSEC[SEXIT] can work if the APs are still in MWAIT. Thus TXT is safely (from the CPU's perspective) torn down, and the wiper can continue to run. To enable this feature, we only need to append a switch to tboot parameters: `ap_wake_mwait=true`.

E. Additional disk access issues

As our custom INT 13h handler only supports parallel ATA (PATA/IDE), and our machine comes with only SATA controllers (as most off-the-shelf PCs), we manually change the mode of SATA controllers from AHCI to IDE in the BIOS setup. If Windows is installed with AHCI, modifying the registry to use pciide.sys instead of msahci.sys is needed.

Even with all the above modifications, our Windows booting still failed with UNMOUNTABLE_BOOT_VOLUME (0x000000ED). The reason code of 0xC000014F indicates a disk hardware problem, which cannot be true as we could boot Windows without the tboot MLE. We found that the logic of the call stack for INT 13h was apparently correct, but the returned data contained seemingly random bytes (see Appendix A for debugging details). We suspected the data transfer mechanism in Windows disk driver as a possible reason. By checking the default mode of the corresponding ATA channel, we found it was in "Ultra DMA Mode 5". We changed the ATA channel to use the "PIO" mode and restarted the system. We could see that the driver returned the correct data, and Windows booted successfully. For the same reason, we suspect that the original INT 13h handler also relies on DMA to communicate with hard drives (but could not confirm from the manufacturer's documentation).

VI. Implementation with SED

Our Gracewipe prototype for TrueCrypt with hidden volume support is an example of Gracewipe's applicability for a PDE-FDE system. To show that Gracewipe's design is easily adapted to support hardware-based FDE, we implement Gracewipe for a Self-Encrypting Drive (SED). This implementation shares several parts with the TrueCrypt prototype. In this section, we discuss the parts where our SED implementation differs from the basic architecture (Section III) and the TrueCrypt implementation (Sections IV and V).

A. Gracewipe on SED

Here we replace TrueCrypt passwords with an ATA security user password, which is actually a high entropy key (referred as ATA key). Note that there is no hidden volume with the SED-based Gracewipe. In the deployment mode, the user is prompted for the ATA key (as KH) and the user-chosen passwords (PH for KH and PD for deletion). Both PH and PD are hashed and XOR'd with a sealed confounder (C) and used to protect the ATA key in TPM NVRAM. In the normal mode, the correct password PH decrypts KH from TPM, and KH unlocks the drive by decrypting MEK. If one of the PDs is entered, Gracewipe erases KH and goes through the same quoting process as explained in Section IV-D.

We did not use vendor-specific APIs, e.g., Seagate's DriveTrust, to operate on MEK to achieve cryptographic deletion. Since MEK is encrypted with a key derived from the user password (KH), erasure of KH disables access to MEK, and hence makes all the data protected by MEK inaccessible. Note that the ATA Security API as we use for the password interaction, is available on most password-protected drives.

However, as user data is unencrypted on regular password-protected drives, as emphasized in Section II-C, we mandate only SEDs to ensure that when the ATA key is erased, the user data remains encrypted with no key to decrypt.

B. Implementation

Recall from Section V-B that when rewriting the INT 13h handler, we communicated with ATA compliant drives through I/O ports by sending commands and receiving responses (PIO read_ext/write_ext 0x24/0x34). For unlocking ATA security, we choose the same approach (with command 0xF2). In the wiper, before switching back to real-mode, when the correct ATA key is decrypted from TPM, we provide it to the hard drive as if it is entered by the user. This is equivalent to typing the password when prompted by BIOS at boot time.

The deletion and quoting occur before any interaction with the ATA interface, and are therefore independent of the newly added functions. This is how a similar framework can be kept for SED-based implementation as with the TrueCrypt-based one. Since unlocking occurs in protected-mode, there is no data to be passed to the chainloading part of the wiper. Instead, we retrieve the first sector of the unlocked disk (in place of the TrueCrypt MBR) to simulate the behavior of a regular BIOS that performs chainloading. The same context for loading Windows is also created (e.g., rewritten INT 13h handlers and disabled DMA). We were able to boot up Windows, and the SED-based implementation required adding only about 80 lines of code to the TrueCrypt prototype.

VII. Limitations

Below, we summarize limitations originating from our somewhat unusual way of leveraging TPM/TXT technologies.

Degraded disk I/O without DMA. As discussed in Section V, switching back from TXT into real-mode affects DMA functionality. As of writing, we have partially identified the root cause, which we believe is Windows memory mapping being unaware of TXT, and it is not easy to customize Windows for such mapping due to the unavailability of Windows source code. As a temporary fix, we turned off DMA in Windows. Therefore, the system performs slower as in any other case when DMA is disabled. Note that this particular DMA issue is unrelated to Gracewipe's design and thus can be easily avoided. We have successfully booted up Linux with Gracewipe in TXT without any DMA problems. Gracewipe in itself is a boot-time tool, which does not run along-side the user OS.

TPM deadlock. By design, TPM NVRAM is intended to provide secure storage and protected access to confidential data. Nevertheless, such protection (especially that with authdata access) is unsuitable to be used as a general purpose encryption/decryption oracle: the key difference is that a program accessing NVRAM is expected to supply the correct authdata secret, and a failed attempt is considered as part of a guessing attack or an anomaly. For a general decryption oracle, online guessing is a non-issue, i.e., a wrong key will generate random output but no anomaly is recorded.

As we attempt to consecutively access one to three NVRAM indices with the same user password, i.e., until we can unlock a key or fail at all three authdata-protected indices, we effectively treat NVRAM authdata protection as generic decryption. Therefore, TPM actually counts each failed attempt as a violation and may enter a lockout state where TPM will not respond to subsequent operations until an explicit reset or timeout occurs; for details, see under *dictionary attack considerations* in the TPM specification [56]. TPM vendors are required to provide "some protection" against such attacks, and actual mechanisms are vendor specific [44]; more robust counter measures have also been proposed, see e.g., [10].

Therefore, no universal fix can be applied for the NVRAM deadlock. TPM_ResetLockValue may be used to reset the fail counter, and temporarily put TPM back to normal; we are however, unsure of any negative effects of issuing this command too often. As observed, our TPM chip (Winbond, chip version: 1.2.3.69) behaves as follows:

- If the TPM chip is in a fresh state (i.e., no *recent* failed attempts), or it has been reset using TPM_ResetLockValue, we see more tolerance to failed attempts (the number is between 10 and 20).
- If the TPM chip was in a lockout state, but automatically recovered because of the timeout (approximately 10-20 minutes), it can tolerate only one failed attempt before getting into lockout state again.

We relied on TPM_ResetLockValue and time-out during our development. On the positive side, the anti-guessing mechanisms raise the bar for the adversary, if he wants to mount a dictionary attack on Gracewipe passwords (of course, with the existing risk of entering the deletion password).

VIII. Security Analysis

In this section, we extend the discussion from Section II-C, and analyze possible attacks that may affect the correct functionality of Gracewipe. Note that, the verifiability of Gracewipe's execution comes from a regular TPM attestation process. Since the good values (publicly available) only rely on Intel's SINIT modules, tboot binaries and Gracewipe, as long as the PCR values (via quoting) are verified to match them, it can be guaranteed that the desired software stack has been run.

(a) Attacks on TPM. Although TPMs offer some physical tamper-resistance, they have been successfully attacked in the past (e.g., [28, 51, 49, 31, 63]). As Gracewipe relies on TPM, such attacks may affect its functionality. In a TPM reset attack [28, 49], a TPM's LRESET pin is grounded, which causes a hardware reset and thus reinitializes PCR values without rebooting the computer. This exploits the low speed of the LPC bus. It has been verified with SRTM but no proof of effectiveness can be found yet with DRTM. If feasible, it may have two impacts. (a) The adversary will be able to attest to any forged Gracewipe binary as genuine due to the PCR values of his choice. However, as he is the verifier in this scenario, he has no incentive to do so. (b) The attacker can also unseal C (with the forged correct measurements). This allows him to mount an offline dictionary attack against $PH/PD/PN$, equivalent to when no C is used.

TPM pins can also reveal the data being transferred, if not encrypted. In TPM 1.2, encrypted communication is supported; however, metadata and command ordinals are still in the cleartext. Therefore, the adversary may detect the deletion attempt (by filtering certain commands) and interrupt it, e.g., cutting the power, or punish the victim (see an experiment by Kursawe et al. [31]). This attack can be addressed in two ways:

use a TPM chip that does not expose pins to probing (e.g., integrated TPM in SuperIO [53]); or, perform an NVRAM write (as required in our deletion operation), each time a password is entered, i.e., effectively making Gracewipe always behave the same on all entered passwords.

(b) Evil-maid attacks. In 2009, Rutkowska demonstrated the possibility of an evil-maid attack [43] (also termed as *bootkit* by Kleissner [30] in a similar attack) against software-based FDEs. The key insight is that the MBRs must remain unencrypted even for FDE disks, and thus can be tampered with. We consider two situations directly applicable to Gracewipe: 1) In normal operation (i.e., not under duress), the user may expose her password for the hidden system (PH). As soon as such an attack is suspected (e.g., when PH fails to unlock the hidden volume), users must reinitialize Gracewipe, and change PH (and other attempted passwords); note that, the user is still in physical control of the machine to reset it, or physically destroy the data. 2) Under duress, we assume that the user avoids revealing PH in any case. However, the adversary may still learn valid PN/PDs as entered by the user without the risk of losing the data (due to the lack of Gracewipe protection). The use of multiple valid PDs can limit this attack. Note that if an attacker copies encrypted hidden data, and then collects the hidden password through an evil-maid attack, the plaintext data will still remain inaccessible to the attacker due to the use of TPM-bound secrets (see under "Sealing in NVRAM" in Section III). The attacker must steal the user machine (at least, the motherboard and disk) and launch the evil-maid attack through a look-alike machine. Existing mechanisms against evil-maid attacks, e.g., STARK [36] and MARK [19] can also be integrated with Gracewipe.

(c) Attacks on SED. Müller et al. [35] adapted several existing attacks on software-based FDEs (e.g., DMA-based, cold-boot, and evil-maid attacks), and show that these attacks are still effective against SED disks. They also identified a new, simple *hot plug* attack that can be summarized as follows. As SED conforms to the legacy ATA security standard, an SED drive is in a status of either locked or unlocked. When unlocked, although data is encrypted (write) and decrypted (read) on the fly, the output is always plaintext; thus, as long as the power cable is connected, the drive is never relocked, and accessible through another computer's SATA connection. This attack may succeed even when the victim computer is in a standby mode. Therefore, we mandate shutting down the computer before entering a possibly coercive environment.

Another possible attack is capturing the cleartext ATA password from the IDE/SATA interface. The password may be extracted by probing the interface with a logic analyzer. In a coercive situation, the user is expected only to enter the deletion password, which will erase the TPM-stored ATA password, instead of sending it to the IDE/SATA interface.

(d) Undetectable deletion trigger. As discussed under "Sealing in NVRAM" in Section III, sealing prevents guessing attacks without risking key deletion. Sealing also prevents an attacker from determining which user-entered passwords may trigger deletion, before the actual deletion occurs. If the adversary alters Gracewipe, any password, including the actual deletion password, will fail to unseal the hidden volume key from NVRAM. Since the deletion indicator (i.e., a special bit/index) lies only within the sealed data in NVRAM, the adversary will be unable to detect whether an entered password is for deletion or not (e.g., by checking the execution of a branch instruction triggered by the deletion indicator).

(e) Quoting for detecting spoofed environment. In the current implementation of Gracewipe, we only generate a quote in the case of secure deletion. However, in normal operations, the user may want to discern when a special type of evil-maid attack has happened, e.g., when the whole software stack is replaced with a similar environment (e.g., OS and applications). For this purpose, we can generate a quote each time Gracewipe is run and store it in NVRAM. By checking the last generated quote value, the user can find out if Gracewipe has been altered or not. In both secure deletion and normal operation, the selection of a proper nonce is of great importance. We currently support both arbitrary user-chosen strings and timestamps. Nevertheless, the use of a timestamp is susceptible to a pre-play attack, where one party can approximately predict (especially, if the other party is not paying attention) the time of the next use, and pre-generate a quote while actually running an altered binary. This is feasible because the malicious party has physical access, and thus, is able to use TPM to sign the well-known good PCR values for Gracewipe and the timestamp he predicts. Therefore, for spoofed environment detection, we recommend the use of user-chosen strings during quote generation, although it requires user intervention.

(f) Booting from non-Gracewipe media. The attacker may try to bypass Gracewipe by booting from other media. For an SED-based implementation, such attempts cannot proceed (i.e., the disk cannot be mounted). Even if he can mount the disk, e.g., with a copy of Gracewipe-unaware TrueCrypt, he must use the unmodified version of Gracewipe to try passwords that are guessed or extracted from the user (e.g., under coercion), as TrueCrypt volumes are now encrypted with long random keys (e.g., 256-bit AES keys), as opposed to password-derived keys. Brute-forcing such long keys is assumed to be infeasible even for state-level adversaries.

(g) User diligence. We require users to understand how security goals are achieved in Gracewipe, and diligently choose which password to use depending on a given context. If the deletion password is entered accidentally, the protected data will be lost without any warning, or requiring any confirmation. Note that, we do not impose any special requirement on password choice; i.e., users can choose any generally-acceptable decent passwords (e.g., 20 bits of entropy may suffice). We do not mandate *strong* passwords, as the adversary is forced to guess passwords online, and always faces the risk of guessing the deletion password. Also, the user must reliably destroy her copy of the TrueCrypt keys or ATA keys when passing them to configure TrueCrypt or SED devices. We can automate this key setup step at the cost of enlarging the trusted computing base. However, we believe that even if the whole process is without any user intervention, the adversary may still suspect the victim to have another copy of the key or the confidential data. Here we only consider destroying the copy that the adversary has captured.

IX. Related Work

Solutions related to secure deletion have been explored extensively both by the research community and the industry; see e.g., the recent survey [38]. However, we are unaware of solutions that target verifiability of the deletion procedure,

and unobservability and indistingushability of the triggering mechanism—features that are particularly important in the threat model we assumed. Here we summarize proposals related to secure deletion and coercive environment.

Limited-try approach [40]. In a blog post, Rescorla [40] discusses technical and legal problems of data protection under coercion. Limitations of existing approaches including deniable encryption (such as TrueCrypt hidden volumes), verifiable destruction (Vanish [17]) have been discussed. He also proposes possible solutions, one of which is based on leveraging a hardware security module (HSM) with a *limited-try* scheme. The HSM will delete the encryption key if wrong keys are entered a limited number of times. As mentioned [40], such a system cannot be software-only as the destruction feature can be easily bypassed. Essentially, Gracewipe combines TPM and TXT to achieve HSM-like guarantees, i.e., isolated and secure execution with secure storage (albeit limited tamper-resistance), without requiring HSMs.

Authentication under duress. Clark and Hengartner [11] explore panic passwords for authentication under duress. They discuss limitations of schemes involving two passwords (regular/panic), introduce a comprehensive threat model that considers the attacker's persistence (iterated and randomized sequence of attempts), and propose new panic password schemes. Gracewipe prototypes currently have been implemented to use only one deletion password. We plan to incorporate some of their new schemes to provide support for multiple deletion passwords; however, we are still unsure about how current TPM NVRAM access restrictions may affect such extensions (see under "TPM deadlock" in Section VII). As discussed [11], the use of a panic password, either by the victim or the adversary, must be undetectable; we also require this feature in Gracewipe. The outcome of an entered panic password must also be unobservable, e.g., a submitted vote with a panic password appears to be cast but is actually discarded by the server-end. However, in Gracewipe, after the key deletion has been performed, we enable the victim to prove to a reasonable adversary that Gracewipe has been executed with deletion, and further questioning the victim is of no use.

Beyond panic passwords, recently, two other proposals explore comprehensive mechanisms for authentication under duress. Gupta and Gao [20] propose to use *skin conductance* to derive authentication secrets such as passwords or keys. Experimental results show that skin conductance changes significantly between normal and stressed situations, implying that when under duress a victim cannot *reveal* the actual key, even if she wants to cooperate. However, the effects of using relaxation drugs on the victim by an adversary have not been explored. Bojinov et al. [6] propose to use *implicitly learned* passwords, which cannot be revealed by an user (due to no explicit knowledge); the passwords are implicitly learned by a user through the use of a specially crafted game. Both enrolment and authentication phases require significantly more time than regular authentication mechanisms (30–45 minutes and 5–6 minutes, respectively in their experiments).

Secure deletion survey [38]. Reardon et al. provide a comprehensive survey of existing solutions for secure deletion of user data on physical media, including flash, and magnetic disks/tapes. Solutions are categorized and compared based on how they are interfaced with the physical media (e.g., via user-level applications, file system, physical/controller layers), and the features they offer (e.g., deletion latency, target adversary and device wear). However, SED-based solutions were not evaluated, which is of significance to secure deletion. The authors also presented a taxonomy of adversaries that a secure deletion approach is faced with. The adversary in Gracewipe can be classified as *bounded coercive* as he can detain the victim, and keep the device for as long as he needs with all hardware tools available, but cannot decrypt the Gracewipe-protected data without the proper key. Reardon et al. also discuss a few solutions involving encrypting user data and making it inaccessible by deleting the keys. The authors suggested to be more cautious about such cryptographic deletion and consider the adversary's true *computational bound* (which would be rather high for a state-level adversary).

STARK [36]. Müller et al. propose techniques for mutual authentication between humans and computers, arguing that forged bootloader can trick the user to leak her password (cf. [43, 30]). However, even with TPM sealing, attacks aiming to just obtain the user secret can still occur, as demonstrated by the tamper-and-revert attack to BitLocker [61]. STARK allows the user to set up a sealed user-chosen message, which should be unsealed by the machine before it authenticates the user. The user can then verify if it is her message. Nevertheless, if the attacker stealthily turns on the machine and notes down the unsealed secret, he can still forge the bootloader. As a countermeasure, after each successful login, STARK replaces the old message with a new user-chosen one to prevent such replay attacks (hence the message is termed as *monce*, as a message counterpart of nonce). In addition, the monce is stored on a separate USB disk that serves as a physical key to bootstrap the process credibly. Gracewipe may be extended with such techniques to defeat evil-maid attacks.

DriveCrypt Plus Pack [47]. DCPP can be considered the closest prior art to Gracewipe. It is a closed-source FDE counterpart of TrueCrypt, with several interesting features, such as deniable storage (hidden volumes), the use of USB tokens for multi-factor authentication, UEFI support, destruction passwords and security by obfuscation. A user can define one or two destruction passwords (when two are defined, both must be used together), which, if entered, can immediately cause erasure of some regions of the hard drive, including where the encryption keys are stored. What DCPP is obviously still missing is a trusted environment for deletion trigger, and measurement for the deletion environment. The adversary may also alter DCPP (e.g., through binary analysis) to prevent the deletion from happening. More seriously, the adversary can clone the disk before allowing any password input.

X. Conclusion

We consider a special case of data security: making data permanently inaccessible when under coercion. We want to enable such deletion with additional guarantees: (1) verification of the deletion process; (2) indistingushability of the deletion trigger from the actual key unlocking process; and (3) no password guessing without risking key deletion. If key deletion occurs through a user supplied deletion password, the user may face serious consequences (legal or otherwise). Therefore, such a deletion mechanism should be used only for very high-value data, which must not be exposed at any cost, and where even accidental deletion is an acceptable risk (i.e., the data may be

backed up at locations beyond the adversary's reach). We design and implement Gracewipe for two widely-available real-world FDE schemes: TrueCrypt with hidden volume support and SED; both prototypes eventually boot Windows, which poses some additional challenges. We introduce no additional hardware/architectural requirements. We use TPM for secure storage and enforcing loading of an untampered Gracewipe environment. For secure and isolated execution, we rely on Intel TXT. Millions of consumer-grade machines are already equipped with a TPM chip and TXT/SVM capable CPU. Thus, Gracewipe can immediately benefit its targeted user base. The source code of our prototypes can be obtained via: https://madiba.encs.concordia.ca/software.html.

Acknowledgement

We are grateful to anonymous NDSS2015 reviewers, Jonathan McCune, N. Asokan, Thomas Nyman, and Jeremy Clark for their insightful suggestions and advice. We also thank the members of Concordia's Madiba Security Research Group for their enthusiastic discussion on this topic. The second author is supported in part by an NSERC Discovery Grant and FRQNT Programme établissement de nouveaux chercheurs.

References

[1] GRUB4DOS. http://sourceforge.net/projects/grub4dos/.

[2] TrouSerS: The open-source TCG software stack. Version: 0.3.8. http://trousers.sourceforge.net/.

[3] 16s.us. TCHunt. Tool for detecting encrypted hidden volumes (version: 1.6, release date: Jan. 29, 2014). http://16s.us/software/TCHunt/.

[4] AMD.com. AMD64 architecture programmer's manual volume 2: System programming. Oct. 2013. http://support.amd.com/TechDocs/24593.pdf.

[5] R. Anderson, R. Needham, and A. Shamir. The steganographic file system. In *International Workshop on Information Hiding (IH'98)*, Portland, OR, USA, 1998.

[6] H. Bojinov, D. Sanchez, P. Reber, D. Boneh, and P. Lincoln. Neuroscience meets cryptography: Designing crypto primitives secure against rubber hose attacks. In *USENIX Security Symposium*, Bellevue, WA, USA, Aug. 2012.

[7] T. Bonaci, J. Herron, C. Matlack, and H. J. Chizeck. Securing the exocortex: A twenty-first century cybernetics challenge. In *IEEE Conference on Norbert Wiener in the 21st Century*, Boston, MA, USA, June 2014.

[8] D. Boneh and R. J. Lipton. A revocable backup system. In *USENIX Security Symposium*, San Jose, CA, USA, July 1996.

[9] G. Chappell. The x86 BIOS emulator. http://www.geoffchappell.com/studies/windows/km/hal/api/x86bios/index.htm?tx=7.

[10] L. Chen and M. Ryan. Attack, solution and verification for shared authorisation data in TCG TPM. In *Formal Aspects in Security and Trust (FAST'09)*, Eindhoven, The Netherlands, Nov. 2009.

[11] J. Clark and U. Hengartner. Panic passwords: Authenticating under duress. In *USENIX Workshop on Hot Topics in Security (HotSec'08)*, San Jose, CA, USA, July 2008.

[12] CNet.com. Turkish police may have beaten encryption key out of TJ Maxx suspect. News article (Oct. 24, 2008). http://news.cnet.com/8301-13739_3-10069776-46.html.

[13] G. D. Crescenzo, N. Ferguson, R. Impagliazzo, and M. Jakobsson. How to forget a secret (extended abstract). In *Symposium on Theoretical Aspects of Computer Science (STACS'99)*, Trier, Germany, Mar. 1999.

[14] A. Czeskis, D. J. S. Hilaire, K. Koscher, S. D. Gribble, T. Kohno, and B. Schneier. Defeating encrypted and deniable file systems: TrueCrypt v5.1a and the case of the tattling OS and applications. In *USENIX HotSec'08*, San Jose, CA, USA, 2008.

[15] Dban.org. Darik's boot and nuke. Open-source tool for hard-drive disk wipe and clearing. http://www.dban.org.

[16] M. Frank, T. Hwu, S. Jain, R. Knight, I. Martinovic, P. Mittal, D. Perito, and D. Song. Subliminal probing for private information via EEG-based BCI devices. Tech-report (Dec. 20, 2013). http://arxiv.org/abs/1312.6052.

[17] R. Geambasu, T. Kohno, A. Levy, and H. M. Levy. Vanish: Increasing data privacy with self-destructing data. In *USENIX Security Symposium*, Montreal, Canada, Aug. 2009.

[18] Gnu.org. The multiboot specification. http://www.gnu.org/software/grub/manual/multiboot/multiboot.html.

[19] J. Götzfried. Mutual authentication to resist keylogging. Version: 0.2, Oct. 2013. https://www1.informatik.uni-erlangen.de/filepool/projects/mark/index.html.

[20] P. Gupta and D. Gao. Fighting coercion attacks in key generation using skin conductance. In *USENIX Security Symposium*, Washington, DC, USA, Aug. 2010.

[21] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *USENIX Security Symposium*, San Jose, CA, USA, July 1996.

[22] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, San Jose, CA, USA, 2008.

[23] HGST.com. Data security from the inside out. http://www.hgst.com/hard-drives/self-encrypting-drives.

[24] Intel.com. Trusted boot (tboot). Version: 1.8.0. http://tboot.sourceforge.net/.

[25] Intel.com. *Intel 64 and IA-32 Architectures Software Developer's Manual*, September 2014. Volume 2C: Instruction Set Reference.

[26] Intel.com. *Intel 64 and IA-32 Architectures Software Developer's Manual*, September 2014. Volume 2A: Instruction Set Reference, A–M.

[27] ISO.org. ISO/IEC FDIS 27040: Information technology – security techniques – storage security. Target publication: Apr. 21, 2015. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=44404.

[28] B. Kauer. OSLO: Improving the security of trusted computing. In *USENIX Security Symposium*, Boston, MA, USA, Aug. 2007.

[29] Kernel.org. Trusted boot (tboot) documentation. https://www.kernel.org/doc/Documentation/intel_txt.txt.

[30] P. Kleissner. Stoned bootkit. Black Hat USA (July 2009). http://www.blackhat.

com/presentations/bh-usa-09/KLEISSNER/BHUSA09-Kleissner-StonedBootkit-PAPER.pdf.

[31] K. Kursawe, D. Schellekens, and B. Preneel. Analyzing trusted platform communication. In *ECRYPT Workshop, CRASH – CRyptographic Advances in Secure Hardware*, Leuven, Belgium, Sept. 2005.

[32] A. D. McDonald and M. G. Kuhn. StegFS: A steganographic file system for Linux. In *International Workshop on Information Hiding (IH'99)*, Dresden, Germany, 1999.

[33] Microsoft.com. Windows kernel debugger. http://msdn.microsoft.com/en-us/library/windows/hardware/dn745912(v=vs.85).aspx.

[34] T. Müller, F. C. Freiling, and A. Dewald. TRESOR runs encryption securely outside RAM. In *USENIX Security Symposium*, San Francisco, CA, USA, Aug. 2011.

[35] T. Müller, T. Latzo, and F. Freiling. Self-encrypting disks pose self-decrypting risks: How to break hardware-based full disk encryption. Technical report, Friedrich Alexander University, 2012. http://www1.cs.fau.de/sed.

[36] T. Müller, H. Spath, R. Mäckl, and F. C. Freiling. STARK tamperproof authentication to resist keylogging. In *Financial Cryptography and Data Security (FC'13)*, Okinawa, Japan, Apr. 2013.

[37] M. J. Ranum. Cryptography and the law... Newsgroup post at sci.crypt (Oct. 16, 1990). https://groups.google.com/forum/#!msg/sci.crypt/W1VUQlC99LM/ANkI5zdGQIYJ.

[38] J. Reardon, D. Basin, and S. Capkun. SoK: Secure data deletion. In *IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, May 2013.

[39] J. Reardon, S. Capkun, and D. Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *USENIX Security Symposium*, Bellevue, WA, USA, Aug. 2012.

[40] E. Rescorla. Protecting your encrypted data in the face of coercion. Blog post (Feb. 11, 2012). http://www.educatedguesswork.org/2012/02/protecting_your_encrypted_data.html.

[41] M. Russinovich. Inside the Windows Vista kernel: Part 3. *Microsoft TechNet Magazine*, Apr. 2007.

[42] M. Russinovich, D. A. Solomon, and A. Ionescu. *Windows Internals Part 1*. Microsoft Press, 2012.

[43] J. Rutkowska. Evil maid goes after TrueCrypt! Online report (Oct. 16, 2009). http://theinvisiblethings.blogspot.ca/2009/10/evil-maid-goes-after-truecrypt.html.

[44] A.-R. Sadeghi, M. Selhorst, C. Stüble, C. Wachsmann, and M. Winandy. TCG inside? – a note on TPM specification compliance. In *ACM Workshop on Scalable Trusted Computing (STC'06)*, Alexandria, VA, USA, Nov. 2006.

[45] Salon.com. James Holmes and the ethics of "truth serum": Putting the Aurora shooter through a narcolanalytic interview won't provide truth or prove sanity. News article (Mar. 13, 2013). http://www.salon.com/2013/03/13/james_holmes_the_ethics_efficacy_of_truth_serum/.

[46] Seagate.com. Protect your data with Seagate secure self-encrypting drives. http://www.seagate.com/tech-insights/.

[47] SecurStar.com. DriveCrypt Plus Pack. http://www.securstar.com/disk_encryption.php.

[48] M. M. G. Slusarczuk, W. T. Mayfield, and S. R. Welke. Emergency destruction of information storing media. Institute for Defense Analyses Report R-321 (Dec. 1987). http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA202147.

[49] E. R. Sparks. A security assessment of trusted platform modules. Technical report, Dartmouth College, 2007. http://www.cs.dartmouth.edu/reports/TR2007-597.pdf.

[50] R. Strong. BIOS enhanced disk drive services 4.0. Technical report, Intel.com, 2008. http://www.t13.org/documents/UploadedDocuments/docs2008/e08134r1-BIOS_Enhanced_Disk_Drive_Services_4.0.pdf.

[51] C. Tarnovsky. Security failures in secure devices. Black Hat DC (Feb. 2008). https://www.blackhat.com/presentations/bh-dc-08/Tarnovsky/Presentation/bh-dc-08-tarnovsky.pdf.

[52] TheRegister.co.uk. Computing student jailed after failing to hand over crypto keys. News article (July 8, 2014). http://www.theregister.co.uk/2014/07/08/christopher_wilson_students_refusal_to_give_up_crypto_keys_jail_sentence_ripa/.

[53] ThinkWiki.org. Integrated TPM in SuperIO chip. http://www.thinkwiki.org/wiki/Embedded_Security_Subsystem.

[54] TrueCrypt.org. Free open source on-the-fly disk encryption software. Version 7.1a (July 2012). http://www.truecrypt.org/.

[55] Trusted Computing Group. *TCG PC Client Specific TPM Interface Specification (TIS)*. Specification Version 1.3 (March 21, 2013).

[56] Trusted Computing Group. *TPM Main: Part 1 Design Principles*. Specification Version 1.2, Level 2 Revision 116 (March 1, 2011).

[57] Trusted Computing Group. *TPM Main: Part 2 TPM Structures*. Specification Version 1.2, Level 2 Revision 116 (March 1, 2011).

[58] Trusted Computing Group. *TPM Main: Part 3 Commands*. Specification Version 1.2, Level 2 Revision 116 (March 1, 2011).

[59] Trusted Computing Group. *TCG PC Client Specific Implementation Specification for Conventional BIOS*, February 2012.

[60] Trusted Computing Group. *TCG Storage Security Subsystem Class: Opal*, February 2012.

[61] S. Türpe, A. Poller, J. Steffan, J.-P. Stotz, and J. Trukenmüller. Attacking the BitLocker boot process. In *Trusted Computing (Trust'09)*, Oxford, UK, Apr. 2009.

[62] A. Winter. The making of "truth serum," 1920-1940. *Bulletin of the History of Medicine*, 79(3):500–533, 2005.

[63] J. Winter and K. Dietrich. A hijacker's guide to communication interfaces of the trusted platform module. *Computers and Mathematics with Applications*, 65(5):748–761, Mar. 2013.

[64] R. Wojtczuk and J. Rutkowska. Attacking Intel trusted execution technology. Black Hat DC (Feb. 2009). http://www.blackhat.com/presentations/bh-dc-09/Wojtczuk_Rutkowska/BlackHat-DC-09-Rutkowska-Attacking-Intel-TXT-slides.

pdf.

[65] R. Wojtczuk, J. Rutkowska, and A. Tereshkin. Another way to circumvent Intel trusted execution technology. Technical report, Invisible Things Lab, 2009. http://invisiblethingslab.com/resources/misc09/Another%20TXT%20Attack.pdf.

[66] X3T10 Technical Committee. *Information Technology - AT Attachment Interface with Extensions (ATA-2)*, March 1996. http://www.t13.org/Documents/UploadedDocuments/project/d0948r4c-ATA-2.pdf.

APPENDIX A
DEBUGGING CONSIDERATIONS

We faced several issues while debugging our prototypes, as Gracewipe works at an early stage during system boot, and involves several existing tools/components, which are unaware of each other. Here, we briefly discuss few selected challenges and how we derived fixes for them.

The debugging complexity partly arises from the diverse forms of Gracewipe components, specifically: (i) the TrueCrypt modules are gzip'ed and written on contiguous sectors following the MBR, and only extracted at run-time; (ii) different boot loaders (e.g., Windows VBR) are located on encrypted volumes, and only available after run-time decryption; (iii) unavailability of source code (e.g., Windows BootMgr); and (iv) invocation of our BIOS interrupt handler in different caller contexts. Also, as of writing, no x86 emulator with TXT support exists, and thus it is infeasible to rely on simulation-based debuggers.

A. Debugging methods and tools

We use a combination of different methods/tools. From the time tboot gets control before the wiper switches to real-mode for chainloading, we print debug information on the screen (similar to tboot). Around the mode switch (about 30 lines of assembly), we rely on less efficient *dead loops* to locate a failing point (i.e., using "`jmp $`" to halt the system and moving this instruction around suspicious code). When Windows takes over, we make use of WinDbg [33]. It has two modes, *Windows Boot Debugger* for debugging Windows BootMgr, and *Windows Kernel Debugger* when connected to the Windows kernel.

We need an out-of-band measure and simulation-based debugging to identify effects of TXT on Windows loading. Since no support for TXT exists in off-the-shelf emulators, we managed to carefully separate the function sets that can be tested without TXT. For instance, if we load the wiper right after GRUB, instead of invoking tboot and disable all TXT related operations in the first stage of the wiper, we can execute the whole solution in QEMU and boot Windows there. In this way, gdb can be used to connect to QEMU via TCP/IP (locally) and debug components other than tboot. When Windows starts up normally, we can add the measurement back with tboot on a physical machine. What we have achieved in QEMU still helps, in the sense that by comparing the behaviors we can tell how tboot contributes to the difference (because the one without tboot already works).

For smaller-sized binary components, we managed to disassemble and analyze them directly (e.g., Windows VBR, 512 bytes). For TrueCrypt modules which are more complicated, we made use of the .cod files generated in TrueCrypt project to correlate the source and the binary (i.e., mapping assembly blocks to C functions).

B. Example debugging scenarios

Here we list a few typical situations that we addressed.

Interpreting BIOS interrupt return codes. During loading Windows VBR, we received a message "A disk read error occurred, Press Ctrl+Alt+Del to restart" and the system was halted. Till this point, the TrueCrypt hidden volume has been successfully mounted and accessible through regular INT 13h calls. Technically, Windows 7/Vista must be run with INT 13h Extensions (function codes 0x4X instead of 0x0X). It is queried through function code 0x41, and if supported, CF should be cleared with BX set to 0xAA55. However, in addition to checking CF and BX, the VBR also specifically verifies whether CX is 1 (`TEST CX,0001`), which implies that the extension supports device access using the packet structure, the way LBA works. We worked around it by restoring the original INT 13h handler and reverse-engineered (with gdb and QEMU) the return of function 0x41. We had to set CX to 7 (0b111) to stop VBR from complaining. As the value 7 also includes BIOS enhanced disk drive services [50] (EDD) that supports 64-bit LBA as part of the extension (0b100), we believe Windows mandates EDD.

Dealing with Windows BootMgr and Shadowed BIOS. BootMgr runs in protected-mode and is considered as the bootloader of Windows. Right after we selected an item in BootMgr, a system halt occurred with only one line of message: "BlInitializeLibrary failed 0xc0000001". With the help of Windows Boot Debugger, we identified and solved this problem by adding the support of function code 0x15 (cf. Section V-B).

To our surprise, Windows still invokes our interrupt handler even in winload.exe (which loads the kernel) with *hal!x86BiosExecuteInterruptShadowed*. Windows has a mechanism called "Shadowed BIOS" that allows to continue using BIOS calls while it is initializing in protected-mode, perhaps before some necessary drivers have been loaded (see [9]).

Debugging TrueCrypt kernel driver in Windows. As soon as winload.exe finishes loading the kernel and boot-class device drivers, multithreading and other large Windows modules make black-box debugging impossible. The Windows Driver Kit (WDK[8]) allows us to insert debug output into the source of truecrypt.sys to be printed at the console of Windows Kernel Debugger; note that truecrypt.sys is the Windows driver that takes over the job of INT 13h filter for real-time decryption/encryption. In function *MountDrive()*, we printed out the variables and the buffer read as the volume header from the disk. We could clearly see how the data read with DMA differed from the PIO mode, which caused the mismatch of the TrueCrypt boot drive signature. This solved the UNMOUNTABLE_BOOT_VOLUME error as discussed in Section V-E.

[8] http://www.microsoft.com/en-ca/download/details.aspx?id=11800