Authentication and Data Protection under Strong Adversarial Model

Lianying Zhao

A thesis
in
The Concordia Institute
for
Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy (Information and Systems Engineering) at
Concordia University
Montréal, Québec, Canada

July 2018

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Mr. Lianying Zhao**

Entitled: **Authentication and Data Protection under Strong Adversarial Model**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Information and Systems Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

Dr. Ketra Schmitt	Chair
Dr. Urs Hengartner	External Examiner
Dr. Otmane Ait Mohamed	External to Program
Dr. Jeremy Clark	Examiner
Dr. Lingyu Wang	Examiner
Dr. Mohammad Mannan	Thesis Supervisor

Approved by ______________________________
Dr. Chadi Assi, Graduate Program Director

June 21, 2018 ______________________________
Dr. Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

Authentication and Data Protection under Strong Adversarial Model

Lianying Zhao, Ph.D.

Concordia University, 2018

We are interested in addressing a series of existing and plausible threats to cybersecurity where the adversary possesses unconventional attack capabilities. Such unconventionality includes, in our exploration but not limited to, crowd-sourcing, physical/juridical coercion, substantial (but bounded) computational resources, malicious insiders, etc. Our studies show that unconventional adversaries can be counteracted with a special anchor of trust and/or a paradigm shift on a case-specific basis.

Complementing cryptography, hardware security primitives are the last defense in the face of co-located (physical) and privileged (software) adversaries, hence serving as the special trust anchor. Examples of hardware primitives are architecture-shipped features (e.g., with CPU or chipsets), security chips or tokens, and certain features on peripheral/storage devices. We also propose changes of paradigm in conjunction with hardware primitives, such as containing attacks instead of counteracting, pretended compliance, and immunization instead of detection/prevention.

In this thesis, we demonstrate how our philosophy is applied to cope with several exemplary scenarios of unconventional threats, and elaborate on the prototype systems we have implemented. Specifically, *Gracewipe* is designed for stealthy and verifiable secure deletion of on-disk user secrets under coercion; *Hypnoguard* protects in-RAM data when a computer is in sleep (ACPI S3) in case of various memory/guessing attacks; *Uvauth* mitigates large-scale human-assisted guessing attacks by receiving all login attempts in an indistinguishable manner, i.e., correct credentials in a legitimate session and incorrect ones in a plausible fake session; *Inuksuk* is proposed to protect user files against ransomware or other authorized tampering. It augments the hardware access control on self-encrypting drives with trusted execution to achieve data immunization. We have also extended the Gracewipe scenario to a network-based enterprise environment, aiming to address slightly different threats, e.g., malicious insiders.

We believe the high-level methodology of these research topics can contribute to advancing the security research under strong adversarial assumptions, and the promotion of software-hardware orchestration in protecting execution integrity therein.

Acknowledgments

My thesis supervisor, Dr. Mohammad Mannan, has always been the driving force, reliable support and rigorous guide for my Ph.D. research, which lead to my smooth conversion to an academic mindset from the industry. I appreciate the meticulousness and rigorousness that are “genetically” implanted into my research habits and even the way I think, just because of him.

I am grateful for all members of Madiba Security Research Group (especially Xavier de Carné de Carnavalet), as well as the rest of my research colleagues of the CIISE department, for the pleasant discussions we have had, regarding research, career and life. When disappointment or frustration comes, it is them who make me look forward.

I also would like to express my gratitude to my family members, who have been backing me up and understanding the devoted nature of Ph.D. studies, without which this thesis would not have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

What if a user of 256-bit AES encrypted confidential data is forced to give away the key? What if rootkit malware terminates antivirus and other security tools on a computer? What if an adversary with physical access extracts sensitive information from a stolen laptop? Such questions lead to identification of certain attack vectors that have not been (sufficiently) considered in most state-of-the-art security solutions. In this thesis, we explore approaches to such security problems under a strong adversarial model.

1.1 Unconventional Attack Capabilities

The notion of unconventional attack vectors largely comes from observation. Individually such vectors may have overlap in their real-world instances (e.g., an attack can involve both coercion and physical control). Nevertheless, this does not affect their significance or usefulness in identifying unsolved problems and studying them based on the commonness.

Definition. In the domain of authentication and data protection (e.g., for integrity and confidentiality), we consider attack vectors that have already been included in the state-of-the-art threat models in both academic and industrial solutions as *conventional*; and those that have not been (sufficiently) addressed are considered *unconventional*. For instance, rootkit ransomware is an unconventional attack vector because it is only found formally considered in one academic proposal [122] with a confined solution.

In this section, to facilitate discussion in the subsequent chapters, we classify the defined unconvential attack vectors/capabilities as follows:

1. Physical access. Unlike traditional threat models in network communications [70], physical access to computer systems can also be a serious threat to today's cybersecurity. The adversary can tap exposed pins and eavesdrop on any traffic, manipulate the computer execution (e.g., by warm/cold boot [55, 288, 287, 108]), or exploit side channels to extract/learn secrets, such as in DMA ([246, 174]) and physical side-channel attacks [89]. Especially, when combined with abundant computational resources, offline dictionary/guessing attacks [198] can become a serious threat to encrypted data.

2. Coercion. The victim can be either coerced physically [224] or threatened with a consequence, such as imprisonment (e.g., law enforcement), torture, harming a hostage or revealing a scandal. In this situation, the victim can be forced to unlock the system or decrypt the secret the same way he usually does it, but for the adversary.

3. Human assistance. In automated large-scale attacks, certain logic is used to distinguish machine from human, such as Captchas [152]. The adversary can forward requests to sweatshops in the underground market (or through crowdsourcing [191]), or even trick legitimate users visiting compromised legitimate websites into solving them.

4. Privileged programs. Most security solutions assume the adversarial entity to be lower privileged (e.g., antivirus tools assuming a user-space adversary and hypervisor-based tools assuming guest OS only). However, rootkit malware has been around for decades (as long as malware exploits the "root" privilege) and rootkit ransomware's existence is also obvious (e.g., Petya [215]). More importantly, even systems booted with secure/trusted booting are still subject to run-time exploitation, potentially escalating to the highest privilege.

5. Privileged personnel. To perform the designated tasks in an organization, employees like system administrators must be granted the highest privilege. If they turn hostile (e.g., due to being laid off), a serious harm can be caused [210, 177] to the organization.

1.2 Methodology

Two major principles are reflected throughout all the research topics and form the foundation of our research methodology.

1.2.1 Hardware security primitives – P1

What computer systems do can be categorized into execution (performing the task) and communications (input, output, or networking). Securing communications largely depends on cryptography with properly designed protocols, where dedicated hardware is not essential, as the assumed worst-case scenario is communication channel compromise (e.g., [185]), but the communicating parties are never able to access each other's internal state (e.g., processor, memory, user input), or otherwise it is no longer a communications problem. However, securing execution usually involves both software and hardware, because the adversary may co-locate with the code in question and see any secrets used in crypto operations. The legacy approaches mostly rely on isolation and access control provided by the OS or hypervisor.This might be insufficient in the face of the aforementioned unconventional attack vectors.

Both the physical and privilege vectors are eventually an arms race of who preempts whom, in terms of restriction enforcement. Intuitively, hardware can attain the lowest protection level (highest privilege) in the battle with various adversaries. This comes in the forms of "minus" privileges if it participates in the CPU execution, hardware-isolated crypto- or access-operations as a co-processing device, and self-contained security features as a stand-alone peripheral/storage device. See Figure 1 for an incomplete classification of hardware security primitives.

Architecture-shipped features. Anchoring the root of trust at or close to the processor helps make the TCB (Trusted Computing Base) minimal. Architecture-shipped features provide isolated execution, code integrity/measurement, attestation, memory encryption, etc. They are exposed to the applications as additional CPU instructions or I/O-mapped resources. They form the foundation of trusted computing and an intrinsic starting point for addressing unconventional attack vectors.

On-board devices. The category of on-board devices in itself has some ambiguity because of the rapidly developing semiconductor technologies and the same functionality shifts between circuit boards and microchips as their design evolves. Generally, on-board devices have a strong bond with the processor, e.g.,

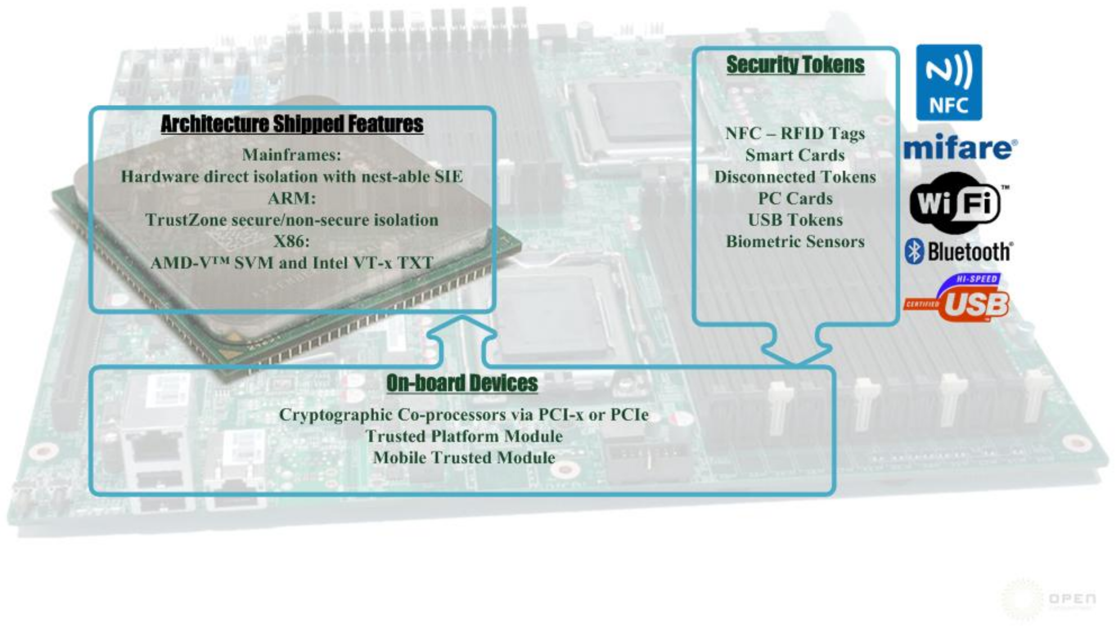


Figure 1: An overview of security features in hardware

co-processing dependently or complementing processor functions. They provide sometimes architecture-agnostic support that the processor does not already have.

Security tokens. As opposed to the aforementioned categories, a security token is totally removable from the system. An additional implication by its name is that their functions are usually self-contained and do not need CPU intervention. Such devices are often used for authentication (physical access control, presence check) or digital signature purposes. If generalized, many of such products help enable two-factor authentication [176] (or even multi-factor authentication). Moreover, we also consider in the same category any on-device security feature such as that of hard drives.

We make use of hardware security features where appropriate as the primary research principle (P1).

1.2.2 Passive but resilient defense – P2

Reacting to threats head-on is a common practice in cybersecurity, as exemplified by the current variety of tools/proposals for detection, prevention and destruction of, and sometimes recovery from malware or network intrusion. Deception is one exception (cf. [233, 310, 59, 27]), which is applied more in networking or multi-host scenarios, for the purpose of attack monitoring, data collection and anomaly signaling (honeypots or honeynet).

With this consideration in mind, we are inspired by the philosophy of T'ai Chi [154], which advocates yielding instead of counteracting, to achieve equivalent or even better defense in face of powerful attacks. When attacks are assumed inevitable, we propose to use redirection (for protecting the target), undetectable destruction (of the target), and immunization (to armor the target).

This principle (P2) is combined with and enhances the hardware security primitives.

1.3 Thesis Statement

Our research is aimed at solving selected typical security problems caused by unconventional attack vectors, which fall in (or at the intersection of) authentication, data integrity and data confidentiality, using the combination of hardware-assisted security approaches (P1) and passive but resilient defense (P2).

Our proposed approaches may serve as a lead-in to more formalization and generalization, into how today's hardware security primitives can fortify the defense against unconventional attack capabilities under the strong adversarial model.

1.4 Main Contributions

Our research is aimed at solving several realistic security scenarios caused by unconventional attack vectors, and as a lead-in to more formalization, shedding some light on the application (and importance) of hardware-assisted security approaches under the strong adversarial model. The selected practical scenarios not being concentrated in a narrow area indicates the actuality and prevalence of the unconventional attack vectors.

In this section, we advance the problems and solutions of our research topics from their individual chapters as follows. The proposed solutions target a primary concern in the first place and usually also solve a wider scope of related problems.

- *Protecting data-at-rest against coercion.* As the Achilles' heel of cryptography, giving away the decryption key leaves little space for traditional mechanisms to work. For instance, in physical attacks when the attacker has full control over the target machine, or he can coerce the machine owner into revealing decryption passwords, most traditional defenses might be defeated. We believe such a strong threat is in accordance with current state-level adversaries with high technical capabilities and legal/questionable/illegal powers [271] (e.g., US FISA, clandestine NSA programs, physical/psychological tortures). **Gracewipe** (Chapter 2, full implementation, published) addresses such threats as well as offline guessing/dictionary attacks, enabling stealthy and indistinguishable secure deletion (P2) with verification and machine state binding (P1). The deletion trigger is undetectable by the adversary; the deletion process is prompt, and once completed, can be verifiable by a diligent adversary; finally but more importantly the adversary is also held back from offline brute-forcing without relying on the victim or Gracewipe.

- *Secure remote data erase.* When the aforementioned encrypted disk data is stored in a remote computer owned by a special professional (e.g., human right activist) or an enterprise (e.g., about to announce a lay-off plan), being able to securely and remotely destroy data is needed. In this case, convincing verifiability is also necessary but to the user. **Gracewipe Remote** (Chapter 3, full implementation) inherits Gracewipe's platform-bound encryption and guessing prevention, and further employs certain hardware-assisted remote control to securely communicate the deletion trigger to the target computer. In such end-to-end verifiable remote erase, the trust is not anchored in what the remote firmware (e.g., BIOS) says but in the attestable hardware primitive (P1). No intermediate parties other than the device manufacturer should be trusted (hence end-to-end) and especially any simple indication of success from the remote computer never suffices.

- *Protecting data-in-sleep.* Users may neglect the fact that most of the time their laptops are not in a powered-off state and the sensitive data in RAM can be extracted in several ways by the adversary. Moving from the encrypted disk data (data-at-rest) in Gracewipe over to S3 data in RAM (data-in-sleep), **Hypnoguard** (Chapter 4, full implementation, published) addresses the threats of device loss and theft as well as coercion. The Gracewipe scenario is extended to any sleeping devices not attended by the legitimate owner. The same guessing prevention and deletion triggering (P2) and hardware binding (P1) are applied. The protected in-RAM data always stays encrypted if it falls in the wrong hands and the encryption/decryption process is fast, because of hardware acceleration, so as not to downgrade user experience.

- *Data integrity against rootkit ransomware.* Turning from data confidentiality (as in Gracewipe and Hypnoguard) onto data integrity, we focus on unauthorized data alteration from rootkit malware or remote attackers. Specifically, very few proposals have taken into account that ransomware can be a rootkit at the same time. With the same privilege as or higher than that of the defense mechanism, ransomware can stay undetected or suppress any mitigation tools. As opposed to platform-bound encryption/deletion, we propose *platform-bound write-protection*, as the foundation of **Inuksuk** (Chapter 5, full implementation). It does not detect, prevent or counteract the adversary head-on where undesired alteration is initiated (P2). Instead, Inuksuk "armors" the protected data with the hardware write protection (P1) to achieve data immunization so that naturally no reaction is necessary. Gracewipe's and Hypnoguard's trusted decryption/deletion (of data-at-rest and data-in-sleep) now becomes trusted write-protection (of data-at-rest). The hardware write-protection is immune to any privileged software; meanwhile it is made difficult for the adversary to impersonate the user and bypass the write-protection.

- *COTS One-Time Programs.* Further to Inuksuk, we shift from data integrity to execution integrity. Functions that are allowed to evaluate only once and only on one set of input have many application scenarios (hence OTP – the one-time programs [92]). Seeing state-of-the-art one-time programs being mainly based on special/expensive hardware (e.g., FPGA) or involving assumptions (e.g., multi-party oblivious interaction), we propose to achieve them using commercially off-the-shelf (COTS) devices (full implementation, see Chapter 6, co-authored [1] with the work (writing/implementation) mostly done in collaboration). It can be considered as a further generalized form of Gracewipe, i.e., from enforcing correct decryption/deletion over to enforcing arbitrary logic but for a limited number (one) of times. Several variants are proposed for different security and performance requirements. Our design of the OTP can also be configured for N-time execution.

[1] The co-authors are Joseph Choi (University of Florida), Didem Demirag (Concordia University), Kevin Butler (University of Florida), Mohammad Mannan(Concordia University), Erman Ayday (Case Western Reserve University), and Jeremy Clark (Concordia University).

- *Defender-helped guessing attackers.* We consider guessing attacks against passwords in offline scenarios (as in Gracewipe and Hypnoguard). As a continuation, we also consider such attacks against password-protected online services, especially, when the attacker can use crowd-sourcing if faced with Captcha challenges. The ultimate issue of regular authentication is that a Yes/No answer is always fed back to the adversary. With that answer, he can keep guessing until a Yes is seen. He does not need to hit the rate-limit (trying different users at large scale) and it may not necessarily take that long, thanks to dictionaries and advanced algorithms. **Uvauth** (Chapter 7, published) addresses human-assisted online guessing/dictionary attacks. As with Inuksuk, it does not react in a way easily/programmatically distinguishable by the attacker, but passively contains the attack attempts (P2). The adversary should be in no way informed of the authentication results (success/failure); but the legitimate user must either discern by herself or learn the authentication outcome via human-readable channels.

1.5 Related Publications

Part of our research topics in this thesis has been peer-reviewed. The corresponding publications are listed below.

1. Explicit authentication response considered harmful. L. Zhao and M. Mannan. *New Security Paradigms Workshop 2013 (NSPW'13, pp. 77-86)*, September 9–12, 2013, Banff, Canada.

2. Gracewipe: Secure and verifiable deletion under coercion. L. Zhao and M. Mannan. *Network and Distributed System Security (NDSS'15) Symposium*, February, 8–11 2015, San Diego, CA, USA.

3. Hypnoguard: Protecting Secrets across Sleep-wake Cycles. L. Zhao and M. Mannan. *ACM Conference on Computer and Communications Security (CCS'16)*, October 24-28, 2016, Vienna, Austria.

4. Deceptive Deletion Triggers Under Coercion. L. Zhao and M. Mannan. *IEEE Transactions on Information Forensics and Security (TIFS)*, 11(12): 2763–2776, December 2016.

1.6 Outline

The rest of the thesis is organized as follows. Chapters 2, 4, 5, 7, and 6 introduce the design and detail the implementation of Gracewipe, Hypnoguard, Inuksuk, Uvauth and our TEE-based OTP, respectively. Chapter 3 discusses Gracewipe Remote, an extension of Gracewipe to address a different security scenario and its implementation. Chapter 8 explains in brief how Gracewipe can be ported to mobile platforms (with partial engineering attempt), then casts some light on certain open issues observed from our research, and concludes the thesis. Because of the wide scope of our research topics (for demonstrating the problem's prevalence and the solution's wide applicability), we have the discussion on related work in each individual chapter instead of a dedicated chapter of the thesis.

Chapter 2

Deceptive Deletion Triggers under Coercion

Coercion falls into our priority concerns under the strong adversarial model. It is considered as the Achilles' heel of cryptography. To start with, we first focus on "data at rest" with powered-off devices. In this chapter, we present our design and implementation of Gracewipe, a boot-time tool which securely unlocks/decrypts user data upon successful authentication, or stealthily and verifiably erases the decryption key through predefined password schemes in the case of coercion/emergency.

2.1 Introduction and Motivation

Plausibly deniable encryption (PDE)[1] schemes for file storage were proposed more than a decade ago; see Anderson et al. [17] for the first academic proposal (1998). In terms of real-world PDE usage, TrueCrypt [275] 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 [9]). A user may provide a *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

[1]Encryption techniques for which the plaintext cannot be proven to exist, e.g., the user can convincingly deny that a given piece of data is the outcome of encryption.

as deemed necessary by the attacker. Such coercion is also known as *rubberhose cryptanalysis* [224], which is alleged to be used in some countries (e.g., Turkey [58], USA [22]); several incidents of forced password extractions during border crossings have also been reported in the recent past (e.g., USA [247], France [273]). The use of multiple hidden volumes or security levels (e.g., as in StegFS [184]), 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, unbeknownst to the adversary (i.e., triggered in an undetectable way). 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 [66] that rely on data overwriting are not acceptable solutions (cf. [107]). Otherwise, the attacker can simply terminate the tool being used by cutting off the 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. [252]. For a quick deletion, cryptographic approaches appear to be an appropriate solution, as introduced by Boneh and Lipton [40] (see also [62, 227]). 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 [243], HGST/Western Digital [115] (cf. ISO/IEC WD 27040 [141]). SEDs allow overwriting of the data decryption 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 undetectable deletion trigger does not mean undetectable execution (e.g., 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 chapter, we discuss the design and implementation of *Gracewipe*, a solution implemented on top of TrueCrypt[2] 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 (types of) 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 (schemes with multiple *PDs* are discussed in Section 2.5). These 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. When coerced, the user can disclose *PDs* or PN, but not PH. Attackers' success probability of accessing the hidden volume can be configured to be very low (e.g., deletion after a single invalid password), and will depend on their use of user-supplied or guessed passwords, and/or the deployed variant in Gracewipe-XD; see Section 2.5. Deletion (overwriting with zeros) of the hidden volume key occurs within the TPM 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.

The relatively simple design of Gracewipe however faced several challenges when implemented with real-world systems such as TrueCrypt and SEDs. To support FDE (where the OS is also encrypted), as in TrueCrypt, Gracewipe needs to work in the pre-OS stage. However, no ready-made TPM interfacing support is available. We

[2]Projects that are based on the TrueCrypt codebase or related to TrueCrypt can also be used/adapted with Gracewipe; e.g., TCnext (`http://truecrypt.ch`), CipherShed (`https://ciphershed.org`), VeraCrypt (`https://veracrypt.codeplex.com`).

have to construct TPM protocol messages on our own. Furthermore, we primarily base Gracewipe on TrueCrypt as 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. However, our Windows-based prototypes require a few heuristic changes specific to the versions of tboot [134] and Windows. This is due to Intel TXT's incompatibility with real-mode (switching from protected to real-mode is required by Windows boot) and Windows' unawareness of TXT. Booting a Linux-based OS after Gracewipe would have been easier to implement (we also managed to do so), but that would have less utility than the 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., [272]; 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 [228].

Authentication schemes under duress have been explored in recent proposals, e.g., [106, 38]. 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. SED-based Gracewipe is discussed elsewhere [311].

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. To the best of our knowledge, Gracewipe is the first project to enable running a fully-functional Windows OS at the end of a trusted execution session (Intel TXT).

5. We also analyze and compare several schemes for triggering password-based deletion, with considerations respectively on plausibility, security, and usability; some of these schemes are adapted from Clark and Hengartner [57]. We also discuss the implementation of some selected schemes. We label these schemes as Gracewipe-XD (Gracewipe Extended Deletion). Users may choose a scheme suitable to their threat model.

2.2 Goals and Threat Model

Gracewipe leverages several existing tools and mechanisms, such as multiboot [91], chainloading,[3] tboot [134], and TrueCrypt. We assume the reader is familiar with these techniques (for a brief introduction, see Appendix A).

2.2.1 Goals and terminology

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.

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

(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 the software-based FDE with support for plausible deniability (termed as *PDE-FDE*, e.g., TrueCrypt under Windows). A *decoy system* refers to the one containing non-confidential data for everyday entertainment and insensitive work. We inherit this notion largely from TrueCrypt with its original purpose. 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. [65]), e.g., browsing files by mounting the partition from the decoy system, or accessing shared resources such as the same remote server. *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* (see Section 2.5), but to simplify discussion, we consider only one here. *KN* and *KH* are stored/sealed in TPM NVRAM, which can be retrieved using the corresponding password, only within the verified Gracewipe environment. We use hidden/protected/confidential data interchangeably.

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 [134]. 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., via a TPM quote. 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 sealing, 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 (otherwise unsealing would fail). 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 by-product 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.

2.2.2 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. [228]). He is diligent enough to verify the TPM quote when key deletion occurs, and then (*optimistically*) stop punishing the victim, 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 a TrueCrypt 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 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 [108] to retrieve KH; such attacks are excluded, but see also TRESOR [194].

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 [234, 156], but Gracewipe can be extended with existing solutions against such attacks (e.g., [195]).

5. We exclude inadvertent leakage of secrets/passwords from human memory via side-channel attacks, e.g., the EEG-based *subliminal probing* [84]; see Bonaci et al. [39] for counter-measures. We also exclude *truth-serum* [293] induced attacks; effectiveness of such drugs is also strongly doubted (cf. [237]).

6. Gracewipe facilitates secure key deletion, but relies on FDE-based schemes for data confidentiality. For our prototypes, we assume TrueCrypt adequately protects user data and is free of backdoors.

7. We assume the size of *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[4]). 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 trusted measurements (hence only genuine Gracewipe unseals the keys); past attacks [297, 299] on TXT include exploiting the CPU's SMM (System Management Mode) to intercept TXT execution. SMM attacks can be addressed by Intel SMI transfer monitor (STM [133], for further reasons why SMM attacks do not pose a

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

threat, see Section 5.6 and Section 6.7). We also assume that hardware-based debuggers cannot compromise Intel TXT. We could not locate any documentation from Intel in this regard.[5] As documented [15], AMD's SVM disables hardware debugging.

2.3 Gracewipe Design

In this section, we expand the basic design as outlined in Section 2.1. 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. The FDE-only version (e.g., based on SED, not discussed here) is simpler than the PDE-FDE (TrueCrypt) design, e.g., no decoy volume and no chainloading are needed. 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.

2.3.1 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, and Windows bootloader. 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 with good entropy and configured as TrueCrypt "passwords". Each valid user password (including any *PD*) will unlock a corresponding key in TPM NVRAM for a specific purpose. See Fig. 2 for Gracewipe components.

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 the user password, and its behavior is determined by the entered password (or more precisely, by the data retrieved from TPM NVRAM with that password). Namely, if the retrieved data contains only a regular key (KH/KN),

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

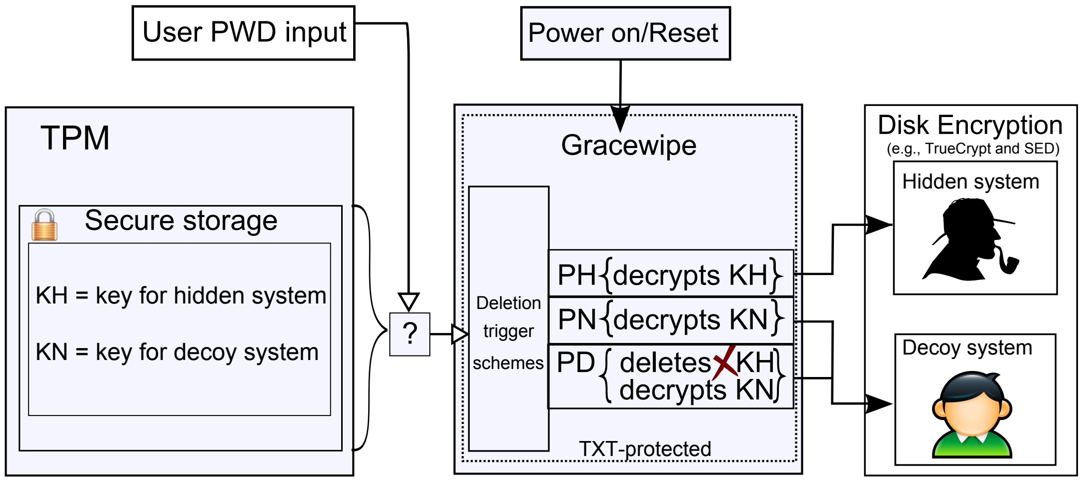


Figure 2: A generalized representation of Gracewipe. PN = password for the decoy system; PH = password for the hidden system; PD = deletion password. PH unlocks KH, the key that decrypts the hidden system; PN unlocks KN, the key that decrypts the decoy system; PD deletes KH from TPM NVRAM, and may optionally unlock 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., no password prompt), and boot the corresponding encrypted system. Note that each user password corresponds to one TPM NVRAM index. Each index contain an indicator value (byte-long; 'P' for PH, 'K' for KN and 'D' for PD) concatenated with a proper decryption key (KH/KN, or for the deletion index either some random data or KN). Both the indicator and key values are protected by TPM sealing, and an attacker cannot exploit the indicator values (see Section 2.8 under item (b)).

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 [8]) 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 (see Appendix A), failure of which causes several troubles including system crash (see Section 2.4.4).

2.3.2 Execution steps

Gracewipe's execution flow is outlined in Fig. 2. It involves the following steps: (1) The system BIOS loads GRUB, which then loads tboot and other modules including the wiper and ACM SINIT module (cf. Appendix A). (2) Tboot performs necessary checks and calculates/matches the platform measurement with the values stored in the TPM (halts on failure). (3) The wiper prompts the user for a password, and uses the entered password to access TPM indices where we store KH/KN one by one (the extended schemes are different, see Section 2.5). Note that we merely provide the entered password to the TPM specifying an index to unlock and receive the result; no decryption happens in the Gracewipe code. If no index is accessible, an invalid password is received (resulting reboot/halt). (4) As part of the TPM accessed data, an indicator field shows if the entered password is PH, PN or PD. Upon reception of PD, the wiper immediately erases KH from TPM, and performs a quote to display the attestation string on the screen. (5) In the case of PH or PN, the wiper copies the decryption key (i.e., TrueCrypt "password") to a memory location to be retrieved later by TrueCrypt. (6) The wiper switches the system back to real-mode, reinitializes it by mimicking what is done by BIOS at boot time. (7) The TrueCrypt MBR is executed, which proceeds as if the wiper-copied "password" is typed by the user and loads the system as usual (the decoy or the hidden one).

2.3.3 Sealing in NVRAM

TPM specifications mandate mechanisms against guessing attacks on password-protected NVRAM data (e.g., only a few passwords may be consecutively tested within a specific period of time). 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 2.4.4). 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 the TPM's data sealing technique, so that if an altered software stack (i.e., anything other than the genuine copy of 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, when Gracewipe is active, each guess may unlock the hidden/decoy data, or trigger key deletion.

Instead of directly sealing the keys into NVRAM, we make use of the access-control-based PCR binding to achieve the same goal. When an NVRAM index is defined, selected PCRs are specified as the access requirement in addition to the user password (authdata). The stored key can be accessed only if both the password and the PCR values (correct environment) are satisfied. This design choice prevents offline guessing of user passwords protecting the sealed keys, as opposed to the following construction: use *PN* as the authdata secret to protect *KN* stored in NVRAM in its sealed form. Without the correct environment, *KN* cannot be unsealed. 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; the attacker can then use the (guessed) valid passwords in Gracewipe to unlock corresponding keys. With our current construction, TPM will output the same failure message, if either PCR values or passwords are incorrect.

2.3.4 Password management

Under the strong adversarial model in Gracewipe, the user (e.g., security personnel) is expected to properly maintain the configured passwords, and if they are lost the recommended solution is redeployment; i.e., there must be no *password recovery.* The data or keys protected under Gracewipe must not be backed-up in any Internet-accessible storage under any circumstances; this will enable easy coercion even after a successful local deletion. However, password update can still be supported; we propose a simple mechanism below (can be extended to accommodate other schemes in Section 2.5).

At the *same* password prompt where the user normally unlocks the system at boot-time, password update mode can be triggered by using a special key sequence (e.g., "Ctrl+Enter" instead of the regular "Enter" key). The received password is first handled by the deployed deletion triggering scheme, i.e., deletion will be initiated if *PD* is typed. After the password update mode is entered (i.e., deletion is not triggered), the user is prompted to enter an existing password to be changed ($PH/PN/PD$), and then type the new password twice. Gracewipe will try to access indices one by one until a success and replace the protecting password with the new one. Note that in order not to reveal any further information in the password update mode, no explicit feedback is provided, i.e., merely “Update process is done!” is displayed regardless of a successful update or failure. Also, a random delay can be added so that timing characteristics do not help distinguish valid passwords. We have implemented this scheme.

2.4 Implementation with TrueCrypt

In this section, we summarize certain implementation considerations of Gracewipe specific to TrueCrypt. We also discuss several side effects resulting from our implementation choices and corresponding workarounds. The implementation effort mainly involves the wiper (Gracewipe core), TrueCrypt modifications, and a few configuration steps to make the components work together.

Our choice of Windows is largely in consideration of its prevalence, and TrueCrypt FDE's availability only under Windows. We have also successfully booted up Linux via Gracewipe with fewer changes compared to Windows. Gracewipe in itself is a boot-time tool, which does not run along-side the user OS. 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.

2.4.1 Implementing the wiper

Approximately, the wiper has 400 lines of code in assembly, 700 lines in C and 1300 lines of reused code from tboot. As discussed in Section 2.3.1, the TPM must be accessed by the wiper, which runs at an early stage of system boot, i.e., right after GRUB and tboot. Due to lack of TPM access support at boot time (at least for NVRAM storage as in our case), we must handle the communications between TPM and the wiper, and implement a subset of the TCG software stack [8].

We choose to communicate with with TPM through the MMIO interface for future compatibility and consistency with tboot. After being able to send commands to the TPM via MMIO, we implement the authdata-protected NVRAM access functions (for secure storage of Gracewipe keys). Due to inadequate documentation, we had to reverse-engineer the related functions in the TCG stack for our implementation.

Moreover, for verifying the correct execution of Gracewipe (and thus the deletion of *KH*), the wiper must be able to perform 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). We allow the verifier (adversary) to enter a string of his choice as the nonce and store the quote value in an unprotected NVRAM index as well as displaying it on the screen.

At the end, we have developed the following TPM functions: *tpm_nv_read_value_auth()*, *tpm_nv_write_value_auth()*, *tpm_loadkey2()*, *tpm_nv_define_release()*, *tpm_quote2()*; we reuse most other functions from tboot.

2.4.2 Adapting TrueCrypt

To make TrueCrypt aware of Gracewipe, we require some changes in its source code. We keep such changes to a minimum for easier maintenance and deployment. They are mostly in BootLoader.com.gz (BootMain.cpp), and a few minor changes in BootSector.bin (BootSector.asm). In BootSector.bin, changes are for the modified version of BootLoader.com.gz to pass the original integrity check (CRC32). 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.

2.4.3 Orchestrating components

Additional deployment-time efforts are needed to make Gracewipe components work seamlessly. Such efforts include configuring GRUB (with a Gracewipe-customized menu.lst), initializing TPM, and installing TrueCrypt. Also, as the integrity of the Gracewipe environment relies on tboot's policy enforcement, it is critical to ensure the proper setup of the MLE policy and tboot's custom policy.

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 user is prompted for the three passwords (PN, PH and PD) of her choice and the two keys (KN and KH) generated in the OS. The wiper seals the two keys with the current environment measurements into NVRAM indices with the three passwords. The indices can be user-configured to avoid conflicts with other use of the TPM; however, the order of password to index assignment is randomized to avoid any possibility of interference with the deletion process using time differences (cf. Section 2.6). Note that different unlocking schemes may involve different procedures, see Section 2.5. In the end, the wiper toggles the mode value for the next time to run in the normal mode.

2.4.4 Windows and TPM issues

Disabling TXT DMA protection for Windows. During implementation, we faced several issues related to Windows, mostly due to Windows being unaware of protections enabled by Intel TXT. Unlike Linux, Windows also cannot be adapted for TXT as it is closed-sourced. Below, we discuss a DMA problem and its solution.

TXT protects the execution environment from unauthorized code. The I/O protection (i.e., no peripheral on the bus other than the measured code can access protected memory regions) is enforced in hardware by IOMMU in collaboration with the chipset. By default, it is left enabled, when TXT is torn down with instruction GETSEC[SEXIT] (see [136]); the guest OS is supposed to be aware of the DMA protection, and perform any additional cleanup operations.

In addition to the platform-specific fixed DMA Protected Range (DPR, usually 3MB in size), custom Protected Memory Regions (PMRs [137]) can also be specified to SINIT for DMA protection. SINIT guarantees that the measured program (in our case: tboot and the wiper) is covered by either the DPR or one of the PMRs; otherwise, the program cannot be started.

The consequence of the aforementioned DMA protection depends on the OS taking control after TXT exit; e.g., if the OS is aware of IOMMU, the protected ranges can be avoided or remapped, or IOMMU should be disabled. For Gracewipe with Windows, IOMMU must be disabled due to Windows' unawareness. There is an IOMMU register *DMAR_PMEN_REG*, and by setting its *DMA_PMEN_EPM* bit to 0, the IOMMU PMR can be disabled.

If the PMRs are left enabled to Windows, as in the initial implementation of

Gracewipe [311], the system will behave unpredictably, when memory access hits the protected regions. For example, right after Windows switches to the hard disk device driver from BIOS calls, the booting process fails with UNMOUNT-ABLE_BOOT_VOLUME (0x000000ED). The reason code of 0xC000014F indicates a disk hardware problem, which is incorrect as we could boot Windows without tboot. Initially, we changed the ATA channel to use the "PIO" mode instead of "Ultra DMA Mode 5", and Windows booted successfully, but with disabled DMA for disk operations (i.e., degraded disk performance). Disabling IOMMU PMRs solved this issue without affecting disk performance.

TPM deadlock. Here, we discuss an issue originating from our somewhat unusual way of leveraging a TPM. By design, TPM NVRAM is intended to provide protected access to confidential data. Such protection, especially with authdata access, is unsuitable to be used as a general purpose decryption oracle: 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.

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. Therefore, TPM actually counts each failed attempt as a violation and may enter a lockout state (released by an explicit reset or timeout); for details, see under *dictionary attack considerations* in the TPM specification [277]. We relied on TPM_ResetLockValue and time-out during our development.

Note that this limitation is mitigated by the DL-distance and pattern-based Gracewipe-XD schemes (Section 2.5), where the secret data to compare with is unsealed from NVRAM and the user-typed password is not used as authdata (thus no failed authentication to access NVRAM).

2.5 Extended Unlocking Schemes

In the basic version of Gracewipe, only one or a few predefined *PDs* are allowed. In this section, we discuss password-based deletion triggers to avoid limitations of the basic Gracewipe design (see below). We adapt some existing schemes and explore new ones, and implement the most promising variants (called Gracewipe-XD).

Limitations of few deletion passwords. (1) The adversary's risk in guessing passwords is rather low. One or a few deletion passwords represent a very small fraction of a large set of possible passwords, e.g., millions in the case of brute-forcing, or at least hundreds, in the case of a small dictionary of most frequent passwords. (2) In terms of plausibility, the user is left with too few choices when coerced to provide a list of valid passwords; passwords other than PN, PH or PD will unlock no system nor trigger the deletion, and generate an error message. The adversary may choose to punish the user for any invalid password. With three valid passwords, the attacker's chance of guessing PH is at least $\frac{1}{3}$ (although the risk of triggering deletion is also the same).

2.5.1 Existing panic password schemes

We summarize several existing panic password proposals [57] (primarily for Internet voting), and analyze their applicability in our threat model (client-only).

2P. The user has a regular password (in our case, PH) and a panic password (in our case, PD). The 2P scheme applies to situations where authentication reactions are *unrecoverable*; e.g., if PD is entered in Gracewipe, further adversary actions cannot help data recovery, as the target key KH is now permanently inaccessible. However, if the attacker can extract both passwords from the victim, the chance of triggering deletion/panic is $\frac{1}{2}$; for 3P, the chance is: $\frac{2}{3}$, and so on. Thus 2P resembles the mechanism in the basic Gracewipe, which has both the aforementioned limitations.

2P-lock. When the reactions are *recoverable*, i.e., after PD is entered, knowing PH is still useful for the adversary (unlike Gracewipe), the adversary may continue guessing until he finds PH, but is bound to a time limit to end coercion (e.g., for escaping). In this case, a lockout mechanism can be applied to allow only one attempt, and make the two passwords indistinguishable. If a valid password is entered, the system always behaves the same (the panic passwords would signal coercion silently); then within a specified period, if a second valid but different password is used, the system locks out for a period longer than the adversary's time limit. However, 2P-lock is ill-suited for Gracewipe as there is no trusted clock to enforce the lockout (the BIOS clock can be easily reset).

P-Compliment. This scheme is applicable against *persistent* adversaries (i.e., reactions are recoverable and no time limit for coercion). Instead of having a limited number of panic passwords, any invalid password (i.e., other than the correct one) can be considered a panic password. This simple rule will result in user typos to trigger unwanted panic/deletion. To alleviate, passwords that are close to the correct one (i.e., easily mistyped) can be considered invalid (instead of panic), and thus the password space is divided into three sets based on edit distance: the correct password, invalid passwords and panic passwords. The user can now provide a large number of panic passwords, and typos are tolerated. Note that for a persistent adversary, it is assumed that there is no fatal consequence when a panic password is used (e.g., as in the case of online voting, the account is locked for a while). Thus, if the panic password and invalid password spaces are not well mixed, the adversary can try to approximate the boundary between them with multiple attempts. In Section 2.5.3, we discuss a Gracewipe variant derived from P-Complement.

5-Dictionary. For better memorizability, a user can choose 5 words from a standard dictionary, using a password space division similar to P-Compliment: any 5 words in the dictionary other than the user-chosen ones are considered panic passwords; any other strings are invalid. This scheme tolerates user typos and provides a large set of panic passwords. However, the number of panic passwords (P_5^n, for a dictionary of n words) could still be much smaller than the invalid ones. We propose an adapted version of this scheme in Section 2.5.4.

5-Click. For image-based schemes, any valid region in an image (excluding parts used for the correct login) can be used to communicate the panic situation. As Gracewipe relies only on text passwords, we exclude such schemes.

2.5.2 Counter-based deletion trigger

We design a counter-based mechanism by adapting 2P-lock to limit adversarial iterative attempts without increasing the risk of accidental deletion (by user). Reaching the limit of failed attempts is used to trigger deletion, instead of locking out the system. Below, we provide the design and implementation of this adapted scheme.

Design. We keep the functionality of $PD/PH/PN$ as in the basic Gracewipe design, i.e., entering PD will still initiate an immediate deletion. In addition, we now count the number of invalid attempts (i.e., entry of passwords other than $PD/PH/PN$), and use the counter value as a deletion trigger when a user-defined preset threshold (e.g., 10) is reached. The counter must be integrity-protected—i.e., can be updated only by the correct Gracewipe environment.

An important consideration is when to reset the counter value. Because a legitimate user may also mistype sometimes, and as such errors accumulate the deletion will be triggered eventually. We consider two options for resetting the counter value: (1) *Timeout.* It is mainly used in online authentication systems. However, without a reliable clock source, it is inapplicable to Gracewipe. (2) *Successful login.* Assuming that typos are relatively infrequent, a legitimate user will successfully login before the threshold is reached. We use such login to reset the counter. Note that only the entry of a valid PH is considered a successful login (but not PN, which can be revealed to the adversary when needed).

Implementation. This scheme is implemented by simply adding checks to the code where the entered password has failed to unlock any indices and where KH is successfully unlocked. The counter value is sealed in a separate NVRAM index with the environment measurements. We also bind the measurements to both read and write access of this index so that a modified program cannot even read it, not to mention updating. At deployment time, the counter is initialized to 0. A user-specific trigger value is secured the same way as the counter (i.e., no access outside the correct Gracewipe environment). If the adversary tries to reset either of them by re-initiating Gracewipe deployment, he will have KH erased first before both values are reset. The logic is as follows: Any invalid passwords will increment the counter; entry of PN does not affect the counter; entry of PH will reset it to 0. Whenever the counter value is equal to the trigger value, deletion is initiated.

2.5.3 Edit-distance-based password scheme

The counter-based deletion trigger can severely limit guessing attempts. However, if the user is forced to reveal all *valid* passwords, the attacker's guessing success rate will increase, due to the limited number of valid passwords (PN and few PDs). We design the following variant to counter both threats.

Design. Following the P-Compliment scheme, we use *edit distance* to divide the password space. Instead of predefined $PD/PH/PN$, we develop a rule to determine which category the password falls into during authentication. There will be no invalid passwords any more, and actions are taken silently (unlock the hidden system, decoy system, or trigger deletion).

We must balance between the risk of user typos and the coverage of passwords the adversary may guess. To measure the closeness between two passwords, we use *edit distance*: the number of operations (edits) required to convert one string to another. By centering to user-defined PH, we can treat the rest of the password space according to edit distance. The farther a password is from PH, the more likely it is entered by the adversary, and vice versa.

There are different variants of edit distance metrics, mostly depending on the

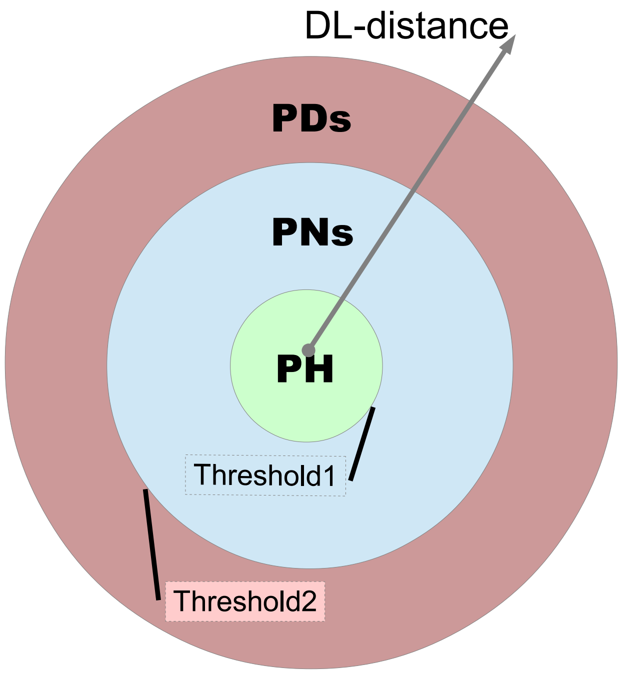


Figure 3: Dividing password space with DL-distance

types of allowed edit operations (e.g., insertion, deletion, substitution, and transposition). These metrics usually provide similar performance in distinguishing strings but with various computation complexity (less critical for Gracewipe). We use the Damerau-Levenshtein distance [30] (DL-distance), which considers only the following operations: insertion (one character), deletion (one character), substitution (one character) and transposition (two adjacent characters).

The choice of edit distance metrics may also involve other considerations. For example, we can take into account cognitive aspects (e.g., words with interchangeable meanings or user-specific typing habits), and device/physical aspects (e.g., common keyboard layouts). Especially, the CapsLock key must be checked, which can lead to large edit distance even when the correct password characters are typed, and convert all characters into lower case before processing. If such aspects are parametrized, a training process can also be introduced to customize Gracewipe-XD for a specific user.

If we denote the entered password as PX, the overall logic is as follows (see Fig. 3): if *DL-distance (PX, PH)* is less than or equal to *Threshold1*, *PH* is received; if *DL-distance (PX, PH)* is greater than *Threshold1* but less than or equal to *Threshold2*, then *PN* is received; otherwise, *PD* is received, which triggers the deletion process (including quote generation). We convert both *PX* and *PH* into lowercase for the DL-distance calculation (to avoid accidental CapsLock on status).

Note that, using DL-distance, multiple *PHs* can be allowed to access the hidden volume (e.g., by allowing *Threshold1* to be greater than 0). However, the usability benefit may be insignificant, as the range has to be centered to one *PH*, and thus forgetting *PH* may also indicate not remembering those that are only one or two characters different. On the other hand, allowing multiple *PHs* will increase the adversary's guessing probability (*PHs* cover more in the guessable space). At the end, we kept *Threshold1* to 0, i.e., a single *PH* is used.

In contrast to P-Complement [57], password spaces for *PN* and *PD* may not need to be well-mixed in our variant. However, we still re-examine any potential security consequence of our choice as follows:

1. Only a single *PD* will suffice to make the target data inaccessible. Thus approximating *PH* with multiple provided *PDs* or *PNs* is infeasible due to the high risk of deletion.
2. The adversary may extract non-*PH* words (i.e., *PNs* and *PDs*) from a victim, before launching a guessing attack through the Gracewipe interface. Such seemingly non-secret information may help the adversary to identify boundaries between password spaces (cf. [52]). In Gracewipe-XD, edit distance is omnidirectional (unlike the simple depiction in Figure 3, where values are centralized to one *PH* on the same plane), and also parametrized by character sets, maximum length etc. We thus argue that the attacker cannot easily identify a trend/pattern pointing to *PH*.

Implementation. A significant change in the edit-distance-based scheme is that we must store *PH* in an NVRAM index for the DL-distance calculation. We seal *PH* with the Gracewipe environment measurements. At evaluation time, *PH* must be loaded to the system memory (with the correct Gracewipe environment), which may provide a chance to launch cold boot attacks [108]; note that DMA attacks are prevented by TXT. However, in our implementation, *PH* stays in memory for a short period of time—*PH* is unlocked after the candidate password is entered, and erased immediately after the DL-distance calculation (on average, 3-4 milliseconds in our test environment, for 8-character passwords). In this case, we argue that timing the cold boot attack to extract *PH* would be infeasible; for complexities of such attacks, see e.g., [102, 48]. Alternatively, *PH* can be copied directly to CPU registers to bypass memory attacks (cf. TRESOR [194]).

2.5.4 Other possible schemes

We have also explored more possibilities for the password schemes and deletion triggers. They can be further examined and implemented for specific use-cases.

Pattern-based deletion passwords. The user is allowed to define her own customized pattern for *PDs*, e.g., using regular expressions. Any string that does not match such pattern will be treated as *PNs* or invalid. This may provide better memorizability while allowing a large number of *PDs* (users must remember the pattern, but not the actual *PDs*). A foreseeable downside is that the adversary may learn the pattern (e.g., through text-mining) from passwords extracted from the victim, and then avoid passwords of such pattern when guessing. Also, this scheme does not address mistyping.

Misremember-tolerant deletion passwords. A user may accidentally enter a deletion password (e.g., due to stress, misremember) and realize the mistake instantly. In the basic Gracewipe, this would be fatal, as KH will be deleted immediately after receiving a PD. To reduce such accidental dental deletion, we adapt the counter-based scheme as follows.

For any entered PD, before triggering deletion, a counter value is checked; if it is already 1 (or any custom threshold), deletion is triggered as usual; otherwise, the counter value is incremented and the entered PD is just treated as PN. The counter value is initialized to 0 during deployment. A correct entry of PH will reset the counter. Thus, at the cost of allowing the adversary to try an additional password, accidental deletion can be avoided.

Small-dictionary scheme. The use of a built-in dictionary may serve as an alternative for tolerating user typos. The assumption here is that a mistyped word is more likely to be absent in the dictionary (but not always, e.g., *race* and *face*). Multiple user-chosen words (e.g., 5) form a *passphrase*. We adapt the 5-Dictionary scheme [57] to incorporate the following considerations: (1) We would like to follow the principle in the edit-distance-based scheme, i.e., the number of *PDs* is arbitrarily large to make sure that the probability of triggering deletion is rather high, meanwhile with *PNs* serving as a buffer zone to accommodate typos. (2) To increase the number of *PDs*, we can treat the invalid passwords in 5-Dictionary as *PDs*. However, most such invalid passwords are formed by non-dictionary words, and therefore can be easily entered by mistyping. Thus, we would like to design an adapted scheme that triggers deletion with non-dictionary words but tolerates mistyping.

Instead of using large natural dictionaries (e.g., English vocabulary), the user defines a custom dictionary that contains her memorizable strings (words or non-words). The size of the custom dictionary is relatively small that fits in TPM NVRAM, e.g., 50–100 words; such a small dictionary ensures that at a very high probability a random word falls outside the dictionary and may eventually lead to deletion. However, the custom dictionary must be both confidentiality and integrity protected, unlike the public standard dictionary in 5-Dictionary.

A PH consists of three (or more) segments, each picked from the custom dictionary. If *none* of the three segments of the typed password belong to the dictionary, the password is considered as PD, and deletion is triggered. If *all* of the three segments of the typed password belong to the dictionary, the password is considered as PH. Otherwise, the typed password is treated as PN. We assume that the probability of mistyping all three segments is low, reducing the chance of inadvertent deletion trigger by mistyping or even misremembering (see below). In contrast, without the knowledge of the custom dictionary, the attacker's guessable password space is as large as 5-Dictionary.

This scheme also partially addresses accidental deletion due to misremembering. If the user chooses to include all potential words/strings she may use in her passphrases for other accounts, even if she misremembers, still one or more segments of the misused passphrase fall in the custom dictionary and thus will be only treated as a PN.

Another benefit of the small-dictionary scheme is that the invalid password space and the deletion password space can be better mixed (for plausibility). Also, the hidden password is more difficult to approximate from extracted passwords (i.e., not reflecting constant space "away" from other passwords), because the custom dictionary diffuses candidate invalid/deletion passwords from the hidden one. Also, the custom dictionary being small allows it to be stored securely in TPM NVRAM, which removes access to the sealed dictionary without the correct Gracewipe environment (as opposed to sealed data stored on disk).

2.6 Performance Overhead

By design, Gracewipe merely replaces the user authentication part of an existing FDE scheme at boot-time, and does not interfere with the runtime performance of the OS or applications. In this section, the boot-time overhead of Gracewipe in normal operations is evaluated to demonstrate Gracewipe's practicality. We exclude the one-time deployment phase and quote generation after deletion as they occur only in special cases, and introduce only a delay of less than a second (for operations like loading keys in TPM and quoting).

Methodology of measurement. Unlike Linux kernel's *do_gettimeofday()*, we lack a reliable clock source in the pre-OS environment. We use CPU's Time Stamp Counter (TSC) via the *rdtsc* instruction. TSC stores the total number of machine cycles since the processor reset. A divisor (denoted as *N.TSC* hereafter) can be calculated so that TSC/N.TSC produces the total number of elapsed milliseconds (instead of machine cycles). This process is called TSC calibration, where the hardware 8253 Programmable Interval Timer (PIT) is programmed to produce a millisecond-long interval and the TSC value difference before and after is N.TSC. We do not try more recent alternatives (e.g., the *invariant TSC* feature in recent CPUs) as the original calibration-based approach has been tested and used in well-established projects, e.g., tboot, Linux kernel and GRUB2.

In our test machine, we get N.TSC values roughly between 3494388 and 3504892 across multiple calibration attempts (error ± 0.003ms). Instead of taking an average of the calibrated values, we use the actual N.TSC value right before each measurement to calculate the elapsed milliseconds, as per-measurement calibration reflects real-time characteristics. We perform each measurement 15 times and use the *R* project to calculate statistics.

Tboot. The choice of using tboot (as opposed to dealing with TXT with custom code) is justified by the fact that it has undergone sufficient public/expert scrutiny and thus is more reliable especially for the crucial TXT-handling logic. It also introduces an apparently acceptable level of latency.

By default, tboot enables debugging (to VGA, serial port or memory), which slows it down significantly, taking 30 seconds or more to complete. We disable debugging by passing necessary arguments. Our 15 independent measurements demonstrate coherent execution times: mean 1611.20ms, median 1611.96ms, standard deviation (sd) 6.08.

The basic Gracewipe. As the basic design tries to unlock the three defined indices in sequence until a success, we separately time the three cases: (1) Success at the first index (including deletion, if it stores *PD*): mean 646.83ms, median 645.98ms, and sd 2.81. (2) Success at the first index (excluding deletion): mean 560.21ms, median 558.90ms, sd 3.78. (3) Success at the second index: mean 616.81ms, median 617.16ms, and sd 3.47. (4) Success at the third index: mean 746.97ms, median 743.40ms, and sd 10.61.

DL-distance-based Gracewipe-XD. User response time, such as password typing, is excluded from our measurement, since it is also needed for regular FDE. We hard-code the user input corresponding to each scenario for measuring only the execution time. We count from the point where control is taken over from tboot to the point where TrueCrypt is about to be loaded. Our attempts to measure the DL-distance-based scheme result in an average of 591.16ms, median of 589.71ms and standard deviation of 7.74.

Promptness of deletion. We also measure the duration of the deletion operation (releasing and overwriting an NVRAM index). Over the 15 attempts, we found that deletion takes about 87ms (mean 86.62 and median 87.05), with a very small deviation (sd 1.39), supporting our claim for a quick deletion.

In summary, the overall latency introduced by Gracewipe is between 2 and 2.5 seconds.

2.7 Generalized Workflow and Comparison

Different password and deletion schemes provide flexibility, and can be used in different application scenarios. However, the core Gracewipe features are always provided: plausible user compliance, undetectable deletion trigger, risky guessing and verifiability. In this section, we provide a generalized workflow for Gracewipe variants and compare them in terms of security benefits and ease of use.

Generalized workflow. At deployment time, in addition to setting up secrets (KH, KN), according to the actual variant of Gracewipe in use, the user defines corresponding parameters (thresholds, rules, or a custom dictionary). Each time the system is booted into Gracewipe in a TXT session (loaded by GRUB and tboot). The user is prompted for a password. The difference of Gracewipe variants is reflected in the evaluation of the entered password, which eventually produces an outcome (KN, KH, or deletion). Thereafter, the system is unlocked, or a quote is generated for later verification if deletion is triggered.

In normal operations, the user chooses to enter PN or PH, which unlocks KN for the decoy system or KH for the hidden system, respectively. If she mistypes or misremembers the password, the Gracewipe variant in use determines to what extent she can avoid triggering the deletion. When the user is coerced, she can be forced to provide a list of valid password, the number of which depends on the password scheme used.

Comparison. Table 1 summarizes several security and ease-of-use features of different deletion triggering/password schemes.

Large Deletion Space denotes the availability of many plausible deletion passwords that the user can reveal to pretend compliance. The basic Gracewipe only supports one or a few deletion passwords; the counter-based and misremember-tolerant variants are used with other schemes, and do not offer this property alone.

High Guessing Risk represents a relatively high probability of triggering deletion with a guessed password. For the basic Gracewipe, it is just a few out of a large password space. Other schemes offer this feature by either rate-limiting (the counter-based one), or having a large deletion password space.

Typo-tolerant means the scheme tolerates user typos. This feature can be achieved either by using static passwords (assuming they are not close in terms of edit distance), or carefully managing the password distribution (e.g., greater distance between *PDs* and *PHs*). For the pattern-based scheme, typo tolerance is determined by the defined pattern.

Reduced Accidental Deletion denotes reduced risk from accidental deletion, e.g., mistakenly typing *PD* instead of *PH*. Schemes with fixed *PD* (s), e.g., the basic Gracewipe, obviously do not offer this feature. *PDs* are not predefined in the DL-distance and pattern-based schemes; however, misremembering *PH* or the pattern can still trigger accidental deletion. Small-dictionary partially tolerates misremembering, when at least one segment of a misremembered passphrase is found in the dictionary. The misremember-tolerant add-on can reduce the risk of accidental deletion in any variant.

Non-RAM Secrets indicates that plaintext e.g., *PH*, custom dictionary, are not exposed to the system memory. Although the feasibility of cold boot attacks is

	Deletion triggers/password schemes	Large Deletion Space	High Guessing Risk	Typo-tolerant	Reduced Accidental Deletion	Non-RAM Secrets
Gracewipe	Single/few deletion passwords			●		●
Gracewipe-XD	Counter-based (add-on)	—	●	—		—
	DL-distance-based	●	●	●		
	Pattern-based	●	●	○		
	Misremember-tolerant (add-on)	—	—	●	●	—
	Small-dictionary	●	●	●	○	○

Table 1: Comparison of Gracewipe password schemes. Keys: ● (offers the feature); ○ (partially offers the feature);
— (scheme-dependent, "add-ons" may not be evaluated alone for certain properties); blank (lacks the feature).

arguably low (recall that DMA attacks are already prevented by TXT), due to the very-short in-RAM password exposure period, avoiding plaintext secrets in memory is a better design choice. The basic Gracewipe's password evaluation is entirely TPM-bound. In contrast, the DL-distance-based scheme loads *PH* and the pattern-based scheme loads the pattern (e.g., a regex) in memory at evaluation-time. The small-dictionary scheme can partially avoid loading secrets in memory, if the custom dictionary is unlocked chunk by chunk to be matched with the typed passphrase (at the cost of performance), so that at a specific time only a small portion of the dictionary is in RAM.

2.8 Security Analysis

In this section, we 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. For all Gracewipe variants we exclude known physical attacks on TPM chips, as either they could have been patched by the vendor or the user is motivated to choose a model/situation where such attacks do not apply due to their being ad-hoc, e.g., TPM integrated in the SuperIO chip; we briefly discuss several (historical) attacks on TPM elsewhere [311].

(a) Evil-maid attacks. In 2009, Rutkowska demonstrated the possibility of an evil-maid attack [234] (also termed as *bootkit* by Kleissner [156] 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 2.3). 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., MARK [96], can also be integrated with Gracewipe.

(b) Undetectable deletion trigger. As discussed under “Sealing in NVRAM” in Section 2.3, 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 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).

(c) Quoting for detecting spoofed environment. Currently, we generate a quote only 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 detect any modifications to Gracewipe. In both secure deletion and normal operation, the selection of a proper nonce is required. We currently support both arbitrary user-chosen strings and timestamps as nonces. Nevertheless, the use of a timestamp is susceptible to a pre-play attack, where one party can approximately predict 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.

(d) 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.

(e) 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 when passing them to configure TrueCrypt. 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.

2.9 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 [226]. 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 [228]. In a blog post, Rescorla [228] 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 [88]) 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 [228], 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.

Secure deletion survey [226]. 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 a significantly long time with 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 and MARK [195]. Müller et al. propose a protocol for mutual authentication between humans and computers, arguing that a forged bootloader can trick the user to leak her password (cf. [234, 156]). 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 [280]. 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. Each time a new message is set by the user to maintain the freshness, hence its name *monce*. Its improved version MARK uses a special USB device as secure storage to bootstrap the process credibly. Gracewipe may be extended with such techniques to defeat evil-maid attacks.

DriveCrypt Plus Pack [245]. DCPP can be considered the closest prior art to Gracewipe. It is a closed-source FDE counterpart of TrueCrypt, with the support for deniable storage (hidden volumes), 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.

2.10 Concluding Remarks

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 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`.

Chapter 3

Extending Gracewipe to Network-based Environments

Based on Gracewipe, we shift our focus from coercion to remote data erase. We have noticed the paradigm of secure remote deletion is not positioned with the unconventional threats considered. Especially, motivated by malicious insiders (e.g., a privileged employee, receiving the lay-off notice, may try to steal confidential data from servers they manage)[177], we would like to look further to see what can be done to improve secure remote deletion.

3.1 Introduction

The need for secure remote deletion of user data (also referred to as remote erase or remote wipe) has made it the de-facto standard for certain services/products in the industry. Examples include features available in cloud-storage client apps (e.g., [72] and [188]), shipped with the OS (e.g., [93] and [18]), and supported by firmware (e.g., Dell's Remote Data Delete service [67]), targeting both enterprise and consumer markets. Academic proposals also have been around for years (see [264] and [87]) as well as some patents (e.g., [240, 238]).

The purpose thereof is mainly to ensure data secrecy with lost or stolen devices where the owner has lost physical control. To ensure prompt effect, cryptographic deletion is usually used (i.e., encrypting data first and erasing only the encryption keys) instead of the time-consuming overwriting-based deletion.

However, the claimed security therein usually only considers authentication (although sometimes even mutual authentication), i.e., making sure that the device can only be erased by authorized persons and that the right device is being erased (no impersonation). When considering that the device can be potentially compromised or that the adversary has physical control over the device, the erase process might be interfered with and as a result the data would not be properly erased. Moreover, even without the presence of an adversary, device misconfiguration (changes after initial deployment) and untrusted provider employees may still pose an uncertainty to the correct erase operation.

Therefore, in certain scenarios, in addition to initiating the erase operation, verifiability is more crucial, i.e., the ability to tell whether the deletion is provably (depending on the threat model) successful or aborted. Simply checking the operation status returned from the target device is insufficient because if the software stack is compromised the returned status may be forged.

Repositioning secure remote deletion. We shift our purpose from increasing the likelihood of deletion success (e.g., in the case of lost or stolen devices) due to the difficulty stated above, to being able to tell whether the deletion has succeeded or not (cryptographically), hence trusted remote deletion.

Enriching secure remote deletion. We also demonstrate that the effectiveness of remote deletion is highly associated with the way the secret (to be deleted) is stored and used. So the trusted remote deletion must be integrated or collaborate with the underlying data protection mechanism (e.g., an FDE scheme).

Contributions.

1. We formalize various factors affecting the effectiveness and usefulness of remote secure erase by surveying existing academic/industrial solutions and propose user-verifiability to be of the utmost importance to remote secure erase solutions.

2. We accommodate the above factors and put forward a high-level design for an end-to-end verifiable remote data erase that can be later instantiated on different platforms.

3. Targeting PCs and enterprise workstations, we design and implement a proof-of-concept of the proposed framework, using Intel TXT with TPM and Intel AMT.

3.2 Threat Model and Assumptions

Goals. Our objective is to create a high-level framework for remote secure data erase whose verifiability does not rely on the intermediate parties (i.e., end-to-end), and which is integrated with how the secret is stored/used, so that the confirmation of success does indicate that the secret has been irreversibly deleted.

We argue that an effective remote solution does not simply stop at ensuring successful issuance of the wipe command and receipt of the acknowledgment.

Assumptions.

a) **Owned device.** The device where data erase happens belongs to the initiating user, i.e., we do not consider the case when a user tries to delete a file from a server owned by another party.

b) **Trust anchoring.** All hardware-based approaches have to rely on at least the manufacturer for correctly implementing the security functions (e.g., without a backdoor). That being said, we already minimize the number of trusted parties by excluding various service providers and other entities.

c) **Storage media.** We do not consider physical storage media in our discussion. For instance, magnetic media and flash storage demonstrate different characteristics in terms of the chance by which deleted data can be recovered (and if any, to what extent). This is in light of our consideration of only cryptographic deletion (cf. Section 3.3), where the deleted data is of very small size (e.g., a 128-bit key).

d) **Complete coverage.** Traces elsewhere left by the data to be erased are out of scope. If any, we assume whatever proves the ever-existence of erased data has already been included in the erase. Full disk encryption (FDE) is one of the examples that address this, hence leading to *full disk erase.*

e) **Confidentiality of utmost importance.** We prioritize data confidentiality over all other threats, such as denial-of-service attacks or monetary loss, so that the sensitive data never falls in the wrong hands.

f) **Correct initial deployment.** In addition to trusted parties, everything should be assumed to be correct at the time the device is first set up.

3.3 An Analysis and Status-quo of Remote Secure Erase

The main purpose of current remote erase solutions is to perform the erase operation on a target device, to which the user does not have physical access. The operation can usually be initiated from any computer system (e.g., web browser or client app on a cellphone, tablet or PC), as long as certain authentication is satisfied. Below, we analyze a few factors that should be considered when evaluating existing remote erase approaches.

Application scenarios. Today's computing devices tend to apply data encryption for persistent storage (e.g., flash or hard drive) as a common practice. This is especially true in our setting when confidentiality is the first priority. However, encryption does not save the need of remote (secure) erase for reasons as follows:

- Breaking of cryptographic algorithms. Encrypted data is merely computationally safe at present. Over time, hardware processing power will improve and cryptanalysis may also evolve. For important sensitive data, it may happen that the encryption is defeated before the data becomes useless.

- Implementation flaws. Vulnerabilities in system implementation always cause sensitive data disclosed earlier than broken algorithms. Depending on how keys are stored or derived, there is the possibility of secret leakage as long as the data is kept on the device.

In view of such facts, the study of erasing encrypted data remotely and securely is of great values. The former one may still persist as long as we perform cryptographic deletion (overwriting of data takes a much longer time). Minimizing the TCB and enforcing cryptographic validation of the key deletion process with hardware security primitives can at least improve the latter situation.

Service availability. A first important factor to consider in the setting of remote erase is whether the device can be erased (reached) in a timely manner as needed by the user. For example, if the device is found to be lost/stolen, or a threat of data revelation is detected for a remote device the user owns, an erase operation needs to be performed as soon as possible.

However, remote erase possesses two intrinsic dependencies that must be satisfied: *power supply* and *connectivity*. If the device can be completely switched off or disconnected by the adversary, remote erase is impossible. In very rare cases, when the data is highly valuable, a battery-backed tamper-resistant device still allows erase operations (e.g., IBM CryptoCards [20]) within the device.

On the other hand, connectivity (which includes Internet access) is as essential but more susceptible, i.e., a device is more easily isolated than removed from power supply. An electromagnetic shielding box would do the job. In spite of this, there are multiple proposals that take a step back assuming a limited communication channel still exists. Kuppusamy et al. [163] allow SIM card changes (thus with a different phone number) by having the target device contact a pre-configured server in various ways including SMS. If the adversary also tries to take out the SIM card, disabling both SMS and cellular Internet access, Yu et al. [309] propose to make use of emergency calls as the communication channel for sending erase commands. However, their approach relies on the operator's support, as emergency calls can only reach designated numbers (e.g., 911 in North America).

Promptness. Nowadays, cryptographic deletion has widely replaced (multi-round) overwriting-based erase, which is slow and subject to interruption. Usually, user data on the device is first encrypted before saved to non-volatile storage. Therefore, in cryptographic deletion, only the encryption key is erased and this process takes very trivial time and is thus non-interruptible.

Effectiveness. A significant difference between generic secure remote computing and secure remote erase is that the latter also requires secure local I/O to reach the persistent storage where the target data resides. Here we skip the discussion of non-cryptographic deletion (overwriting or file system based) for obvious security and performance reasons.

Therefore, in addition to the correctness of the execution logic (e.g., the right command has been issued with the expected response), to achieve an effective erase of the keys in persistent storage, two aspects must be considered: 1) The operation was not interfered with. This can be achieved by running high-privilege/exclusive code avoiding lower-level threats, or containing the keys in TEE. 2) There is not another copy of the deleted keys (at least on this target device). This is about how the secret is stored and used.

Put another way, a remote secure erase solution encompasses the whole of the data protection solution, hence ensuring that the erase operation does lead to data unavailability. Such effectiveness of remote erase is not explicitly discussed in the literature, although some (industrial) solutions may have achieved it. For instance, in a comprehensive survey [168] of recent secure remote wipe solutions, the authors only list “Acknowledge Source that Wipe is Completed” (a confirmation of success issued by the target device), “Secure Delete” (whether the deleted data is recoverable from the media) and “Secure Wipe Command” (simply encrypting commands to avoid sniffing and tampering), instead of the two aspects above.

Trusted parties. The user always has to trust certain third parties with her invaluable data, which is inevitable if she would like to have data stored remotely. However, even if the company that the user must rely on is benign and honest, the trustworthiness of the user’s erase operation may still be doubtable. Examples include: a) Post-deployment misconfiguration. The provider’s infrastructure changes with time frequently. Due to human mistakes or program errors, an initially correct configuration can malfunction. b) Malicious employees. A proper environment will not allow employees to have access to user data and the keys. However, they may somehow interfere with the erase process (e.g., by duplicating encrypted data). c) Compulsory government cooperation. There have been multiple incidents [284] where the provider is legally obliged to expose customers’ data.

In most cases, the user has to go through an enrollment process with a trusted service provider [168], and when a remote erase is needed the initiating device logs in to the service where the provider contacts the target device. This manner has a benefit of better service availability (device reachability) to a certain extent (e.g., the SIM card has been replaced but SMS from the device to the service provider is still possible).

There are two options to avoid such trusted parties (but not the hardware manufacturers):

- End-to-end connection with the target device (e.g., direct TLS session with no online server).
- Using remote attestation protocol opaque to the server (logically end-to-end).

Guessing prevention. While most approaches that do consider password bruteforcing, they mainly focus on guessing the password for illegitimate erase (i.e., a form of denial-of-service attack) with no user consent. Here we argue that if the data protection is susceptible to guessing (e.g., encryption key derived from a weak password), a separate strong password for remote data erase does not help with data leakage.

Leom et al. [168] briefly discuss an iCloud brute-force attack which also leads to data leakage in addition to simply illegitimate erase, which was patched soon thereafter.

In a setting of end-to-end approaches (logically or physically as discussed above), the guessing prevention mechanism must be implemented on the target device instead of any intermediate parties.

User verifiability. We propose to consider the user verifiability of remote erase operations through cryptographic attestation, which leads to what we refer to as trusted remote erase. To the best of our knowledge, this has not been discussed for remote secure erase in the literature.

3.4 End-to-end Verifiable Secure Deletion

In this section, we propose a high-level framework that enables end-to-end trusted remote erase, agnostic to underlying platforms (we later instantiate it in Section 3.5 on x86 PCs). We consider the factors discussed in Section 3.3, and make use of TEE technologies to achieve user verifiability.

Figure 4 depicts the architecture.

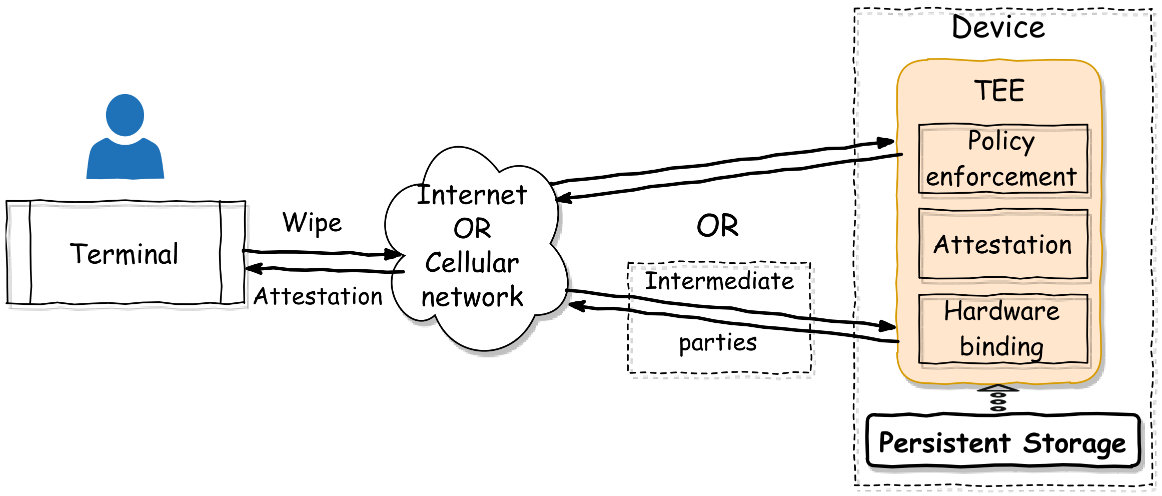


Figure 4: The framework of trusted remote erase

3.4.1 Design considerations

Communication channel. Our approach is independent of the media type through which the wipe command is sent. Cellular communications, direct Internet connection and even radio frequency access are all considered physically end-to-end.

If an intermediate party is involved (e.g., iCloud), we leave its current workflow intact (such as device management, authentication requirement and communication protocol) as long as the target device is eventually triggered to perform the erase operation. An advantage of this is that pending erase is possible. When a target device is powered off or has temporarily lost connectivity, the next time it comes back it will try to contact the intermediate party and thus the pending erase is still performed at the soonest possibility.

End-to-end attestation. Usually all forms of TEE provide a quote-like data element signed (or the like) with the measurement of the execution environment, hence bound to the machine state. Examples include the Intel TXT/TPM quotes and the Intel SGX local/remote attestation protocol. The correctness of such attestation is only determined by the two participating parties (with the freshness nonce blended from the initiator/verifier).

3.5 A Proof-of-concept on x86 PCs

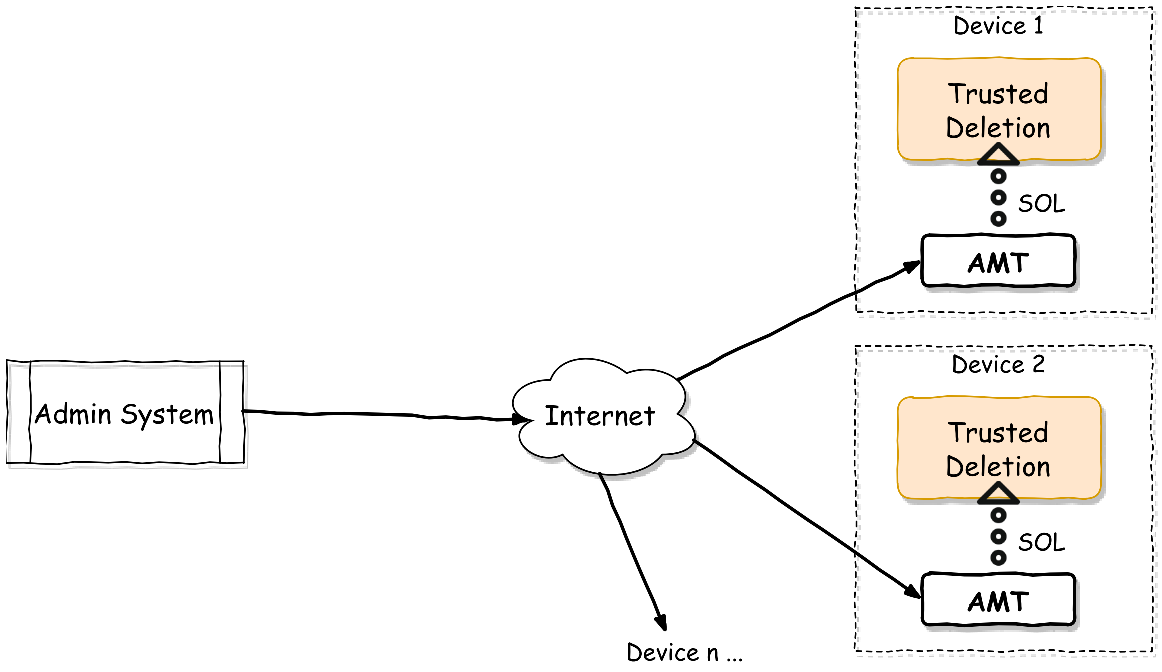


Figure 5: An x86 instantiation of the trusted remote deletion with Intel AMT

In this section, we explain the design and implementation of a tool for end-to-end trusted remote deletion that instantiates the high-level design discussed in Section 3.1). We choose to target x86 PCs and enterprise workstations to showcase the feasibility.

Although we do not focus on deletion success (e.g., against an adversary with physical proximity cutting off the power), the user must be able to initiate deletion at any time, even if the device is manually powered off, halted, in sleep modes or looping infinitely. To this end, we employ Intel AMT [128] as an out-of-band management channel which is available on many off-the-shelf motherboards and located outside the processor complex.

3.5.1 Assumptions and terminology

Admin system. The admin system refers to the computer where the user initiates the erase operation. This system only needs to be network-ready, since all operations are through TCP/IP.

Target system. This is the managed computer/device where the data (to be erased) resides and there might be many of such systems. It must be TXT-capable equipped with a TPM, in addition to the AMT support.

SOL. Serial-Over-Lan (aka., SOL [135]) is one of the AMT-shipped features, which simulates the legacy serial port communication via an IP network. It serves as the communication channel between the admin system and the target system.

AMT password. This refers to the password that the user is required to set when activating the AMT feature on the target system. Whoever trying to connect to the target system must be authenticated with this password. There are complex rules for the composition of an AMT password.

Adversarial model. As we only aim to detect incomplete/failed erase operations, we consider any such detected attempts as denial-of-service (DoS) attacks and exclude them, such as disconnecting the target system from network/power supply, cutting the hard drive cable, tampering with communicated control data, etc. With regard to altering the target system software to evade detection, see the verifiability guarantee in Section 3.5.2

3.5.2 Design overview

Confidence of the erase operation. We base the cryptographic verifiability of an executed program on the combination of Intel TXT and TPM as is used in Gracewipe [311]. Namely, as long as a valid quote received is verified bound to a specific machine, the desired program must have been executed correctly on that machine, indicating a successful erase operation.

Confidentiality of the communication channel. By default, the AMT connection is established with no encryption (i.e., in cleartext HTTP). This is a fatal problem for us since the adversary can eavesdrop the traffic and learn the AMT password sent from the admin system to the target. As client authentication (verifying the identity of the admin system) is already achieved with the AMT password, we merely need two additional defenses: (a) The traffic must be encrypted so that the AMT password is not leaked through eavesdropping. (b) The identify of the target system must be verified or otherwise the password can be leaked through phishing (impersonating the target system to record the password sent).

To achieve the first defense, we can simply change HTTP to HTTPS by enabling TLS, so that a third party can no longer see the plaintext data being transferred. To verify the identity of the target system (acting as the server in terms of the TLS session), performing the regular server authentication will suffice, i.e., a certificate presented by the target system to be verified by the admin system. Any TLS secrets will remain only with AMT, without being exposed to the target host or outside. Note that the TLS client authentication (the admin system presenting its certificate) is not essential in our construction since the admin system is already authenticated by the AMT password. But we can still enable client authentication so that leaking the AMT password does not enable access from any device belonging to the attacker.

Execution of the erase operation. As long as the aforementioned conditions are satisfied, deletion initiation via the network has both the secrecy (through TLS) and integrity (through TXT and TPM) similar to that with the admin's physical presence. The rest of the operation can be executed the same way as with the original Gracewipe.

There are three pieces of key information to be transferred once the communication channel is established: the (deletion) password from Admin to Target triggering the operation, the custom nonce (for generating quotes) from Admin to Target, and the quote values from Target to Admin attesting to the integrity of the operation.

3.5.3 Implementation of Gracewipe Remote

The implementation effort corresponds to two components: the admin client on the admin system that interacts with the admin (human), with GUI or command-line interface; and the new AMT-based functionalities to be integrated to the original Gracewipe on the target system.

As a proof-of-concept for the admin client, we adapt the open-source project *amtterm* [116] with a patch for TLS support [209] by implementing our logic on top of it. We preserve its command-line interface leaving GUI development as future work.

Configuring AMT

In addition to the programmatic implementation, a few AMT configuration steps are required on the target system. Here we omit steps necessary for regular use of AMT (such as setting an AMT password), which are also needed for Gracewipe Remote. No configuration specific to Gracewipe Remote is needed on the admin system.

Basic setup. "Legacy Redirection Mode" must be enabled so that the target system can accept an SOL connection without the need for a management console (see below) to first connect and enable it.

Enabling TLS for server authentication. By default, AMT provides password-only authentication with no encryption (port for SOL: 16994) and TLS can be manually enabled for encrypted communication. To do so, a management console (Intel Manageability Commander Tool [129]) must be used to connect to the target system. Under the Security tab, we may see that the current TLS setting is *Local: NoAuth, Remote: NoAuth* (where Local and Remote refer to local and remote connections respectively). After ticking "Use Transport Layer Security", a dialog pops up prompting for a certificate. Here note that the specified certificate will be stored on the target system and presented to whoever is connecting to it. A root certificate is also needed to be kept on the admin system (and all the systems that would authenticate the target system later). When the new settings are saved, we should see *Local: ServerAuth, Remote: ServerAuth.* Now the SOL accepts TLS connection at the port 16995 (with 16994 closed).

"Accept NON-TLS Connections" must remain unchecked to avoid TLS downgrade attacks. Also, as discussed above, it is not necessary to require the admin system to present a certificate (i.e., "MutualAuth", unless for another layer of protection), which involves more complex steps, such as manually typing the hash value of the root certificate into the MEBX interface (entered by pressing "Ctrl+P" at boot-time).

Certificate management. We make use of the OpenSSL command-line utility to generate both the root certificate (stored on all admin systems) and the leaf certificates to be sent to individual target systems. An AMT-compatible certificate mandates certain requirements which can be satisfied by providing additional parameters to the command line.

Deciding on management modes

The way target systems can be managed varies (e.g., full control vs. remote console access). This leads to the intuitive options to select from as follows:

1. The admin system can reboot the target system remotely by uploading a boot image (the Gracewipe binary and corresponding configuration files). Then based on the target ID, a proper deletion password is retrieved from the database and used to trigger the remote deletion via SOL, depending only on the CPU and TPM of the target system. In this option, Gracewipe must be modified to accept control from and send output to only SOL but not the person present in front of the target system.

2. As an intermediate, we can also boot locally from the target system's hard drive and interact with the local Gracewipe the same way as with the option above.

3. Combining the Manageability Commander Tool and a regular VNC viewer, the third option is similar to remote desktop. It has the advantage of requiring no changes to Gracewipe. But the disadvantage is unacceptable: both sending commands (e.g., typing the deletion password) and retrieving quote values require user interaction and may not be scalable; or if automated, parsing the screen content involves unnecessary implementation complexity.

We decide to choose the first option which has the most advantages. First, the size of the transferred files is trivial (in the order of hundreds of kilobytes, negligible considering the bandwidth of today's network), and doing so leaves less chance to DoS attacks and can maintain proper centralized control. Second, the interaction via SOL has semantics instead of parsing raw VNC screen objects or examining manually, hence allowing easy automation with scalability.

The boot image is transferred with the AMT redirection protocol (also protected by the AMT password), and the deletion password, custom nonce and the quote values are transferred as part of the encrypted SOL communication.

A naive protocol

We use a simple asymmetric protocol for the interaction between the admin system and the target system. Asymmetric here means that data sent to the admin system is organized in packets, while on the target system data received is treated as strings on a serial console. This is in light of the fact that the target system (running Gracewipe at boot-time) has very limited capability in parsing and handling packets with a state machine.

Packets (Target to Admin). We use the following characters to delimit packets:

- The apostrophe (‘) serves as frame start. It always resets the state machine.
- The frame start is always followed by a function code. For example:
  1. 0x02 indicates a new target handshake, followed by the target ID.
  2. 0x01 indicates a literal message to be displayed to the admin, followed by the text.
  3. 0x05 brings back the quote values indicating a successful deletion.
- The Esc (0x1b) is used as frame end.
- The escape character is tilde (˜), to precede any character above and itself.

It is also possible to apply the network-packet-like structures, with header/length information for packet parsing.

Console input/output (Admin to Target). Gracewipe can be either completely switched over to a remote mode where the original standard input/output is (partially) redirected to AMT, or configured to run a separate thread for network communication. Since it does not make much sense to allow both the remote admin and a local user to control the system at the same time, we choose the former for redirection (leaving certain messages on the local console). We follow the way the AT commands [279] in telecommunications are handled and read from console line by line.

Changes to Gracewipe

We try to maintain the minimum changes to Gracewipe. A small but important change is to add dynamic port redirection for all console output functions, switching between the local display and the AMT port. Also, an initial handshake function is added to establish the connection. After this, other trivial adaptations are needed to accommodate the protocol, such as prepending and appending to the original human-readable messages the packet characters discussed above.

3.5.4 Adapting for server-coordinated remote wipe

The principles applied in the design of Gracewipe Remote actually apply to scenarios involving servers as well. Namely, with an isolated trusted execution environment which is attestation-capable (e.g., TXT), the integrity of an operation in that environment can be attested to, regardless of who initiates it and how. One more trusted party is introduced in this case (the owner of the server) and the underlying service must be adapted to accommodate the trusted remote deletion, which is usually nontrivial in practice. Such adaptation concerns both operational logic and storage strategies (see below). Also, the user has to rely on the server to initiate the wipe operation, so no techniques like AMT is needed to establish an end-to-end connection (usually service provides do not allow users to manipulate their servers directly).

Category 1. Like the iCloud remote wipe [18] or Dell remote data delete service [67], the main purpose of remote wipe in this category is to assist the user in erasing lost/stolen devices once they are powered on and connected to Internet. Since data is only stored on the target device, with the help of a plug-in, kernel driver or system application, the wipe operation can be executed in the trusted execution environment on the target device. This requires the service provider's implementation/integration, but the attestation result (e.g., quote values) can be sent to the user either directly or through the trusted server.

Category 2. When the data is stored on the server (i.e., on the cloud, such as [72]), remote wipe is usually not provided explicitly as a feature, to the best of our knowledge. This is because when the user deletes files or folders from her account, it is already remote deletion of data. However, although the service provider is trusted, there is no guarantee that their server is not compromised or the private key is not leaked. It still makes sense to contain the deletion operation inside the trusted execution environment on the server and attest to the result. This may incur a major change to the architecture of the service and thus require cautious consideration. The user may have both "Delete" and "Secure Deletion" options on her client device. Note that if the data has been sync'd to client devices, it is up to the user to make sure no offline copies are leaked.

3.6 Related Work

As most services/products provide features for remote data deletion, as mentioned in the beginning of Section 3.1, we do not list them here as related work, but only point out a few that are comparable to Gracewipe Remote.

Remote Drive Erase (RDE) [131]. As a use case reference design for Intel vPro, RDE was only positioned to demonstrate how AMT can be utilized to achieve secure remote deletion, although the term secure refers to merely multiple rounds of write. With that said, RDE follows very similar execution steps as Gracewipe Remote: an image file called `rde.iso` (a lightweight Linux) is used to boot the remote device; a script erases the hard drive as instructed by the admin and an email is sent as confirmation and documentation. There is no cryptographic proof that the erase process is not interrupted.

Remote Secure Erase (RSE) [132]. Starting from AMT 11.0, Secure Erase Support is added to the firmware (`AMT_BootCapabilities.SecureErase`), i.e., the admin system connected to the target system can issue the erase command without booting any custom image at any time. Its advantage over RDE is that there is less chance that firmware is compromised as compared to the lightweight Linux environment. Nevertheless, just note that, unlike Gracewipe RSE does employ cryptographic deletion, so the erase process is overwriting-based, and thus time-consuming and prone to being interrupted. Likewise, the outcome cannot be attested to.

3.7 Conclusion

In this chapter, we discussed why ensuring successful remote secure wipe is a difficult problem and explained the necessity of verifiability as an alternative solution. We introduced the notion of trusted remote wipe by designing and implementing an extension to Gracewipe [311], named Gracewipe Remote, for end-to-end serverless application scenarios. It is potentially useful in the two exemplary cases of deleting secrets from remote owned computers and avenging laid-off administrators. We also demonstrated how the same methodology could be applied to server-coordinated scenarios. Future work may focus on how new TEE technologies (e.g., Intel SGX with finer granularity) can be incorporated and whether more flexible communication channels are available.

Chapter 4

Hypnoguard: Protecting Secrets across Sleep-wake Cycles

Gracewipe aims to ensuring the confidentiality of disk files, i.e., data-at-rest (in addition to coercion). Meanwhile, we notice that with a high probability the computer faced with physical attacks can be in a suspended mode (data-in-sleep). In this chapter, we apply a similar methodology as Gracewipe and extend the defense scenario to a wider scope, e.g., memory attacks.

4.1 Introduction

Most computers, especially laptops, remain in sleep (S3/suspend-to-RAM), when not in active use (e.g., as in a *lid-close* event); see e.g., [220]. A major concern for unattended computers in sleep is the presence of user secrets in system memory. An attacker with physical access to a computer in sleep (e.g., when lost/stolen, or by coercion) can launch side-channel memory attacks, e.g., DMA attacks [174, 246, 36, 258] by exploiting vulnerable device drivers; common mitigations include: bug fixes, IOMMU (Intel VT-d/AMD Vi), and disabling (FireWire) DMA when the screen is locked (e.g., Mac OS X 10.7.2 and later, Windows 8.1 [174]). A sophisticated attacker can also resort to cold-boot attacks by exploiting DRAM memory remanence effect [108, 102]. Simpler techniques also exist for memory extraction (e.g., [82]); some tools (e.g., [74]) may bypass the OS lock screen and extract in-memory full-disk encryption (FDE) keys.

Some proposals address memory-extraction attacks by making the attacks difficult to launch, or by reducing applicability of known attacks (e.g., [212, 194, 251, 103, 285, 104]; see Section 4.8). Limitations of these solutions include: being too application-specific (e.g., disk encryption), not being scalable (i.e., can support only a few application-specific secrets), and other identified flaws (cf. [35]). Most solutions also do not consider re-authentication when the computer wakes up from sleep. If a regular re-authentication is mandated (e.g., OS unlock), a user-chosen password may not provide enough entropy against guessing attacks (offline/online).

Protecting only cryptographic keys also appears to be fundamentally inadequate, as there exists more privacy/security sensitive content in RAM than keys and passwords. Full memory encryption can be used to keep all RAM content encrypted, as used in proposals for *encrypted execution* (see XOM [169], and a comprehensive survey [113]). However, most such proposals require hardware architectural changes.

Microsoft BitLocker can be configured to provide cold boot protection by relying on S4/suspend-to-disk instead of S3. This introduces noticeable delays in the sleep-wake process. More importantly, BitLocker is not designed to withstand coercion and can provide only limited defence against password guessing attacks (discussed more in Section 4.8).

We propose *Hypnoguard* to protect *all* memory-resident OS/user data across S3 suspensions, against memory extraction attacks, and guessing/coercion of user passwords during wakeup-time re-authentication. Memory extraction is mitigated by performing an in-place full memory encryption before entering sleep, and then restoring the plaintext content/secrets after the wakeup process. The memory encryption key is encrypted by a Hypnoguard public key, the private part of which is stored in a Trusted Platform Module (TPM v1.2) chip, protected by both the user password and the measurement of the execution environment supported by CPU's trusted execution mode, e.g., Intel Trusted Execution Technology (TXT [126]) and AMD Virtualization (AMD-V/SVM [14]). The memory encryption key is thus bound to the execution environment, and can be released only by a proper re-authentication process.

Guessing via Hypnoguard may cause the memory content to be permanently inaccessible due to the deletion of the TPM-stored Hypnoguard private key, while guessing without Hypnoguard, e.g., an attacker-chosen custom wakeup procedure, is equivalent to brute-forcing a high-entropy key, due to TPM protection. A user-defined policy, e.g., three failed attempts, or a special deletion password, determines when the private key is deleted. As a result, either the private key cannot be accessed due to an incorrect measurement of an altered program, or the adversary takes a high risk to guess within the unmodified environment.

By encrypting the entire memory space, except a few system-reserved regions, where no OS/user data resides, we avoid per-application changes. We leverage modern CPU's AES-NI extension and multi-core processing to quickly encrypt/decrypt commonly available memory sizes (up to 8GB, under a second), for avoiding degraded user experience during sleep-wake cycles. For larger memory systems (e.g., 32/64GB), we also provide two variants, for encrypting memory pages of user selected applications, or specific Hypnoguard-managed pages requested by applications.

Due to the peculiarity of the wakeup-time environment, we face several challenges in implementing Hypnoguard. Unlike boot-time (when peripherals are initialized by BIOS) or run-time (when device drivers in the OS are active), at wakeup-time, the system is left in an undetermined state, e.g., empty PCI configuration space and uninitialized I/O controllers. We implement custom drivers and reuse dormant (during S3) OS-saved device configurations to restore the keyboard and VGA display to facilitate easy user input/output (inadequately addressed in the past, cf. [196]).

Several boot-time solutions (e.g., [134, 286, 312]) also perform system integrity check, authenticate the user, and may release FDE keys; however, they do not consider memory attacks during sleep-wake cycles. For lost/stolen computers, some remote tracking services may be used to trigger remote deletion, assuming the computer can be reached online (with doubtful effectiveness, cf. [69, 283]).

Contributions:

1. We design and implement Hypnoguard, a new approach that protects confidentiality of *all* memory regions containing OS/user data across sleep-wake cycles. We provide a defense against memory attacks when the computer is in the wrong hands, and severely restrict guessing of weak authentication secrets (cf. [312]). Several proposals and tools exist to safeguard data-at-rest (e.g., disk storage), data-in-transit (e.g., network traffic), and data-in-use (e.g., live RAM content); with Hypnoguard, we fill the gap of securing *data-in-sleep.*
2. Our primary prototype implementation in Linux uses full memory encryption to avoid per-application changes. The core part of Hypnoguard is decoupled from the underlying OS and system BIOS, for better portability and security. Leveraging modern CPU's AES-NI extension and multi-core processing, we achieve around 8.7GB/s encryption/decryption speed for AES in the CTR mode with an Intel i7-4771 processor, leading to under a second additional delay in the sleep-wake process for 8GB RAM.
3. For larger memory systems (e.g., 32GB), where full memory encryption may add

noticeable delay, we provide protection for application-selected memory pages via the POSIX-compliant system call `mmap()` (requiring minor changes in applications, but no kernel patches). Alternatively, Hypnoguard can also be customized to take a list of applications and only encrypt memory pages pertaining to them (no application changes).

4. We enable wakeup-time secure processing, previously unexplored, which can be leveraged for other use-cases, e.g., OS/kernel integrity check.

4.2 Terminologies, Goals and Threat Model

We explain the terminologies used for Hypnoguard, and our goals, threat model and operational assumptions. We use CPU's trusted execution mode (e.g., Intel TXT, AMD-V/SVM), and the trusted platform module (TPM) chip. We provide brief description of some features as used in our proposal and implementation; for details, see, e.g., Parno et al. [213], Intel [126], and AMD [14].

4.2.1 Terminologies

Hypnoguard key pair (HG$_{pub}$, HG$_{priv}$): A pair of public and private keys generated during deployment. The private key, HG$_{priv}$, is stored in a TPM NVRAM index, protected by both the measurement of the environment and the Hypnoguard user password. HG$_{priv}$ is retrieved through the password evaluated by the TPM with the genuine Hypnoguard program running, and can be permanently deleted in accordance with a user-set policy. The public key, HG$_{pub}$, is stored unprotected in TPM NVRAM (for OS/file system independence), and is loaded in RAM after each boot.

Memory encryption key (SK): A high entropy symmetric key (e.g., 128-bit), randomly generated each time before entering sleep, and used for full memory encryption. Before the system enters sleep, SK is encrypted using HG$_{pub}$ and the resulting ciphertext is stored in the small non-encrypted region of memory.

Hypnoguard user password: A user-chosen password to unlock the protected key HG$_{priv}$ at wakeup-time. It needs to withstand only a few guesses, depending on the actual unlocking policy. This password is unrelated to the OS unlock password, which can be optionally suppressed.

TPM "sealing": For protecting HG$_{priv}$ in the TPM, we use the `TPM_NV_DefineSpace` command, which provides environment binding (similar to `TPM_Seal`, but stores

HG_{priv} in an NVRAM index) and authdata (password) protection. We use the term "sealing" to refer to this mechanism for simplicity.

4.2.2 Goals

We primarily consider attacks targeting extraction of secrets through physical access from a computer in S3 sleep (unattended, stolen, or when the owner is under coercion). We want to protect memory-resident secrets against side-channel attacks (e.g., DMA/cold-boot attacks), but we do not consider compromising a computer in S3 sleep for *evil-maid* type attacks (unbeknownst to the user).

More specifically, our goals include: (*G1*) Any user or OS data (secrets or otherwise), SK, and HG_{priv} must not remain in plaintext anywhere in RAM before resuming the OS to make memory attacks inapplicable. (*G2*) The protected content (in our implementation, the whole RAM) must not be retrieved by brute-forcing SK or HG_{priv}, even if Hypnoguard is not active, e.g., via offline attacks. (*G3*) No guessing attacks should be possible against the Hypnoguard user password, unless a genuine copy of Hypnoguard is loaded as the only program in execution. (*G4*) The legitimate user should be able to authenticate with routine effort, e.g., memorization of *strong* passwords is not required. (*G5*) Guessing the user password when Hypnoguard is active should be severely restricted by the penalty of having the secrets deleted.

An additional goal for coercion attacks during wakeup (similar to the boot-time protection of [312]): (*AG1*) when deletion is successful, there should be a cryptographic evidence that convinces the adversary that the RAM secrets are permanently inaccessible.

4.2.3 Threat model and assumptions

1. The adversary may be either an ordinary person with skills to mount memory/guessing attacks, or an organization (non-state) with coercive powers, and considerable but not unbounded computational resources. For example, the adversary may successfully launch sophisticated cold-boot attacks (e.g., [108, 102]), but cannot brute-force a random 128-bit AES key, or defeat the TPM chip and CPU's trusted execution environment (for known implementation bugs and attacks, see e.g., [265, 298, 248]); see also Item (f) in Section 4.7.
2. Before the adversary gains physical control, the computer system (hardware and OS) has not been compromised. After the adversary releases physical control, or a

lost computer is found, the system is assumed to be untrustworthy, i.e., no further use without complete reinitialization. We thus only consider directly extracting secrets from a computer in sleep, excluding any attacks that rely on compromising first and tricking the user to use it later, the so-called evil-maid attacks, which can be addressed by adapting existing defenses, e.g., [97] for wakeup-time. However, no known effective defense exists for more advanced evil-maid attacks, including hardware modifications as in NSA's ANT catalog [98]. Note that, our AES-GCM based implementation can restrict modification attacks on encrypted RAM content.

3. The host OS is assumed to be general-purpose, e.g., Windows or Linux; a TXT/SVM-aware kernel is not needed. Also, the Hypnoguard tool may reside in an untrusted file system and be bootstrapped from a regular OS.
4. We assume all user data, the OS, and any swap space used by the OS are stored encrypted on disk, e.g., using a properly configured software/hardware FDE system (cf. [193, 65]). A secure boot-time solution should be used to enforce strong authentication (cf. [312]). The FDE key may remain in RAM under Hypnoguard's protection. This assumption can be relaxed, only if the data on disk is assumed non-sensitive, or in the case of a diskless node.
5. Any information placed in memory by the user/OS is treated as sensitive. With full memory encryption, it is not necessary to distinguish user secrets from non-sensitive data (e.g., system binaries).
6. The adversary must not be able to capture the computer while it is operating, i.e., in Advanced Configuration and Power Interface (ACPI [11]) S0. We assume the computer goes into sleep after a period of inactivity, or through user actions (e.g., *lid-close* of a laptop).
7. The adversary may attempt to defeat Hypnoguard's policy enforcement mechanism (i.e., when to delete or unlock HG_{priv} during authentication). With physical access, he may intervene in the wakeup process, e.g., by tampering with the UEFI boot script for S3 [296], and may attempt to observe the input and output of our tool and influence its logic. In all cases, he will fail to access HG_{priv}, unless he can defeat TXT/SVM/TPM (via an implementation flaw, or advanced hardware attacks).
8. In the case of coercion, the user never types the correct password but provides only deletion or incorrect passwords, to trigger the deletion of HG_{priv}. We have also considered coercion as a threat during boot-time (see Chapter 2), requiring the computer to be in a powered-off state before the coercive situation. We consider coercion during wakeup; ideally, both systems should be used together.

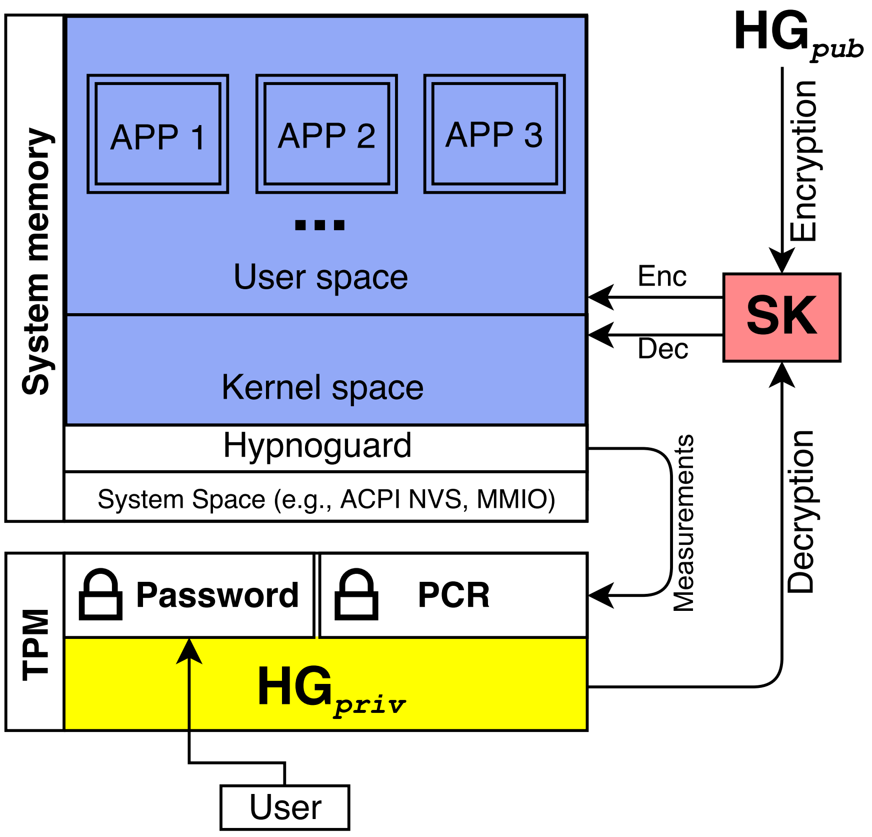


Figure 6: Memory layout and key usage of Hypnoguard. Shaded areas represent encrypted/protected data; different patterns refer to using different schemes/key types.

9. We require a system with a TPM chip and a TXT/SVM-capable CPU with AES-NI (available in many consumer-grade Intel and AMD CPUs). Without AES-NI, full memory encryption will be slow, and users must resort to partial memory encryption.

4.3 Design

In this section, we detail the architecture of Hypnoguard, and demonstrate how it achieves the design goals stated in Section 4.2.2. Technical considerations not specific to our current implementation are also discussed.

Overview. Figure 6 shows the memory layout and key usage of Hypnoguard across sleep-wake cycles; the transition and execution flows are described in Section 4.4.1. User secrets are made unavailable from RAM by encrypting the whole system memory, regardless of kernel or user spaces, with a one-time random symmetric key SK before entering sleep. Then SK is encrypted using HG_{pub} and stored in system memory. At this point, only HG_{priv} can decrypt SK. HG_{priv} is sealed in the TPM chip with the measurements of the genuine copy of Hypnoguard protected by a user password.

At wakeup-time, Hypnoguard takes control in a trusted execution session (TXT/SVM), and prompts the user for the Hypnoguard user password. Only when the correct password is provided in the genuine Hypnoguard environment, HG_{priv} is unlocked from TPM (still in TXT/SVM). Then, HG_{priv} is used to decrypt SK and erased from memory immediately. The whole memory is then decrypted with SK and the system exits from TXT/SVM back to normal OS operations. SK is not reused for any future session.

4.3.1 Design choices and elements

Trusted execution mode. We execute the unlocking program in the trusted mode of modern CPUs (TXT/SVM), where an unforgeable measurement of the execution environment is generated and stored in TPM (used to access HG_{priv}). The use of TXT/SVM and TPM ensures that the whole program being loaded and executed will be reflected in the measurement; i.e., neither the measurement can be forged at the load time nor can the measured program be altered after being loaded, e.g., via DMA attacks. The memory and I/O space of the measured environment is also protected, e.g., via Intel VT-d/IOMMU, from any external access attempt.

We choose to keep Hypnoguard as a standalone module separate from the OS for two reasons. (a) *Small trusted computing base (TCB):* If Hypnoguard's unlocking program is integrated with the OS, then we must also include OS components (at least the kernel and core OS services) in the TPM measurement; this will increase the TCB size significantly. Also, in a consumer OS, maintaining the correct measurements of such a TCB across frequent updates and run-time changes, will be very challenging. Unless measuring the entire OS is the purpose (cf. Unicorn [176]), a TXT/SVM-protected application is usually a small piece of code, not integrated with the OS, to achieve a stable and manageable TCB (e.g., Flicker [183]). In our case, only the core Hypnoguard unlock logic must be integrity-protected (i.e., bound to TPM measurement). The small size may also aid manual/automatic verification of the source code of an implementation. (b) *Portability:* We make Hypnoguard less coupled with the hosting OS except for just a kernel driver, as we may need to work with different distributions/versions of an OS, or completely different OSes.

TPM's role. TPM serves three purposes in Hypnoguard:

1. By working with TXT/SVM, TPM's platform configuration registers (PCRs) maintain the unforgeable measurement of the execution environment.
2. We use TPM NVRAM to store HG_{priv} safely with two layers of protection. First, HG_{priv} is bound to the Hypnoguard environment (e.g., the Intel SINIT module and the Hypnoguard unlocking program). Any binary other than the genuine copy of Hypnoguard will fail to access HG_{priv}. Second, an authdata secret, derived from the Hypnoguard user password, is also used to protect HG_{priv}. Failure to meet either of the above two conditions will lead to denial of access.
3. If HG_{priv} is deleted by Hypnoguard (e.g., triggered via multiple authentication failures, or the entry of a deletion password), we also use TPM to provide a quote, which is a digest of the platform measurement signed by the TPM's attestation identity key (AIK) seeded with an arbitrary value (e.g., time stamp, nonce). Anyone, including the adversary, can verify the quote using TPM's public key at a later time, and confirm that deletion has happened.
4. For generation of the long-term key pair HG_{priv} and HG_{pub}, and the per-session symmetric key SK, we need a reliable source of randomness. We use the `TPM_GetRandom` command to get the required number of bytes from the random number generator in TPM [277] (and optionally, mix them with the output from the RDRAND instruction in modern CPUs).

Necessity of HG_{priv} and HG_{pub}. Although we use a random per sleep-wake cycle symmetric key (SK) for full memory encryption, we cannot directly seal SK in TPM (under the Hypnoguard password), i.e., avoid using (HG_{priv}, HG_{pub}). The reason is that we perform the platform-bound user re-authentication only once at the wakeup time, and without involving the user before entering sleep, we cannot password-seal SK in TPM. If the user is required to enter the Hypnoguard password *every time* before entering sleep, the user experience will be severely affected. We thus keep SK encrypted under HG_{pub} in RAM, and involve the password only at wakeup-time to release HG_{priv} (i.e., the password input is similar to a normal OS unlock process).

4.3.2 Unlock/deletion policy and deployment

Unlocking policy. A user-defined unlocking policy will determine how Hypnoguard reacts to a given password, i.e., what happens when the correct password is entered vs. when a deletion or invalid password is entered. If the policy allows many/unlimited online (i.e., via Hypnoguard) guessing attempts, a dictionary attack might be mounted, violating goal *G5*; the risk to the attacker in this case is that he might unknowingly enter the deletion password. If the composition of the allowed password is not properly chosen (e.g., different character sets for the correct password and the deletion password), an adversary may be able to recognize the types of passwords, and thus avoid triggering deletion.

Static policies can be configured with user-selected passwords and/or rule-based schemes that support evaluating an entered password at run-time. Security and usability trade-offs should be considered, e.g., a quick deletion trigger vs. tolerating user mistyping or misremembering (cf. [57]). During setup, both unlocking and deletion passwords are chosen by the user, and they are set as the access passwords for corresponding TPM NVRAM indices: the deletion password protects an index with a deletion indicator and some random data (as dummy key), and the unlocking password protects an index containing a null indicator and HG_{priv} (similar to Gracewipe, as in Chapter 2). Note that, both the content and deletion indicator of an NVRAM index are protected (i.e., attackers cannot exploit the indicator values). Multiple deletion passwords can also be defined. We also use a protected monotonic counter to serve as a *fail counter*, sealed under Hypnoguard, and initialized to 0. We use a regular numeric value sealed in NVRAM (i.e., inaccessible outside of Hypnoguard); the TPM monotonic counter facility can also be used. The fail counter is used to allow only a limited number of incorrect attempts, after which, deletion is triggered; this is specifically important to deal with lost/stolen cases.

At run-time, only when the genuine Hypnoguard program is active, the fail counter is incremented by one, and a typed password is used to attempt to unlock the defined indices, sequentially, until an index is successfully opened, or all the indices are tried. In this way, the evaluation of a password is performed only within the TPM chip and no information about any defined plaintext passwords or HG_{priv} is leaked in RAM—*leaving no chance to cold-boot attacks.* If a typed password successfully unlocks an index (i.e., a valid password), the fail counter is decremented by one; otherwise, the password entry is considered a failed attempt and the incremented counter is not decremented. When the counter reaches a preset threshold, deletion is triggered. The counter is reset to 0 only when the correct password is entered (i.e., HG_{priv} is successfully unlocked). Thus, a small threshold (e.g., 10) may provide a good balance between security (quick deletion trigger) and usability (the number of incorrect entries that are tolerated). For high-value data, the threshold may be set to 1, which will trigger deletion immediately after a single incorrect entry.

Deployment/setup phase. With a setup program in the OS, we generate a 2048-bit RSA key pair and save HG_{pub} in TPM NVRAM (unprotected), and ask the user to create her passwords for both unlocking and deletion. With the unlocking password (as authdata secret), HG_{priv} is stored in an NVRAM index, bound to the expected PCR values of the Hypnoguard environment at wakeup (computed analytically); similarly, indices with deletion indicators are allocated and protected with the deletion password(s). There is also certain OS-end preparation, e.g., loading and initializing the Hypnoguard device drivers; see Section 4.4.1.

4.3.3 How goals are achieved

Hypnoguard's goals are defined in Section 4.2.2. *G1* is fulfilled by Hypnoguard's full memory encryption, i.e., replacement of all plaintext memory content, with corresponding ciphertext generated by SK. As the OS or applications are not involved, in-place memory encryption can be performed reliably. SK resides in memory encrypted under HG_{pub} (right after full memory encryption is performed under SK). HG_{priv} can only be unlocked with the correct environment and password at wakeup-time, and is erased from RAM right after its use in the trusted execution mode.

A random SK with adequate length generated each time before entering sleep, and a strong public key pair (HG_{pub}, HG_{priv}) generated during setup guarantee *G2*.

TPM sealing (even with a weak Hypnoguard user password) helps achieve *G3*. Without loading the correct binary, the adversary cannot forge the TPM measurement and trick TPM to access the NVRAM index (cf. [126, 277]); note that, learning the expected PCR values of Hypnoguard does not help the attacker in any way. The adversary is also unable to brute-force the potentially weak user password, if he is willing to program the TPM chip without Hypnoguard, as TPM ensures the consistent failure message for both incorrect passwords and incorrect measurements.

The user is required to memorize a regular password for authentication. If the adversary keeps the genuine environment but does not know the correct password, he may be only left with a high risk of deleting HG_{priv}. The legitimate user, however, knows the password and can control the risk of accidental deletion, e.g., via setting an appropriate deletion threshold. Therefore *G4* is satisfied.

When the adversary guesses within Hypnoguard, the password scheme (unlocking policy) makes sure that no (or only a few, for better usability) guessing attempts are allowed before deletion is triggered. This achieves *G5*.

The additional goal for coercion attacks is achieved through the TPM Quote operation. The quote value relies on mainly two factors: the signing key, and the measurement to be signed. An RSA key pair in TPM called AIK (Attestation Identity Key) serves as the signing key. Its public part is signed by TPM's unique key (Endorsement Key, aka. EK, generated by the manufacturer and never leaves the chip in any operations) and certified by a CA in a separate process (e.g., during setup). This ensures the validity of the signature. The data to be signed is the requested PCR values. In TXT, the initial PCR value is set to 0, and all subsequent extend operations will update the PCR values in an unforgeable manner (via SHA1). As a result, as long as the quote matches the expected one, the genuine copy of the program must have been executed, and thus *AG1* is achieved.

4.4 Implementation

In this section, we discuss our implementation of Hypnoguard under Linux using Intel TXT as the trusted execution provider. Note that Hypnoguard's design is OS-independent, but our current implementation is Linux specific; the only component that must be developed for other OSes is HypnoOSService (see below). We also performed an experimental evaluation of Hypnoguard's user experience (for 8GB RAM); no noticeable latency was observed at wakeup-time (e.g., when the user sees the lit-up screen). We assume that a delay under a second before entering sleep and during wakeup is acceptable. For larger memory sizes (e.g., 32GB), we implement two variants to quickly encrypt selected memory regions.

4.4.1 Overview and execution steps

The Hypnoguard tool consists of three parts: HypnoCore (the unlocking logic and cipher engine), HypnoDrivers (device drivers used at wakeup-time), and HypnoOSService (kernel service to prepare for S3 and HypnoCore). HypnoCore and HypnoDrivers operate outside of the OS, and HypnoOSService runs within the OS. The approximate code size of our implementation is: HypnoCore, 7767 LOC (in C/C++/assembly, including reused code for TPM, AES, RSA, SHA1); HypnoDrivers, 3263 LOC (in C, including reused code for USB); HypnoOSService, 734 LOC in C; GCM, 2773 LOC (in assembly, including both the original and our adapted constructions); and a shared framework between the components, 639 LOC in assembly.

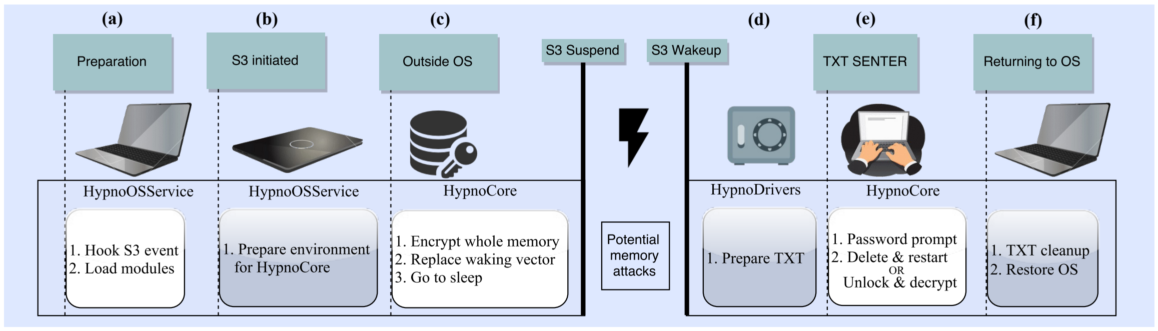


Figure 7: Simplified execution steps of Hypnoguard

Execution steps. Figure 7 shows the generalized execution steps needed to achieve the designed functionalities on an x86 platform. (a) The preparation is done by HypnoOSService at any time while the OS is running before S3 is triggered. HypnoCore, HypnoDrivers, ACM module for TXT, and the TXT policy file are copied into fixed memory locations known by Hypnoguard (see Section 4.4.3). Also, HypnoOSService registers itself to the OS kernel so that if the user or a system service initiates S3, it can be invoked. (b) Upon entry, necessary parameters for S3/TXT are prepared and stored (those that must be passed from the active OS to Hypnoguard), and the kernel's memory tables are replaced with ours, mapped for HypnoCore and HypnoDrivers. (c) Then, HypnoCore encrypts the whole memory in a very quick manner through multi-core processing with AES CTR mode using SK. SK is then encrypted by HG_{pub} (an RSA-2048 key). Before triggering the actual S3 action by sending commands to ACPI, Hypnoguard must replace the original OS waking vector to obtain control back when the machine is waken up. (d) At S3 wakeup, the 16-bit realmode entry, residing below 1MB, of Hypnoguard waking vector is triggered. It calls HypnoDrivers to re-initialize the keyboard and display, and prepares TXT memory structures (TXT heap) and page tables. (e) Then the user is prompted for a password, which is used to unlock TPM NVRAM indices one by one. Based on the outcome and the actual unlocking policy, either deletion of HG_{priv} happens right away and a quote is generated for further verification (and the system is restarted), or if the password is correct, HG_{priv} is unlocked into memory. After decrypting SK, HG_{priv} is erased promptly from memory. HypnoCore then uses SK to decrypt the whole memory. (f) TXT is torn down, and the OS is resumed by calling the original waking vector.

Machine configuration. We use an Intel platform running Ubuntu 15.04 (kernel version: 3.19.0). The development machine's configuration includes: an Intel Core i7-4771 processor (3.50 GHz, 4 physical cores), with Intel's integrated HD Graphics 4600, Q87M-E chipset, 8GB RAM (Kingston DDR3 4GBx2, clock speed 1600 MHz), and 500GB Seagate self-encrypting drive. In theory, our tool should work on most machines with TPM, AES-NI and Intel TXT (or AMD SVM) support, with minor changes, such as downloading the corresponding SINIT module.

4.4.2 Instrumenting the S3 handler

Hypnoguard needs to gain control at wakeup-time before the OS resume process begins. For simplicity, we follow the method as used in a similar scenario in Intel tboot [134]. An x86 system uses ACPI tables to communicate with the system software (usually the OS) about power management parameters. The firmware waking vector, contained in the Firmware ACPI Control Structure (FACS), stores the address of the first instruction to be executed after wakeup; and to actually put the machine to sleep, certain platform-specific data, found in the Fixed ACPI Description Table (FADT), must be written to corresponding ACPI registers.

We must register Hypnoguard with an OS callback for replacing the waking vector, so as not to interfere with normal OS operations. In Linux, the *__acpi_os_prepare_sleep()* callback can be used, which will be invoked in the kernel space before entering sleep. However, we cannot just replace the waking vector in this callback and return to the OS, as Linux overwrites the waking vector with its own at the end of S3 preparation, apparently, to ensure a smooth resume. Fortunately, the required data to be written to ACPI registers is already passed in as arguments by the kernel, and as the OS is ready to enter sleep, we put the machine to sleep without returning to the OS.

4.4.3 Memory considerations

To survive across various contexts (Linux, non-OS native, initial S3 wakeup and TXT), and not to be concerned with paging and virtual memory addressing, we reserve a region from the system memory by providing a custom version of the e820 map [1], so that Linux will not touch it afterwards. This is done by appending a

[1]e820 is shorthand to refer to a table with which the BIOS reports the memory map to the operating system or boot loader.

kernel command line parameter *memmap*. In Windows, this can be done by adding those pages to *BadMemoryList*. 1 MB space at 0x900000 is allocated for HypnoCore, HypnoDrivers and miscellaneous parameters to be passed between different states, e.g., the SINIT module, original waking vector of Linux, policy data, stack space for each processor core, and Intel AES-NI library (see Section 4.5).

Full memory coverage in 64-bit mode. To support more than 4GB memory sizes, we need to make Hypnoguard 64-bit addressable. However, we cannot simply compile the Hypnoguard binary into 64-bit mode as most other modules, especially those for TXT and TPM access, are only available in 32-bit mode, and adapting them to 64-bit will be non-trivial (if possible), because of the significantly different nature of 64-bit mode (e.g., mandatory paging).

We keep HypnoCore and HypnoDrivers unchanged, and write a trampoline routine for the 64-bit AES-NI library, where we prepare paging and map the 8GB memory before switching to the long mode (64-bit). After the AES-NI library call, we go back to 32-bit mode. Also, the x86 calling conventions may be different than x86-64 (e.g., use of stack space vs. additional registers). A wrapper function, before the trampoline routine goes to actual functions, is used to extract those arguments from stack and save them to corresponding registers. In this way, the 64-bit AES-NI library runs as if the entire HypnoCore and HypnoDrivers binary is 64-bit, and thus we can access memory regions beyond 4GB, while the rest of Hypnoguard still remains in 32-bit mode.

4.4.4 User interaction

In a regular password-based wakeup-time authentication, the user is shown the password prompt dialog to enter the password. In addition to the password input, we also need to display information in several instances, e.g., interacting with the user to set up various parameters during deployment, indicating when deletion is triggered, and displaying the quote (i.e., proof of deletion). Providing both standard input and output is easy at boot-time (with BIOS support), and within the OS. However, resuming from S3 is a special situation: no BIOS POST is executed, and no OS is active. At this time, peripherals (e.g., PCI, USB) are left in an uninitialized state, and unless some custom drivers are implemented, display and keyboard remain nonfunctional.

For display, we follow a common practice as used in Linux for S3 resume (applicable for most VGA adapters). HypnoDrivers invoke the legacy BIOS video routine using “lcallw 0xc000,3” (0xc0000 is the start of the VGA RAM where the video BIOS is copied to; the first 3 bytes are the signature and size of the region, and 0xc0003 is the entry point).

For keyboard support, the S3 wakeup environment is more challenging (PS/2 keyboards can be easily supported via a simple driver). Most desktop keyboards are currently connected via USB, and recent versions of BIOS usually have a feature called “legacy USB support”. Like a mini-OS, as part of the power-on check, the BIOS (or the more recent UEFI services) would set up the PCI configuration space, perform USB enumeration, and initialize the *class drivers* (e.g., HID and Mass Storage). But when we examined the USB EHCI controller that our USB keyboard was connected to, we found that its base address registers were all zeros at wakeup-time, implying that it was uninitialized (same for video adapters). As far as we are aware, no reliable mechanisms exist for user I/O after wakeup. TreVisor [196] resorted to letting the user input in a blank screen (i.e., keyboard was active, but VGA was uninitialized). Note that the actual situation is motherboard-specific, determined mostly by the BIOS. We found that only one out of our five test machines has the keyboard initialized at wakeup-time.

Loading a lightweight Linux kernel might be an option, which would increase the TCB size and (potentially) introduce additional attack surface. Also, we must execute the kernel in the limited Hypnoguard-reserved space. Instead, we enable USB keyboard support as follows:

1. Following the Linux kernel functions *pci_save_state()* and *pci_restore_config_space()*, we save the PCI configuration space before entering S3, and restore it at wakeup-time to enable USB controllers in Hypnoguard.
2. We borrow a minimal set of functions from the USB stack of the GRUB project, to build a tiny USB driver only for HID keyboards operating on the “boot protocol” [282].
3. There are a few unique steps performed at boot-time for USB initialization that cannot be repeated during S3 wakeup. For instance, a suspended hub port (connecting the USB keyboard) is ready to be waken up by the host OS driver and does not accept a new round of enumeration (e.g., getting device descriptor, assigning a new address). We thus cannot reuse all boot-time USB initialization code from GRUB. At the end, we successfully reconfigure the USB hub by initiating a port reset first.

With the above approach, we can use both the USB keyboard and VGA display at wakeup-time. This is hardware-agnostic, as restoring PCI configuration simply copies existing values, and the USB stack as reused from GRUB follows a standard USB implementation. We also implement an i8042 driver (under 100 LOC) to support PS/2 keyboards. Our approach may help other projects that cannot rely on the OS/BIOS for input/output support (e.g., [196, 79]).

4.4.5 Moving data around

Hypnoguard operates at different stages, connected by jumping to an address without contextual semantics. Conventional parameter passing in programming languages and shared memory access are unavailable between these stages. Therefore, we must facilitate binary data transfer between the stages. To seamlessly interface with the host OS, we apply a similar method as in Flicker [183] to create a *sysfs* object in a user-space file system. It appears in the directory "/sys/kernel" as a few subdirectories and two files: *data* (for accepting raw data) and *control* (for accepting commands). In HypnoOSService, the sysfs handlers write the received data to the 1MB reserved memory region. When S3 is triggered, HypnoDrivers will be responsible for copying the required (portion of) binary to a proper location, for instance, the real-mode wakeup code to 0x8a000, SINIT to the BIOS-determined location SINIT.BASE and the LCP policy to the *OsMleData* table, which resides in the TXT heap prepared by HypnoDrivers before entering TXT.

4.4.6 Unencrypted memory regions

In our full memory encryption, the actual encrypted addresses are not contiguous. We leave BIOS/hardware reserved regions unencrypted, which fall under two categories. (a) MMIO space: platform-mapped memory and registers of I/O devices, e.g., the TPM locality base starts at 0xfed40000. (b) Platform configuration data: memory ranges used by BIOS/UEFI/ACPI; the properties of such regions vary significantly, from read-only to non-volatile storage.

Initially, when we encrypted the whole RAM, including the reserved regions, we observed infrequent unexpected system behaviors (e.g., system crash). As much as we are aware of, no user or OS data is stored in those regions (cf. [138]), and thus there should be no loss of confidentiality due to keeping those regions unencrypted. Hypnoguard parses the e820 (memory mapping) table to determine the memory regions accessible by the OS. In our test system, there is approximately 700MB reserved space, spread across different ranges below 4GB. The amount of physical memory is compensated by shifting the addresses, e.g., for our 8GB RAM, the actual addressable memory range goes up to 8.7GB.

4.5 High-speed Full Memory Encryption and Decryption

The adoptability of the primary Hypnoguard variant based on full memory encryption/decryption mandates a minimal impact on user experience. Below, we discuss issues related to our implementation of quick memory encryption.

For all our modes of operation with AES-NI, the processing is 16-byte-oriented (i.e., 128-bit AES blocks) and handled in XMM registers. In-place memory encryption/decryption is intrinsically supported by taking an input block at a certain location, and overwriting it with the output of the corresponding operation. Therefore, no extra memory needs to be reserved, and thus no performance overhead for data transfer is incurred.

4.5.1 Enabling techniques

Native execution. We cannot perform in-place memory encryption when the OS is active, due to OS memory protection and memory read/write operations by the OS. Thus, the OS must be inactive when we start memory encryption. Likewise, at wakeup-time in TXT, there is no OS run-time support for decryption. We need to perform a single-block RSA decryption using HG_{priv} to decrypt the 128-bit AES memory encryption key SK. On the other hand, we need fast AES implementation to encrypt the whole memory (e.g., 8GB), and thus, we leverage new AES instructions in modern CPUs (e.g., Intel AES-NI). AES-NI offers significant performance boost (e.g., about six times in one test [46]). Although several crypto libraries now enable easy-to-use support for AES-NI, we cannot use such libraries, or the kernel-shipped library, as we do not have the OS/run-time support. We use Intel's AES-NI library [232], with minor but non-trivial modifications (discussed in our tech report [313]).

OS-less multi-core processing. Outside the OS, no easy-to-use parallel processing interface is available. With one processor core, we achieved 3.3–4GB/s with AES-NI, which would require more than 2 seconds for 8GB RAM (still less satisfactory, considering 3 cores being idle). Thus, to leverage multiple cores, we develop our own multi-core processing engine, mostly following the Intel MultiProcessor Specification [127]. Our choice of decrypting in TXT is non-essential, as SK is generated per sleep-wake cycle and requires no TXT protection; however, the current logic is simpler and requires no post-TXT cleanup for native multi-core processing.

Modes of operation. Intel's AES-NI library offers ECB, CTR and CBC modes. We use AES in CTR mode as the default option (with a random value as the initial counter); compared to CBC, CTR's performance is better, and symmetric between encryption and decryption speeds (recall that CBC encryption cannot be parallelized due to chaining). In our test, CBC achieves 4.5GB/s for encryption and 8.4GB/s for decryption. In CTR mode, a more satisfactory performance is achieved: 8.7GB/s for encryption and 8.5GB/s for decryption (approximately).

When ciphertext integrity is required to address content modification attacks, AES-GCM might be a better trade-off between security and performance. We have implemented a Hypnoguard variant with a custom, performance-optimized AES-GCM mode; for implementation details and challenges, see our tech report [313].

4.5.2 Performance analysis

Relationship between number of CPU cores and performance. For AES-CTR, we achieved 3.3–4GB/s (3.7GB/s on average), using a single core. After a preliminary evaluation, we found the performance is not linear to the number of processor cores, i.e., using 4 cores does not achieve the speed of 16GB/s, but at most 8.7GB/s (8.3GB/s on 3 cores and 7.25GB/s on 2 cores).

A potential cause could be Intel Turbo Boost [51] that temporarily increases the CPU frequency when certain limits are not exceeded (possibly when a single core is used). Suspecting the throughput of the system RAM to be the primary bottleneck (DDR3), we performed benchmark tests with user-space tools, e.g., mbw [119], which simply measures *memcpy* and variable assignment for an array of arbitrary size. The maximum rate did not surpass 8.3GB/s, possibly due to interference from other processes.

During the tests with GCM mode, our observation demonstrates the incremental improvement of our implementation: 2.5GB/s (1-block decryption in C using one core), 3.22GB/s (1-block decryption in C using four cores), 3.3GB/s (4-block decryption in C using four cores), 5GB/s (4-block decryption in assembly using four cores), and 6.8GB/s (4-block decryption in assembly with our custom AES-GCM [313]). The encryption function in assembly provided by Intel already works satisfactorily, which we do not change further. The performance numbers are listed in Table 2.

At the end, when ciphertext integrity is not considered (the default option), 8.7GB/s in CTR mode satisfies our requirement of not affecting user experience, specifically, for systems up to 8GB RAM. When GCM is used for ciphertext integrity, we achieve 7.4GB/s for encryption and 6.8GB/s for decryption (i.e., 1.08 seconds for entering sleep and 1.18 seconds for waking up, which is very close to our 1-second delay limit). Note that, we have zero run-time overhead, after the OS is resumed.

	CTR (1-core)	**CTR**	CBC	GCM-C1 (1-core)	GCM-C1	GCM-C4	GCM-A4	**GCM-A4T**
Encryption	3.7GB/s	8.7GB/s	4.5GB/s	—	—	—	—	7.4GB/s
Decryption	3.7GB/s	8.7GB/s	8.4GB/s	2.5GB/s	3.22GB/s	3.3GB/s	5GB/s	6.8GB/s

Table 2: A comparative list of encryption/decryption performance. Column headings refer to various modes of operation, along with the source language (when applicable; A represents assembly); the trailing number is the number of blocks processed at a time. A4T is our adapted GCM implementation in assembly processing 4 blocks at a time, with delayed tag verification (see [313]); — means not evaluated.

4.6 Variants

For systems with larger RAM (e.g., 32GB), Hypnoguard may induce noticeable delays during sleep-wake cycles, if the whole memory is encrypted. For example, according to our current performance (see Section 4.5), if a gaming system has 32GB RAM, it will take about four seconds for both entering sleep and waking up (in CTR mode), which might be unacceptable. To accommodate such systems, we propose two variants of Hypnoguard, where we protect (i) all memory pages of selected processes—requires no modifications to applications; and (ii) selected security-sensitive memory pages of certain processes—requires modifications. Note that, these variants require changes in HypnoOSService, but HypnoCore and HypnoDrivers remain unchanged (i.e., unaffected by the OS-level implementation mechanisms).

(i) Per-process memory encryption. Compared to the design in Section 4.3, this variant differs only at the choice of the encryption scope. It accepts a process list (e.g., supplied by the user) and traverses all memory pages allocated to those processes to determine the scope of encryption. We retrieve the virtual memory areas (VMA, of type *vm_area_struct*) from *task* $\rightarrow$ *mm* $\rightarrow$ *mmap* of each process. Then we break the areas down into memory pages (in our case, 4K-sized) before converting them over to physical addresses. This is necessary even if a region is continuous as VMAs, because the physical addresses of corresponding pages might not be continuous. We store the page list in Hypnoguard-reserved memory.

Our evaluation shows that the extra overhead of memory traversal is negligible. This holds with the assumption that the selected apps are allocated a small fraction of a large memory; otherwise, the full memory or mmap-based variant might be a better choice. For smaller apps such as bash (38 VMAs totaling 1,864 pages, approximately 7MB), it takes 5 microseconds to traverse through and build the list. For large apps such as Firefox (723 VMAs totaling 235,814 pages, approximately 1GB), it takes no more than 253 microseconds. Other apps we tested are Xorg (167 microseconds) and gedit (85 microseconds). We are yet to fully integrate this variant into our implementation (requires a more complex multi-core processing engine).

(ii) Hypnoguard-managed memory pages via `mmap()`. There are also situations where a memory-intensive application has only a small amount of secret data to protect. Assuming per-application changes are acceptable, we implement a second variant of Hypnoguard that exposes a file system interface compliant with the POSIX call mmap(), allowing applications to allocate pages from a Hypnoguard-managed memory region.

The mmap() function is defined in the *file_operations* structure, supported by kernel drivers exposing a device node in the file system. An application can request a page to be mapped to its address space on each mmap call, e.g., instead of calling *malloc()*. On return, a virtual address mapped into the application's space is generated by Hypnoguard using *remap_pfn_range()*. An application only needs to call mmap(), and use the returned memory as its own, e.g., to store its secrets. Then the page is automatically protected by Hypnoguard the same way as the full memory encryption, i.e., encrypted before sleep and decrypted at wakeup. The application can use multiple pages as needed. We currently do not consider releasing such pages (i.e., no unmap()), as we consider a page to remain sensitive once it has been used to store secrets. Note that, no kernel patch is required to support this variant. We tested it with our custom application requesting pages to protect its artificial secrets. We observed no latency or other anomalies.

4.7 Security Analysis

Below, we discuss potential attacks against Hypnoguard; see also Sections 4.2.3 and 4.3.3 for related discussion.

(a) Cold-boot and DMA attacks. As no plaintext secrets exist in memory after the system switches to sleep mode, cold-boot or DMA attacks cannot compromise memory confidentiality; see Section 4.3.3, under *G1*. Also, the password evaluation process happens inside the TPM (as TPM receives it through one command and compares with its preconfigured value; see Section 4.3.2), and thus the correct password is not revealed in memory for comparison. At wakeup-time, DMA attacks will also fail due to memory access restrictions (TXT/VT-d).

(b) Reboot-and-retrieve attack. The adversary can simply give up on waking back to the original OS session, and soft-reboot the system from any media of his choice, to dump an arbitrary portion of the RAM, with most content unchanged (the so-called *warm boot* attacks, e.g., [55, 288, 287]). Several such tools exist, some of which are applicable to locked computers, see e.g., [82]. With Hypnoguard, as the whole RAM is encrypted, this is not a threat any more.

(c) Consequence of key deletion. The deletion of HG$_{priv}$ severely restricts guessing attacks on lost/stolen computers. For coercive situations, deletion is needed so that an attacker cannot force users to reveal the Hypnoguard password after taking a memory dump of the encrypted content. Although we use a random AES key SK for each sleep-wake cycle, simply rebooting the machine without key deletion may not suffice, as the attacker can store all encrypted memory content, including SK encrypted by HG$_{pub}$. If HG$_{priv}$ can be learned afterwards (e.g., via coercion of the user password), the attacker can then decrypt SK, and reveal memory content for the target session.

If a boot-time anti-coercion tool, e.g., Gracewipe (cf. Chapter 2), is integrated with Hypnoguard, the deletion of HG$_{priv}$ may also require triggering the deletion of Gracewipe secrets. Hypnoguard can easily trigger such deletion by overwriting TPM NVRAM indices used by Gracewipe, which we have verified in our installation. From a usability perspective, the consequence of key deletion in Hypnoguard is to reboot and rebuild the user secrets in RAM, e.g., unlocking an encrypted disk, password manager, or logging back into security-sensitive websites. With Gracewipe integration, triggering deletion will cause loss of access to disk data.

(d) Compromising the S3 resume path. We are unaware of any DMA attacks that can succeed when the system is in sleep, as such attacks require an active protocol stack (e.g., that of FireWire). Even if the adversary can use DMA attacks to alter RAM content *in sleep*, bypassing Hypnoguard still reveals no secrets, due to full memory encryption and the unforgeability of TPM measurements. Similarly, replacing the Hypnoguard waking vector with an attacker chosen one (as our waking vector resides in memory unencrypted), e.g., by exploiting vulnerabilities in UEFI resume boot script [138, 296] (if possible), also has no effect on memory confidentiality. Any manipulation attack, e.g., insertion of malicious code via a custom DRAM interposer, on the encrypted RAM content to compromise the OS/applications after wakeup is addressed by our GCM mode implementation (out of scope for the default CTR implementation).

(e) Interrupting the key deletion. There have been a few past attacks about tapping TPM pins to detect the deletion when it is triggered (for guessing without any penalty). Such threats are discussed elsewhere (e.g., [312]), and can be addressed, e.g., via redundant TPM write operations.

(f) Other hardware attacks. Ad-hoc hardware attacks to sniff the system bus for secrets (e.g., [37]) are generally inapplicable against Hypnoguard, as no secrets are processed before the correct password is entered. For such an example attack on Xbox, see [121], which only applies to architectures with LDT (HyperTransport) bus, not Intel's FSB.

However, more advanced hardware attacks may allow direct access to the DRAM bus, and even extraction of TPM secrets with an invasive *decapping* procedure (e.g., [265], see also [111] for more generic physical attacks on security chips). Note that the PC platform (except the TPM chip to some extent) cannot withstand such attacks, as components from different manufactures need to operate through common interfaces (vs. more closed environment such as set-top boxes). With TPMs integrated into the Super I/O chip, and specifically, with firmware implementation of TPM v2.0 (fTPM as in Intel Platform Trust Technology), decapping attacks may be mitigated to a significant extent (see the discussion in [223] for discrete vs. firmware TPMs). Hypnoguard should be easily adapted to TPM v2.0.

4.8 Related Work

In this section, we primarily discuss related work on memory attacks and preventions. Proposals for addressing change of physical possession (e.g., [250, 83]) are not discussed, as they do not consider memory attacks.

Protection against cold-boot and DMA attacks. Solutions to protecting keys exposed in system memory have been extensively explored in the last few years, apparently, due to the feasibility of cold-boot attacks [108]. There have been proposals based on relocation of secret keys from RAM to other "safer" places, such as SSE registers (AESSE [192]), debug registers (TRESOR [194]), MSR registers (Amnesia [251]), AVX registers (PRIME [86]), CPU cache and debug registers (Copker [103]), GPU registers (PixelVault [285]), and debug registers and Intel TSX (Mimosa [104]).

A common limitation of these solutions is that specific cryptographic operations must be offloaded from the protected application to the new mechanism, mandating per-application changes. They are also focused on preventing leakage of only *cryptographic keys*, which is fundamentally limited in protecting RAM content in general. Also, some solutions do not consider user re-authentication at wakeup-time (e.g., [86, 103]). Several of them (re)derive their master secret, or its equivalent, from the user password, e.g., [192, 194]; this may even allow the adversary to directly guess the master secret in an offline manner.

Memory encryption. An ideal solution for memory extraction attacks would be to perform encrypted execution: instructions remain encrypted in RAM and are decrypted right before execution within the CPU; see XOM [169] for an early proposal in this domain, and Henson and Taylor [113] for a comprehensive survey. Most proposals for memory encryption deal with *data in use* by an active CPU. Our use of full memory encryption involves the sleep state, when the CPU is largely inactive. Most systems require architectural changes in hardware/OS and thus remain largely unadopted, or designed for specialized use cases, e.g., bank ATMs. Using dedicated custom processors, some gaming consoles also implement memory encryption to some extent, e.g., Xbox, Playstation. Similar to storing the secrets in safer places, memory encryption schemes, if implemented/adopted, may address extraction attacks, but not user re-authentication.

Forced hibernation. YoNTMA [140] automatically hibernates the machine, i.e., switch to S4/suspend-to-disk, whenever it detects that the wired network is disconnected, or the power cable is unplugged. In this way, if the attacker wants to take the computer away, he will always get it in a powered-off state, and thus memory attacks are mitigated. A persistent attacker may preserve the power supply by using off-the-shelf hardware tools (e.g., [180]). Also, the attacker can perform in-place cold-boot/DMA attacks.

BitLocker. Microsoft's drive encryption tool BitLocker can seal the disk encryption key in a TPM chip, if available. Components that are measured for sealing include: the Core Root of Trust Measurement (CRTM), BIOS, Option ROM, MBR, and NTFS boot sector/code (for the full list, see [189]). In contrast, Hypnoguard measures components that are OS and BIOS independent (may include the UEFI firmware in later motherboard models). In its most secure mode, Microsoft recommends to use BitLocker with multi-factor authentication such as a USB device containing a startup key and/or a user PIN, and to configure the OS to use S4/suspend-to-disk instead of S3/suspend-to-RAM [187]. In this setting, unattended computers would always resume from a powered-off state (cf. YoNTMA [140]), where no secrets remain in RAM; the user needs to re-authenticate with the PIN/USB key to restore the OS.

BitLocker's limitations include the following. (1) It undermines the usability of sleep modes as even with faster SSDs it still takes several seconds to hibernate (approx. 18 seconds in our tests with 8GB RAM in Windows 10 machine with Intel Core-i5 CPU and SSD). Wakeup is also more time-consuming, as it involves the BIOS/UEFI POST screen before re-authentication (approx. 24 seconds in our tests). On the other hand, RAM content remains unprotected if S3 is used. (2) It is still vulnerable to password guessing to some extent, when used with a user PIN (but not with USB key, if the key is unavailable to the attacker). Based on our observation, BitLocker allows many attempts, before forcing a shutdown or entering into a TPM lockout (manufacturer dependent). A patient adversary can slowly test many passwords. We have not tested if offline password guessing is possible. (3) BitLocker is not designed for coercive situations, and as such, it does not trigger key deletion through a deletion password or fail counter. If a user is captured with the USB key, then the disk and RAM content can be easily accessed. (4) Users also must be careful about the inadvertent use of BitLocker's online key backup/escrow feature (see e.g., [23]).

Recreating trust after S3 sleep. To re-establish a secure state when the system wakes up from S3, Kumar et al. [162] propose the use of Intel TXT and TPM for recreating the trusted environment, in the setting of a VMM with multiple VMs. Upon notification of the S3 sleep, the VMM cascades the event to all VMs. Then each VM encrypts its secrets with a key and seal the key with the platform state. The VMM also encrypts its secrets and seals its context. Thereafter, the VMM loader (hierarchically higher than the VMM) encrypts the measurement of the whole memory space of the system with a key that is also sealed. At wakeup-time, all checks are done in the reversed order. If any of the measurements differ, the secrets will not be unsealed. This proposal does not consider re-authentication at wakeup-time and mandates per-application/VM modifications. More importantly, sealing and unsealing are performed for each sleep-wake cycle for the whole operating context: VMM loader, VMM, VMs. Depending on how the context being sealed is defined, this may pose a severe performance issue, as TPM sealing/unsealing is time-consuming; according to our experiment, it takes more than 500ms to process only 16 bytes of data.

Unlocking with re-authentication at S2/3/4 wakeup. When waking up from one of the sleep modes, a locked device such as an FDE hard drive, may have already lost its security context (e.g., being unlocked) before sleep. Rodriguez and Duda [231] introduced a mechanism to securely re-authenticate the user to the device by replacing the original wakeup vector of the OS with a device specific S3 wakeup handler. The user is prompted for the credential, which is directly used to decrypt an unlock key from memory to unlock the device (e.g., the hard drive). This approach does not use any trusted/privileged execution environment, such as Intel TXT/AMD SVM. Without the trusted measurement (i.e., no sealed master key), the only entropy comes from the user password, which may allow a feasible guessing attack.

Secure deallocation. To prevent exposure of memory-bound secrets against easy-to-launch warm-reboot attacks, Chow et al. [55] propose a secure deallocation mechanism (e.g., zeroing freed data on the heap) to limit the lifetime of sensitive data in memory. This approach avoids modifications in application source, but requires changes in compilers, libraries, and OS kernel in a Linux system (and also cannot address cold-boot attacks). Our solution is also effective against warm-reboot attacks, but requires no changes in applications and the OS stack.

Relevant proposals on mobile platforms. Considering their small sizes and versatile functionalities, mobile devices are more theft-prone and more likely to be caught with sensitive data present when the user is coerced. CleanOS [264] is proposed to evict sensitive data not in active use to the cloud and only retrieve the data back when needed. Sensitive information is pre-classified and encapsulated into sensitive data objects (SDOs). Access to SDOs can be revoked in the case of device theft and audited in normal operations. TinMan [301] also relies on a trusted server, but does not decrypt confidential data in the device memory to avoid physical attacks. Keypad [87], a mobile file system, provides fine-grained access auditing using a remote server (which also hosts the encryption keys). For lost devices, access can be easily revoked by not releasing the key from the server. All these proposals require a trusted third party. Also, under coercion, if the user is forced to cooperate, sensitive data will still be retrieved. Moreover, the protected secrets in Hypnoguard might not be suitable for being evicted as they may be used often, e.g., an FDE key.

Gracewipe. For handling user secrets in the trusted execution environment, we follow the methodology from Gracewipe (cf. Chapter 2), which operates at boot-time and thus can rely on BIOS and tboot. In contrast, Hypnoguard operates during the sleep-wake cycle, when no BIOS is active, and tboot cannot be used for regular OSes (tboot assumes TXT-aware OS kernel). Gracewipe assumes that the attacker can get physical possession of a computer, only when it is powered-off, in contrast to Hypnoguard's sleep state, which is more common. Gracewipe securely releases sensitive FDE keys in memory, but does not consider protecting such keys against memory extraction attacks during sleep-wake. Gracewipe addresses an extreme case of coercion, where the data-at-rest is of utmost value. We target unattended computers in general, and enable a wakeup-time secure environment for re-authentication and key release.

Intel SGX. Intel Software Guard Extensions (SGX [16]) allows individual applications to run in their isolated context, resembling TXT with similar features but finer granularity (multiple concurrent secure enclaves along with the insecure world). Memory content is fully encrypted outside the CPU package for SGX-enabled applications. Considering the current positioning of Hypnoguard, we believe that TXT is a more preferable choice, as running either the protected programs or the entire OS in SGX would introduce per-application/OS changes. TXT also has the advantage of having been analyzed over the past decade, as well as its counterpart being available in AMD processors (SVM).

4.9 Concluding Remarks

As most computers, especially, laptops, remain in sleep while not actively used, we consider a comprehensive list of threats against memory-resident user/OS data, security-sensitive or otherwise. We address an important gap left in existing solutions: comprehensive confidentiality protection for *data-in-sleep (S3)*, when the attacker has physical access to a computer in sleep. We design and implement Hypnoguard, which encrypts the whole memory very quickly before entering sleep under a key sealed in

TPM with the integrity of the execution environment. We require no per-application changes or kernel patches. Hypnoguard enforces user re-authentication for unlocking the key at wakeup-time in a TXT-enabled trusted environment. Guessing attacks bypassing Hypnoguard are rendered ineffective by the properties of TPM sealing; and guessing within Hypnoguard will trigger deletion of the key. Thus, Hypnoguard along with a boot-time protection mechanism with FDE support (e.g., BitLocker, Gracewipe) can enable effective server-less guessing resistance, when a computer with sensitive data is lost/stolen. We plan to release the source code of Hypnoguard at a later time, and for now it can be obtained by contacting the authors.

Chapter 5

Trusted Write-protection Against Privileged Data Tampering

In this chapter, we move our focus from data confidentiality to data integrity, driven by the frequent recent incidents about ransomware. Our proposed approach is applicable to privileged unauthorized data alteration in general, with an emphasis on rootkit ransomware.

5.1 Introduction and Motivation

The first known crypto-ransomware dates back to 1989 (only file/directory names were encrypted [173]; see also [269]). Crypto-based attack vectors were formally introduced by Young and Yung in 1996 [306] (see also [307]). After the CryptoLocker attack in 2013, robust crypto-ransomware families have been growing steadily, with a large number of attacks in 2016 (see the F-Secure ransomware "tube-map" [77]). Examples of recent high-impact ransomware attacks, include [172, 274, 110, 21, 2, 244], affecting individuals and enterprise/government systems alike. An IBM X-Factor survey of 600 business leaders and 1021 consumers in the US reveals the effectiveness of current ransomware attacks: 70% of affected businesses paid the ransom (46% of businesses reported to have been infected); individual users are less willing to pay (e.g., 39% users without children may pay ransom for family photos vs. 55% users with children).

Early-day ransomware had the (symmetric) file encryption keys embedded in their obfuscated binaries, or stored in a C&C server. Keys could be recovered by reverse-engineering their code or intercepting C&C traffic. Ransomware now generally uses a public key to encrypt a random file encryption key, and the private key remains only at the attacker's machine (cf. [306]), and thus much more resilient than before; however, even well-designed ransomware may also have flaws [295] that can be leveraged to recover encryption keys. An exemplary umbrella solution is *NoMoreRansom* [6], clustering file recovery efforts from several public and industry partners. However, relying on ransomware authors' mistakes is a non-solution, and finding such exploits may be too late for early victims.

As public-key based modern ransomware renders data recovery more difficult, a legacy defense venue is detection techniques. Common anti-malware approaches relying on binary signatures are largely ineffective against ransomware (see e.g., [239]). Some solutions rely on system/user behavior signatures, exemplified by file system activity monitoring, e.g., [150, 239, 60, 149]. To complement detection based solutions (or assuming they may be bypassed), recovery-based mechanisms may also be deployed, e.g., Paybreak [159] stores (suspected) file encryption keys on-the-fly, right after generated but before encrypted with the ransomware's public key. Several countermeasures against generic rootkit attacks have also been proposed, focusing on intrusion-resiliency and forensics (e.g., S4 [261]), and preventing persistent infection (e.g., RRD [45]). FlashGuard [122] is the only proposal focusing on rootkit ransomware, which leverages the out-of-place write feature of modern SSDs, providing an implicit backup. It requires modifying SSD firmware and a trusted clock within the SSD (currently unavailable). We discuss academic proposals in more detail in Section 5.7.

Another obvious countermeasure against ransomware is to make *offline* backup of important data regularly (on media disconnected from the computer). Although simple in theory, effective deployment/use of backup tools could be non-trivial, e.g., determining frequency of backups, checking integrity of backups regularly (see Laszka et al. [167] for an economic analysis of paying ransom vs. backup strategies). More problematically, the disconnected media must be connected (online) during backup, at which point, ransomware can encrypt/delete the files (see [122, 181]). For cloud-based backup systems, such as Dropbox (centralized) and `Syncthing.net` (peer-to-peer), a common issue is the size of a bloated TCB (includes a full OS with multiple network-facing servers), which may lead to large-scale data loss, if compromised.

Leveraging widely-available hardware security features in modern CPUs and hard drives, we shift the focus from detection/prevention/recovery to *data immunization* against rootkit ransomware. If user files could remain unmodifiable by ransomware even after the system compromise,[1] no reactive defense would be necessary—enabling data immunization. Separating read/write accesses of a self-encrypting drive (SED, see Appendix A for background) is an intuitive option, as files in a read-only partition cannot be encrypted. However, when write access is enabled, rootkit ransomware can make malicious changes to the protected files. Therefore, to allow controlled write-access, we also need a trusted execution environment (TEE, see Appendix A).

We design `Inuksuk`,[2] combining security features from SEDs and TEEs to protect existing user files from being deleted or encrypted by malware. Inuksuk functions as a *secure data vault*: user-selected files are copied to a write-protected SED partition, and the secret to allow write-access is cryptographically *sealed* to the machine state (i.e., the genuine intact Inuksuk and the correct hardware platform), and hence, allowing file writes to the data vault only from the trusted environment. Meanwhile access to the read/writable original copy is not affected, and will be synced at the next commit.

We must consider three common scenarios: adding new files, updating and deleting existing files. We treat file updates as new files (i.e., new *versions* of an existing file). Thus for new files and file updates, we commit the changes in the protected partition using our updater program in TEE, without mandating user consent. We limit the number of versions by committing changes in batch, e.g., once every 8-12 hours. Deletion is enabled only in the trusted environment (deleting files outside the environment will fail due to the hardware write-protection), through a mini file browser (manual deletion) or by policy (e.g., automatically delete versions older than a year). Users must be careful when manually deleting (older versions of) a file, to make sure that the kept ones are not encrypted by ransomware, and set the auto-deletion duration with care. Long-running attacks (e.g.,stealthy on-the-fly decryption for the user to hide encryption until ransom is demanded) have limited impact as long as file versions are retained for enough long, e.g., the adversary must delay his ransom for a year; for more discussion, see Section 5.6. Note that, most existing ransomware asks for ransom within minutes after infection [122].

Our assumption is that deletions are done only occasionally and preferably in batch (disk space is relatively cheap). In contrast, file additions (updates included) are more common, but handled without user intervention. No user-level secret is needed

[1]Note: read-only folders/files enforced by the OS (e.g., Windows 10 “controlled folder access” [186]), only prevents unprivileged access.

[2]Inuksuk is an Inuit word with multiple meanings, including: a (food) storage point/marker.

for controlling the write-protection. All original files stay as is and the protected copies can be available (optionally) as read-only in the regular OS.

We choose to instantiate the TEE using Trusted Platform Module (TPM) chips, and Intel TXT CPUs (see Section 5.3.3 for reasons, and Appendix A for background). Due to the exclusive nature (which is also a great security benefit) of the TXT environment, during file operations on the protected partition, the system is unavailable for regular use; as a mitigation, Inuksuk is triggered during idle periods (akin to Windows updates). Also note that over the past few years a series of SMM (System Management Mode) attacks have been identified, some even affecting Intel TXT [297]. All these attacks assume a standard bootloader, hypervisor, or OS being loaded in TXT, where SMI (SMM Interrupts) must be enabled. In our case, we merely load a tiny custom binary with SMI disabled all the time, and TXT being exclusive ensures in hardware that no other code can run in parallel to stealthily trigger SMIs, and thus Inuksuk is apparently immune to these attacks (discussed more in Section 5.6).

While Inuksuk can provide strong security guarantees, its implementation faces several technical challenges. For example, the TXT environment lacks run-time support and we must directly communicate with the SED device (for security) and parse the file system therein (which also involves performance considerations). Note that the use of Intel SGX is infeasible for Inuksuk, as SGX allows only ring-3 instructions, i.e., cannot access the disk without the underlying OS. Also, the user OS is unaware of the TXT sessions, so the devices (i.e., keyboard/display for secure user interface) are left in an unexpected state (see Section 5.4).

Contributions.

1. We design and implement Inuksuk against root-privileged data tampering, in a radical shift in threat model from all existing academic/industry solutions. We target *immunization* of existing data, instead of detection/prevention of malware/ransomware.

2. Inuksuk's design is tied to established and standardized hardware-enforced security mechanisms of SED disks and TEE-enabled CPUs (in our case Intel TXT with the TPM chip). Integrating Intel TXT, TPM, and SED/OPAL together in a seamless way with a regular OS (Windows/Linux) is non-trivial, but offers a significant leap in the ransomware arms-race. Our solutions, which will be open-sourced, to several engineering/performance problems within TXT (e.g., handling CPU caching, DMA, disk/file access, keyboard/display) can also be useful for

other TXT applications.

3. We implement Inuksuk on Windows 7 and Linux (Ubuntu). The core design is OS-agnostic, which is important as ransomware today affects all major OSes. Our prototype achieves decent disk access performance within the OS-less TXT environment (around 32MB/s read and 42MB/s write), when committing files to the protected partition, e.g., once every 8–12 hours. The regular disk access to original files from the user OS remains unaffected, i.e., all applications perform as before.

4. Beyond ransomware protection, Inuksuk can be used as a generic secure storage with fine-grained access control, enabling read/write operations and data encryption (with Inuksuk-stored keys), if desired. Inuksuk is locally accessible without any network dependency, and operates with a small TCB. To provide users with a centralized option, we also briefly discuss a generalized Inuksuk design (in Section 5.3.5), which shifts the TXT/TPM/SED complex to a central location, and protects cloud/enterprise storage against ransomware attacks.

5.2 Threat Model and Assumptions

1. We assume that ransomware can acquire the highest software privileges on a system (e.g., root access or even ring-0 on x86), through any traditional mechanisms (often used by rootkits), including: known but unpatched vulnerabilities, zero-day vulnerabilities, and social-engineering. Root-level access allows ransomware to control devices (e.g., keyboard, network interface), GUI, installation/removal of device drivers.

2. Before deployment of Inuksuk, the user system is not infected by any malware. We primarily protect *preexisting data* at the time of ransomware infection, and provide *best-effort protection* thereafter for later added/updated files until the ransomware is detected (or a ransom is demanded).

3. We do not detect/stop the execution of ransomware, or *identify* its actions. Instead, we protect integrity of user data on a protected partition and ensure data accessibility. If the OS is completely corrupted or inoperable, the user can install a new OS copy or boot from another media (e.g., USB) to access her data.

4. We deal with the most common ransomware variants (i.e., cryptoviral extortion), and exclude those that simply lock access to system resources without using encryption (non-encrypting ransomware [217]) or deletion, and those that threaten to publish information stolen from the user (doxware or leakware [204]).

5. We assume all hardware (e.g., the CPU, TPM and the secure drive) and architecture-shipped (e.g., the Intel authorized code module) primitives are properly implemented by the manufactures, and the user is motivated to choose a system with no known flaws.

6. Attacks requiring physical access are excluded (e.g., no evil-maid attacks). We only consider a computer system potentially infected by malware/ransomware from the network or a removable drive. The system is attended by a user who is motivated to protect her data.

7. We assume that after infection, ransomware will act *immediately*; i.e., it will find target user files, encrypt them, and then demand a ransom without much delay (e.g., few minutes/hours, cf. [122] vs. months). If the attacker waits, he risks losing control, e.g., through an OS/anti-malware update, although users may not always promptly apply patches (vendors may also delay a fix). However, with every patched computer, the attacker loses money, and thus cannot remain hidden for long. To accumulate file updates, the attacker may wait for some time (i.e., long enough to collect sufficient content that the user may care), before asking for the ransom. We term such attacks as *persistent* ransomware, and discuss them more in Section 5.6, item (d).

8. We target user-attended personal computers. Since on PC platforms, ransomware is more of a threat to Windows users [76], we thus consider supporting Windows to be more preferable albeit challenging (we also have a Linux prototype).

5.3 Design

We first list our design goals, then systematically consider design choices and methodically discuss their benefits and drawbacks, and provide a relatively generalized design and workflow. We discuss technical challenges/choices in Section 5.4. The terms ransomware and malware in our setting of unauthorized data alteration can be interchangeably used. We may stick with the use of ransomware hereinafter.

5.3.1 Design goals

We list our goals, and briefly sketch the key ideas to fulfill such goals in Inuksuk.

a) Trusted write-protection. Ransomware must not be able to modify or delete protected files. We place the user files in a write-protected mode (read-only) all the time in the user OS. Write operations to the protected files are only allowed inside a trusted environment. The trusted environment treats all changes as new versions (retaining historical versions) and interacts with the user for infrequent (batch) deletions.

b) Hardware enforcement. Rootkit ransomware should not be able to bypass the write-protection, enforced by the disk, where the protected partition resides, instead of any software on the host. Thus without the appropriate authentication key, the partition cannot be unlocked, even if the OS is compromised (ransomware gains all software permissions). The authentication key (a high-entropy random value, e.g., 256-bit long) is protected by, and bound to, the trusted environment (inaccessible from outside).

c) Minimal application interference. Applications (including the user OS) should operate as is. As the original files are untouched and accessed the same way by applications, normal application I/O is not hindered (even for direct I/O as in disk utilities). File copies on the protected partition are available as read-only, which should not concern regular applications.

d) Minimal user involvement. User experience should not be significantly affected. A normal user experience is preserved in Inuksuk with the separation of the original and protected copies. To reduce system unavailability, the update/commit process should be scheduled during *idle* hours, and all updates to a protected file are cached to be committed as a new version periodically (e.g., every 8–12 hours). The user is involved only when files must be deleted (including, removal of old versions of updated files), and manually triggering Inuksuk (for immediate commitment of cached files, when the important files are just edited/added).

Non-goals. Inuksuk is designed to act more like a *data vault* than a traditional backup system; e.g., we commit user data a few times a day in batches, instead of syncing updates instantly. OS/application binaries should not reside on the protected partitions. We do not target data loss due to system failure or accidental deletion. We provide robust data integrity against advanced attacks at the expense of losing some data due to ransomware attacks (e.g., user updates to a file during the commit period). Also, data confidentiality is currently a non-goal (to facilitate unhindered operations of common applications); i.e., the ransomware can read *all* protected user data, and read/modify the OS/unprotected partitions. However, confidentiality and controlled read access can be easily supported; e.g., encrypting data under Inuksuk-protected keys, and enabling password-based access control for read operations on selected files.

5.3.2 Trusted file versioning

We treat all write operations outside the trusted environment as adding new files (automatically approved, similar to S4 [261]), which poses no threat to existing files, and leaving only file deletion with user intervention. Any committed update to an existing file creates a new version, instead of overwriting the current version (the latest one being under the original file name) so that historical changes committed are all retained in the protected partition. We do not set a limit for the number of versions but leave it to the user to clean up in the mini file browser we developed (see Section 5.5.4 for details), or to configure an auto-deletion policy (e.g., after 1–2 years). Our simple versioning may not impose a significant burden on the storage space, considering: a) We commit changes to the protected partition through scheduled invocation of Inuksuk; users can explicitly trigger the updater to commit important changes immediately, which we believe would be very infrequent. So the number of versions that will be stored for a continuously updated file would still be limited, e.g., 1–4 times a day. Auto-save in applications or file access-time change do not trigger an update (it is only on the original copy). b) Nowadays, disk storage is less costly and user computers are usually over-provisioned. To improve storage utilization, specifically for large files, more aggressive versioning may be adopted (e.g., S4 [261]).

User consent is needed when files are to be deleted. We allow file deletion in the mini file browser within the trusted environment. Users can also set a policy for automatic deletion of old versions (e.g., after 1–2 years). Direct deletion of files in the protection partition outside the trusted environment will be ignored;[3] deletion of the original copy in the unprotected partition will not be synchronized to the protected partition. Regular deletion from the user is also consolidated in the same manner as deleting old versions. Both requires the user to select which file(s) to delete. We also hide old versions from the user OS to help usability. When a new version is

[3]Ransomware sometimes deletes the original file and outputs the encrypted content to a new file, instead of doing in-place encryption.

committed, we rename the previous copy by appending its timestamp with the file name, and keep the new version with the original name.

Automatic stale version deletion. To relieve users from deleting unnecessary old versions of the same file, Inuksuk can be configured to automatically delete such versions after a certain time. The retention duration should be long enough to hurt ransomware's business model. For example, if an attacker needs to wait more than a year to monetize his ransomware, it might become much less attractive than now. Defenders are likely to generate reliable detection mechanisms (e.g., signatures) within the wait period, and even be able to identify the attackers. Calculation of the time duration must be done appropriately, as there is no trusted time source available within TXT. As rootkits can change system time, file creation/update time as available from the user OS file system cannot be trusted. A simple solution could be to use digitally signed time values from an NTP service,[4] where the signature verification is done within TXT. The signed value can be obtained through the user OS, and must be sent for each file commit session. The trusted updater must store the last accepted signed value along with NTP verification keys, and check the new timestamp to detect replay (the time value should always be increasing).

File content verification. Before proceeding to manual file deletion, the user must verify the content integrity of the kept version (for versioning), or the correctness of the file selection (for regular deletion). Note that manual deletion may only be necessary if protected storage becomes almost full. Checking meta-data in the mini browser might be easily bypassed by malware, since it only relies on file structures (in addition to magic numbers). The mini browser is trusted but the file content was taken from the untrusted OS. Namely, malicious modification (e.g., encryption) may be concealed and the user is tricked to commit it (e.g., encrypting only selected chunks of a file). A more comprehensive content verification mechanism in the trusted environment may be achieved by using analytic/machine-learning techniques, at the expense of increased TCB, as well as the risk of false negatives and false positives. We also discuss this issue more under "Delayed attacks after deletion" in Section 5.6. A simple but effective solution is to boot a separate trusted OS (e.g., a freshly downloaded OS image) to check the content (cf. RRD [45]).

[4]See Section 6.2.2 at `http://www.ntp.org/ntpfaq/NTP-s-config-adv.htm`. Alternatively, time-stamping services, implemented by several CAs (following RFC 3161), can also be used.

5.3.3 Design choices

Trusted environment. As we assume ransomware to acquire the highest software privilege, write operations in the protected partition must be performed in a trusted environment, meeting the following requirements: 1) No other applications (including the OS) should be able to manipulate or even observe what is running in this environment. Additionally, this environment should be exclusive and isolated, with anything else suspended, to avoid side-channel attacks. 2) Code running in this environment must have access to I/O (disk, keyboard, video), as it is not safe to delegate operations to untrusted processes outside. 3) The integrity of this environment must be attestable, i.e., it cannot be tampered with, or any tampering would be detected, causing execution to abort.

Intel TXT/AMD SVM (see Appendix A) satisfy all three requirements. The more recent Intel SGX fails requirement 2 as it does not run privileged code for I/O access. Also, as SGX is non-exclusive unlike TXT, SGX suffers from numerous side-channel attacks, including the recent Spectre/Meltdown attacks.[5] Potential hypervisor-based solutions (e.g., running in VMM or a dedicated VM) are susceptible to side-channel attacks [171], failing requirement 1; hypervisors also suffer from zero-day vulnerabilities (e.g., [139]). Therefore, our discussion hereinafter will refer to Intel TXT as our TEE.

Minimal TCB. Although a full-fledged OS in TXT (e.g., tboot with Ubuntu) can be used to perform trusted operations, it is preferable to keep a minimal trusted computing base (TCB), for both auditability (e.g., avoiding numerous complex components) and maintainability (e.g., avoiding measuring large and varying files). It is even not technically feasible, because the trusted operations occur in the midst of an active user OS execution (considering the time/effort needed to save and restore various OS states). In the best-case scenario, such switching will be time-consuming, equivalent to using two OSes "alternately" (as TXT is exclusive). Therefore, we develop our own logic as a small-footprint, native program in TXT with no external dependencies.

Hardware write-protection with API. We need to expose write access to the protected partition only inside the trusted environment. The write-protection has to be hardware-enforced, as rootkit ransomware is able to manipulate any program running in the system. Some off-the-shelf secure USB drives offer write-protection [4].

[5]See e.g., `https://github.com/lsds/spectre-attack-sgx` and more discussion at: `https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/754168`.

However, it is either in the form of a hardware switch/button to be pressed by the user, or a key pad on the USB device itself, where a password can be typed. There is no way for the trusted environment to interact with the write-protection mechanism programmatically.

An ideal construction would be that the hardware write-protection can be enabled/disabled programmatically but with a secret/password. This way, by making that secret only available in TXT, we can limit the exposure of the write access within TXT as well. The self-encrypting drive (SED) satisfies this requirement. Also, to our knowledge, SED is the only technology that supports fine-grained protection ranges with separate read/write permissions, which is important as we always allow read access, and deny write access from the user OS. Fine protection granularity also allows the protected partition to coexist with the unprotected OS and other files in the same drive, instead of requiring a dedicated disk. The legacy ATA Security [263] password can also be considered hardware-enforced write-protection (without media encryption). However, it is a non-solution for Inuksuk, because only one-way locked-to-unlocked transition is allowed (SEC4:SEC5), i.e., relocking requires hardware reset, whereas Inuksuk needs the ability to switch back and forth. No support for co-location of unprotected/protected partitions is another disadvantage.

Separation of the protected partition from the original. Technically, we can write file updates immediately in the protected partition. However, unsolicited/frequent write attempts, such as updates from the automatic save feature in text editors (i.e., not initiated by the user clicking on the "Save" button), will create too many versions in the protected partition and make the system unusable due to frequent back and forth between regular and trusted environments; note that, TXT is exclusive, and writing file updates may also take noticeable time. Therefore, we leave user-selected files for protection where they are, and make a copy into the protected partition on SED. All subsequent updates happen to the original files without write-protection. The user can then decide when to commit changes to the protected partition (no versioning in the original partition), manually, or automatically at certain intervals (e.g., every 8-12 hours).

File-system in TXT. For protected write operations, we cannot simply pass the raw sector information (sector number, offset, number of bytes and the buffer) to TXT as we perform file-based operations, and the user also must select files (not sectors) for deletion. Therefore, the TXT program must be equipped with a file system.

Data mobility. The SED can also contain an unprotected partition where the OS resides, because of the fine granularity of protection ranges, while sometimes users may treat it as a stand-alone data drive. In either case, when data recovery is needed (e.g., the user OS is corrupted or compromised), the user can simply reboot from different media on the same machine or mount the SED on a different machine. The data will be readily accessible as read-only, hence aiding data mobility, thanks to the separation of read and write accesses. In case the user needs to update the files, a rescue USB, where all intact Inuksuk binaries are stored as well as a portable OS can be used to boot the same computer (where Inuksuk was provisioned). After booting with the rescue drive, the user can invoke the same updater in TXT for deprovisioning (to remove write-protection) or regular file access.

5.3.4 System components and workflow

Refer to Figure 8 for an overview of our design. The system consists of the following components at a higher level (more technical details are discussed in Section 5.4):

- *Trusted updater.* This is the core component of Inuksuk, and runs inside TXT. It is responsible for copying files from the original partition to the protected partition (in SED write access mode) as new versions, file listing (in a mini file browser), and showing file meta data to the user.

- *TPM.* In conjunction with TXT, TPM makes sure that the secret (the SED password) is securely stored in its NVRAM storage, and can be *unsealed* only if the unmodified trusted updater is executed (as *measured* in TPM's platform configuration registers).

- *Secure drive.* An SED drive hosts the protected partition. Without the high-entropy key/password, its protection (i.e., write-protection in our case) cannot be bypassed. Note that even with physical access to the drive, reinitializing the drive with the PSID (physical secure ID) printed on it will have all data lost.

- *OS drivers.* A few OS-dependent modules are needed to bridge the user, OS and the trusted updater, such as preparing the TXT environment. These modules do not have to be trusted after initial deployment, as the worst case is a DoS attack; see also Section 5.6, item (a).

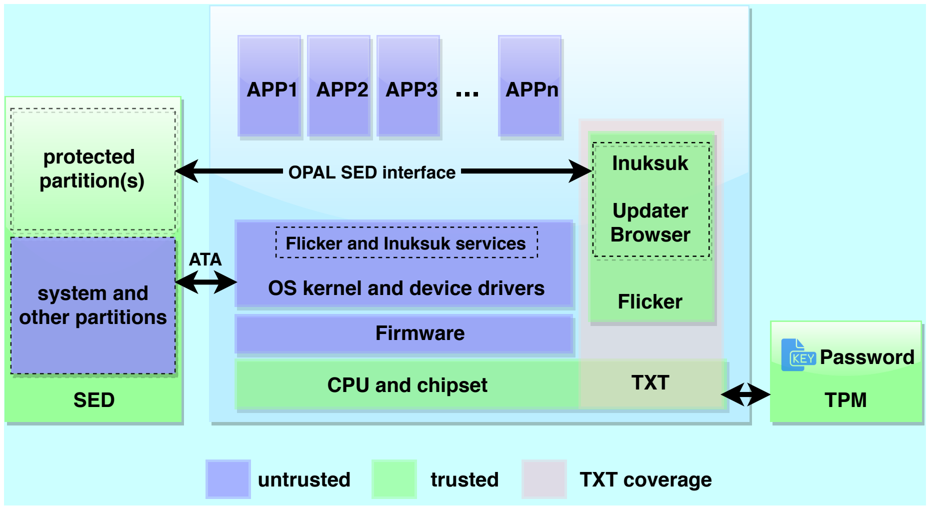


Figure 8: System overview of Inuksuk

Workflow. The generalized workflow of Inuksuk is as follows: (a) At deployment time, a high-entropy secret is generated as the SED password and sealed into the TPM (can only be unsealed in the genuine trusted updater). (b) The protected partition is created with the write-protection of SED. The user also selects the folders to be protected, which are then copied to the protected partition in the first invocation of the trusted updater. After the first-time copying, the user still interacts directly with her files in the original partition. (c) In everyday use, the protected partition is never touched (except for read-only access). As with certain cloud storage services, we use an icon on the original files to indicate which ones are under the protection of Inuksuk. (d) If the user adds or updates files in the original partition and ready to commit her changes, she triggers the trusted updater, and without involving her to verify, changes are committed as new files/versions on the protected partition. The updater is triggered either manually, or automatically, e.g., via scheduled tasks, when the updating-application is closed, or when the system is restarting or shutting down. (e) When the user wants to delete files or old file-versions, she can manually trigger the updater to open a mini file browser, and make the selections (she should be shown necessary file information).

5.3.5 A remote data vault

The functionality of Inuksuk does not rely on any third parties (except the device manufacturer), as the trust is anchored in hardware and all its components are local. Although we primarily present Inuksuk as a stand-alone solution, there is no fundamental barrier in the design for it to be deployed as a remote/networked data vault.

To provide users with a centralized option, as well as extending for enterprise and cloud storage services, we briefly explore a variant of Inuksuk (only the high-level design) where the key components, i.e., TXT CPU, TPM and SED, are shifted to a network location, forming a remote service. Users' data will remain protected at a central, Inuksuk-backed storage service, and users can keep using any device of their choice (i.e., with or without TEE, mobile or desktop). We believe that this variant can be used to protect security-sensitive/user files stored in cloud storage services like Dropbox and OneDrive, or enterprise storage services. Although such services are possibly backed by robust backup measures and strict security policies/tools, if infected, consequences can be high.

The construction goes as follows: any desktop/laptop/mobile device serves as the *front-end* directly used by the user. Through an account, the front-end is connected to a *storage back-end*, which plays the role of the "original partition" in our stand-alone setup, caching file updates. Eventually, an *Inuksuk-equipped backup server*, which has the TXT-capable CPU and chipset, as well as the SED (or more likely, an SED array), is connected with the storage back-end. The Inuksuk-server will periodically copy new/updated files from the storage server, and become unavailable during this period, which should not affect functionality, assuming the Inuksuk-server is not used for other purposes. The storage back-end and user devices remain available all the time.

Note that, in addition to introducing a new trusted party (an enterprise), we do not bloat the Inuksuk TCB, except that the Inuksuk updater must now handle networking (from within TXT) to connect with the storage back-end. Once deployed correctly, without the high-entropy key sealed in TPM, no remote attacker can turn off the write-protection and update/delete the protected files. Our threat model now assumes that the remote attacker can infect the storage and Inuksuk servers, in addition to user devices. As before, only the uncommitted files remain vulnerable, and after written to the Inuksuk protected storage, user files become safe against any data modification attacks. Content on the Inuksuk-equipped server can be maintained by enterprise IT administrators (e.g., deleting old versions). The whole process is transparent to end-users/employees, and the files that need ransomware protection can be identified by enterprise policies.

5.4 Implementation

We implement Inuksuk for both Windows and Linux using existing techniques/tools discussed below. Implementation issues regarding CPU/disk performance are discussed in Section 5.5.

5.4.1 Using Flicker to handle TXT sessions

Since Inuksuk's secure file operations occur alongside the user OS, a mechanism is required for jumping back and forth between the trusted updater and the user OS. It can be implemented as a device driver (in the user OS) dealing with parameters, saving the current OS state, processing TXT logic, and restoring the saved OS state when returned from the trusted updater. Several such operations are already handled in Flicker [183] (also refer to Appendix A), which we use as the base of our prototype. We discuss performance issues related to Flicker in Section 5.5.2.

5.4.2 OPAL access to SED inside TXT

All software outside TXT, including the user OS and all its device drivers, is untrusted in Inuksuk. However, inside the TXT environment, there is no run-time device support, i.e., devices including any SED drive cannot be accessed by default. Therefore, we must implement standalone (and preferably lightweight, to limit the TCB) custom driver for accessing SED devices inside TXT. Various SED protocols rely eventually on the SATA interface (ATA Command Set [270]), with two options to choose: 1) ATA Security password [263]: most SEDs support a user password for compatibility with regular mechanical drives, usually prompted in the BIOS before OS. The entire drive is in a binary state of either unlocked (fully accessible) or locked (fully inaccessible). In this option, SEDs only differ with regular hard drives in that user data is always stored encrypted on the media. 2) The use of dedicated security protocols: such protocols include Seagate DriveTrust [242], IEEE 1667 [124] and Microsoft eDrive (all based on TCG OPAL/OPAL2). They implement support for multiple roles/users corresponding to multiple ranges, with separate passwords for write/read access. They use ATA Trusted Computing features (command TRUSTED SEND/RECEIVE) to transfer protocol payloads.

Granularity in both protection ranges, and separate read/write permissions is important in our design. The same drive can host both protected and unprotected partitions (which cannot be achieved in Option 1). Thus Option 2 is more suited for our needs, and we choose to use TCG OPAL to communicate with SED, as it is an open standard and widely supported by most devices.

There are a few open-sourced tools that can manipulate SED devices with OS support (in addition to proprietary tools for vendor-specific protocols); we have tested msed [222] (now merged into DTA sedutil [71]) and topaz-alpha [3]. They mainly rely on the I/O support from the OS, e.g., SCSI Generic I/O, in the ATA passthrough mode. However, our TXT *piece of application logic* (PAL, the payload in Flicker) is OS-less with no run-time support. We decide to port functions from topaz-alpha [3] as needed. The porting process faces several engineering challenges, see Section 5.4.4.

More details about the OPAL protocol can be found in the TCG specification [276]. Overall, OPAL communication involves level0/1 discovery (protocol handshake to agree on version and parameters), logging into a session using the corresponding password, and manipulating the tables in SED to set permissions (locking/unlocking).

5.4.3 Secure user interface

For file selection, we must provide UI for the user to interact within the trusted environment; note that, during this time, the entire user OS remains suspended. Providing secure UI is critical as the user may make wrong selection based on false information (if the TXT-to-user channel is compromised), or simply the user selection is forged (if the user-to-TXT channel is compromised), leading to arbitrary data of the adversary's choice deleted from the protected partition. As we assume hardware is always trusted (see Section 5.2), we discuss only the software part of these UI channels. We consider the following options for using the *frozen* display (from the user OS):

1. Switching back to 16-bit real mode and resuming 32-bit protected mode after calling Video BIOS for display. This simple approach is ideal for small footprint and for infrequent switching. However, Intel TXT works only with protected mode, making this approach infeasible.

2. Using the Virtual 8086 (v86) mode to invoke Video BIOS. The v86 mode is supported by the CPU (providing separate TSS for tasks) but the software developer must write a v86 monitor (like a VMM) and corresponding components handling interrupts, I/O, and so on. We exclude this considering its complexity and compatibility issues.

3. Developing a custom (preferably, universal) display driver for the TXT session. This involves porting only I/O operations, and does not have the two drawbacks above. But it would be inevitably vendor-specific, as before entering TXT, the OS has already set the video card in a state unknown to us (we tried a few sequences but could not reset it to the legacy VGA mode). It may be technically possible on certain models but not guaranteed to work.

4. With the aforementioned three options ruled out, we propose to use a different but effective method. We ask the entity that knows well the video card (which is Windows in our case) to reset it to the legacy VGA mode, without trusting it (other than the possibility of a DoS attack), and then use our own logic inside TXT to take it over and display the content. This approach preserves both compatibility and compactness.

To realize Option 4, we resort to a set of (less-used) Windows kernel APIs, e.g., the x86 BIOS emulator [50], and develop an OS driver. These APIs use the v86 mode (as in Option 2) to call BIOS functions. We modify the Flicker Windows driver for loading PAL, so that right before entering TXT, the display is reset to legacy VGA using `x86BiosCall()`, and after exiting TXT, we restore it to the previous high resolution via the same function. This way, inside TXT we can manipulate the display as if the system is just freshly booted. In our Linux prototype, we use `vbetool` and `mode3` that can make use of the Video BIOS to set VBE modes similar to Windows. We use a custom sequence of commands to resume display (details omitted for brevity).

DMA in TXT. Currently, USB keyboards are the norm, but they are non-trivial to support within TXT. Unlike other simpler protocols, the controller (e.g., EHCI [281]) requires several host-allocated buffers in the main memory (DMA chunks) for basic communication with the host (e.g., the periodic frame list). The controller accesses the buffers without the CPU's intervention, hence, Direct Memory Access. However, the fundamental protection of TXT (like all other trusted execution environments) must prevent autonomous access from peripherals. The MLE memory is included in either the DMA Protected Range (DPR) or Protected Memory Regions (PMRs), which is mandatory (cf. [130]).

Consequently, since we cannot (and do not want to) exclude the MLE from DMA protection, we have to allocate the USB DMA chunks outside. We do not consider possible security implications of exposing DMA regions outside the MLE in general;

however, in our specific case where physical attacks are excluded and no other code is running in parallel, doing so does not pose a threat. We also support PS/2 keyboard.

In Flicker, the PAL program is assigned Ring 3 with confined memory access only to the MLE (TXT-measured region), justified for security reasons. The base address in its GDT descriptor is set to the start of the MLE region and addressing in it will be offset by the MLE start. To preserve this design, we adapt all MMIO access functions (such as reading/writing EHCC registers) with in-line assembly to temporarily switch the data segment (DS) to a global one, covering the whole address space. With the small tweak, USB with DMA can work transparently in Inuksuk.

5.4.4 OPAL implementation challenges

From C++ to C. All OPAL projects are written in C++, possibly to support complex data structures. However, Flicker PAL provides only an environment in C. We weigh the difficulty between adding C++ support to Flicker and porting a few topaz functions to TXT in C, and eventually choose to go with porting.

However, it is not a straightforward syntactical conversion process. For instance, C++ provides built-in heap management (`new, delete`), and it is ubiquitously used in topaz-alpha. Flicker comes with basic `malloc()/free()` but lacks other essential functions. We had to implement `malloc_size()` by counting adjacent slots so that the size of a dynamic object can be known, which is critical for OPAL structures, e.g., current packet/vector size. Also, `malloc()` in Flicker uses a fixed range of memory (to be measured in TXT) to avoid affecting the suspended OS, and thus we had to regularly reset the heap due to space constraint (discarding all allocated objects and setting heap utilization to zero).

Moreover, the C++ constructor/destructor mechanism invokes custom initialization/cleanup automatically, while we cannot manually trace the life cycle of all objects, especially when they are hierarchical/recursive. So we made quite some trade-offs such as calling an `init()` function after each declaration, and where a next-level pointer (e.g., in the case of a member vector in a struct) is potentially de-referenced.

We also need a C equivalent to `std::vector` in C++, and an important criterion is the support for continuous element addressing, e.g., a packet received into a byte buffer will form a byte vector with each byte becoming an element. For this purpose, we use SCV [123].

OPAL data stores. OPAL communication is in a human-readable format (instead of binary, like ATA/TPM commands), and thus requires complex parsing and packet construction. For example, it has the notion of `atoms` (used to encode data of various sizes and types, in the form of tiny, short, medium, long or empty). Two constructs, Named (e.g., 'MaxPacketSize' = 66028) and List (e.g., [e1,e2,...,ei]), can be used to represent bytes, integers, and even parameter structures; at the storage level, these constructs are composed of atoms. Fortunately, we were able to borrow certain logic from topaz-alpha for data parsing and packet construction.

5.5 Performance Considerations

In this section, we discuss certain performance issues for Inuksuk; our solution techniques can also be useful for other OS-less I/O intensive TXT applications. We perform our development and evaluation on an Intel Core i7-2600 @3.40GHz, 16GB RAM (3GB is usable in 32-bit mode), and Seagate ST500LT025 SED disk. We implement Inuksuk for both Windows 7 and Linux/Ubuntu 12.04.

5.5.1 File system efficiency

As discussed in Section 5.3, we choose to handle updates to the protected partition at file-level instead of raw sectors. This requires at least basic file system functionalities implemented within TXT. As a first step, we explore several open-source FAT32 projects for easy portability to TXT, instead of implementing the entire specification on our own. We exclude FAT32 projects that are tightly coupled with external dependencies, e.g., the FAT driver in Linux interfaces with `inodes` (Linux VFS). FAT32 implementations targeting embedded systems are more fitting for our purpose, but several other factors must be considered. For instance, ThinFAT32 [262] appeared to be a good fit, as it is lightweight, written purely in C, and requires no dynamic memory allocation; nevertheless, it lacks support for the deletion flag `0xE5` (i.e., deleted entries are not recognized, which we patch easily), and more critically, buffering (resulting low performance). Important features that we need in a lightweight FAT32 implementation, include:

1. *Buffering support.* Usually, FAT32 access is sector-wise, and most block devices are also accessed in sectors. However, an I/O request can involve multiple sectors (specifying start sector and number of sectors to read/write), offloading I/O processing from CPU to the DMA controller. For PIO modes, it does not make

much difference to send one bulk request vs. many one-sector requests, as the CPU handles all sectors one by one. It is essential for DMA (see Section 5.5.3) to handle a certain number of sectors for performance. If only one sector is requested, the overhead will be significantly high, and thus no performance is gained with DMA (as in the case of ThinFAT32). Usually the FAT32 implementation exposes the low-level read/write interface to the developer, and if it does not support buffering, the read/write functions we implemented will always receive one sector to read/write. Note that hardcoded pre-fetching for reads is an overkill (reading data never needed), and hardcoded write buffer will hang (waiting for enough number of sectors).

2. *Multi-cluster support for space allocation.* At the file creation time, and when a file grows in size, FAT32 must traverse all clusters to find free clusters to be appended to the cluster chain of the file. Interestingly, with all FAT32 projects targeted for embedded systems that we tested, only one cluster is allowed to be added (we do not see any performance problem for allowing multiple). Therefore, for a 50MB file taking 6400 clusters (8KB cluster-size) and the partition having 131072 free clusters (1GB), it takes more than 800 million iterations. In the end, with DMA enabled but no support for multi-cluster allocation, the time needed for file creation is about 80 times slower than overwriting an existing file (requires no searching).

Our solution. We tested several libraries, including fat_io_lib [1], ThinFAT32 [262], fedit [78], efsl [308], etc. Unfortunately, none of them support both features; e.g., fat_io_lib has buffering but no support for multi-cluster allocation, and efsl supports multiple clusters but is deeply rooted in single-sector disk read/write. We choose fat_io_lib [1] for adaptation, because of its good buffering performance. For each iteration, we start with the cluster where we left off, instead of the first cluster of the partition. We emphasize that Inuksuk is not dependent on any specific file system, and thus FAT32 can be replaced with a more efficient one.

5.5.2 CPU slowdown in Flicker PAL

Our initial prototype was extremely slow: it took more than 23 minutes just to copy a 50MB file. This first led us to doubt the performance of the FAT32 library we used. However, when tested with only a 1-million-iteration loop with NOPs (i.e., no operations), we found it to be about 500 times slower within Flicker, compared to Ubuntu/Windows. By putting this loop in different places in both the Windows Flicker driver and the PAL, we finally found that the slowdown starts right after the Flicker driver updates the Memory Type Range Registers (MTRRs) in preparation for the PAL. MTRRs control the caching properties of the specified RAM regions, i.e., whether caching is allowed for a memory range and how specifically, e.g., uncachable or writeback/writethrough.[6] The SINIT module for TXT mandates that there is a dedicated MTRR entry for it (WB for writeback), and the rest of the memory must be set to one of the supported memory types returned from GETSEC[PARAMETERS] [130]; otherwise, TXT will crash.

Flicker first saves all MTRRs for Windows, creates the SINIT-only entry overwriting the MTRRs, and after the PAL execution finishes, Flicker restores the Windows MTRRs. This sequence is similar to Intel tboot, i.e., tboot saves the boot-time default MTRRs before it runs the TXT MLE, and restores MTRRs before loading the OS (e.g., Ubuntu) so that the OS still sees the boot-time MTRR state; in between within the MLE, policy enforcement is executed, which is not CPU-intensive. In Flicker, its PAL replaces the tboot MLE, and the MTRR values saved/restored are those of Windows. It is not possible to restore the values earlier (before resuming Windows) although the actual payload resides in the PAL. We suspect that at the time Flicker was designed, caching may have not played a major role in the processor's performance for the tasks tested, and thus the slowdown was not as noticeable as ours now (more than 500 times). We verified this behavior by contacting a Flicker author.

Our solution. The first attempt was to keep the same MTRR save/restore sequence, and explore the supported memory types in parallel with SINIT. However, on our test machine, the returned memory type flags corresponded to UC (uncacheable) and WC (writecombining); WC is still extremely slow (as per the 1-million-iteration loop: WB=0.57ms vs. WC=295.61ms). Even if other machines support better types, it will still be machine-specific. Since the restriction on supported memory types is relaxed after SINIT execution (but still before the PAL starts in MLE), we define a second MTRR entry with WB after the SINIT one, to cover the region of the PAL, where our trusted updater resides. Thereafter, the performance is restored, i.e., the execution of the NOP loop takes almost the same time as in Ubuntu (or even faster, because of no multitasking).

[6]See Section 11.11 at: `https://software.intel.com/sites/default/files/managed/7c/f1/253668-sdm-vol-3a.pdf`

5.5.3 Adding support for DMA disk access

After addressing the CPU slowdown issue in Flicker, we have a throughput at roughly 0.5–1MB/s, as we perform disk data transfer through regular port I/O (i.e., without DMA). This speed is expected according to the theoretical speed of ATA PIO modes [7] (also with the overhead of FAT32 logic). However, this is unacceptable from the user-experience perspective, e.g., taking 3–4 minutes to write a 100MB file.

To improve disk access performance, we implement ATA DMA access support inside the PAL. The effort is mainly twofold: (1) setting up the Physical Region Descriptor Table (PRDT) so that the ATA DMA controller knows where to place/retrieve data blocks; and (2) issuing the DMA-version of the SEND (25h) and RECEIVE (35h) ATA commands, and especially accommodating them to the lightweight FAT32 library source code. Usually, DMA relies on interrupts, i.e., when the transfer is done, the interrupt handler will be notified to proceed to the next request (e.g., to maximize CPU time utilization in a multitasking environment). In our case, Flicker is not supposed to work with an interrupt-enabled workload (technically possible with some complex adaptation), and we merely need the performance boost through DMA, i.e., no multi-tasking and thus, requiring no interrupt support. In the end, with DMA enabled, we made file transfer in our trusted updater 50–100 times faster than using just PIO (see Section 5.5.4).

Again, pertaining to the discussion in Section 5.4.3 (DMA in trusted environments), we also need to allocate the PRD tables (DMA regions) outside the MLE (no measurement or I/O protection) for the ATA DMA controller to be able to access them. For the same reason, there is no risk in doing so within our threat model.

5.5.4 Usage scenarios and performance

In this section, we discuss several human factors considered for usability in the design of Inuksuk, and our approach to keep the user experience at an acceptable level in different usage scenarios. Then, we provide performance statistics of the implemented prototype.

Disrupting regular usage. In a regular usage scenario, we have a major source of disruption: the unavailability of the computer for regular tasks. Depending on the file size (and number), for some time the user cannot use her computer during file copying (from original to protected partition), since the TXT session is exclusive.[7] We consider the following file operations, and explain how they are affected by Inuksuk from a usability perspective.

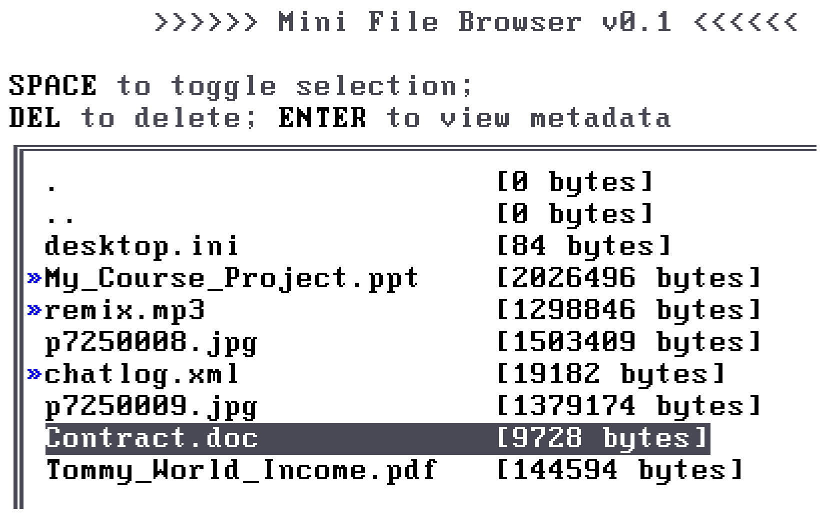
- *Adding new files.* New files can be added by a user or ransomware. However, maliciously creating new files poses merely a DoS attack on storage. Thus, no user consent is needed for new files; they are simply copied to the protected partition when Inuksuk is triggered by the user or after a certain interval (e.g., once every 8–12 hours). Similar to Windows Updates, the user can define her idle/busy hours so that the trusted updater can execute during idle hours.

- *Updating existing files.* With versioning, each update is treated as adding a new file, and requires no user consent. Note that the frequency of triggering the trusted updater must be set with care: it affects both system unavailability, and the number of versions saved for existing files. We group several save events (auto/user-triggered) from applications in the original partition to reduce switching back and forth from TXT; see also Sections 5.3.2, 5.3.3.

- *Deleting existing files.* We have developed a light-weight file browser inside the trusted updater that allows the user to choose multiple files for deletion; see Fig. 9. With more engineering effort, a graphical interface can also be created. There is no technical limitation of creating custom UI within TXT. Also, deletion may involve only flagging the files as deleted in the file system, and thus should be quick. Note that users can manually delete files in the protected partition only from our mini browser; also regular *Recycle Bin* functionality is preserved. Any deletion attempt from the user OS will fail (read-only access to the protected partition), or be ignored, if edited files are removed from the original partition (the updater only copies existing files from the original to the protected partition). Older versions of an existing file can also be scheduled for auto deletion (e.g., after a year). For regular home use with a decent disk size (e.g., 2TB), we believe users may need to manually delete files only very infrequently.

Performance evaluation. The file transfer speed determines the unavailability of the user computer, and affects user experience. However, we argue that the way we implemented DMA and our choice of the FAT32 library (as well as our adaptation to

[7]We also cannot simply initiate the copying process in TXT and return to the normal OS (to complete the bulk copy), as that will expose the protected partition to ransomware.



```
              >>>>>> Mini File Browser v0.1 <<<<<<

SPACE to toggle selection;
DEL to delete; ENTER to view metadata

  .                            [0 bytes]
  ..                           [0 bytes]
  desktop.ini                  [84 bytes]
 »My_Course_Project.ppt        [2026496 bytes]
 »remix.mp3                    [1298846 bytes]
  p7250008.jpg                 [1503409 bytes]
 »chatlog.xml                  [19182 bytes]
  p7250009.jpg                 [1379174 bytes]
  Contract.doc                 [9728 bytes]
  Tommy_World_Income.pdf       [144594 bytes]
```

Figure 9: A screenshot of the mini file browser inside the trusted updater. Selected files are designated with "»"; group selection can be specified by the first and last files. Metadata includes full file path.

it) are confined by the engineering effort and time. Therefore, the numbers we show here should be just the lower bounds.

We executed 10 measurements on the files we selected with typical sizes; see Table 3. 50MB represents common media files (the same order of magnitude) and 500KB represents miscellaneous files of trivial sizes. Note that without our adaptation to enable multi-cluster allocation support, the creation of a 50MB file can be done only between 0.5–1MB/s, while overwriting of an existing file of the same size runs at about 40MB/s.

To demonstrate Inuksuk's performance in a realistic usage scenario, we invoke the trusted updater to copy 50 random photos (JPG files, size ranging from 1009KB to 2416KB, totaling 85.6MB) from the original partition to the protected partition. We measured the duration for 10 times, and the performance seems reasonable (mean: 23.3853 seconds), and relatively stable (standard deviation: 0.58989). This is a combination of read, write, file opening/closing, accumulating space fragments, etc. We also evaluated only the transition time between the OS and trusted updater. It varies between 2–4 seconds, including screen mode switching.

If we take into account any extra processing during file transfer, the time needed may also be affected. The basic versioning Inuksuk uses is not *incremental*, i.e., the whole of the source file in the original partition is copied over to the protected partition as a new version. We may consider some open-source version control systems like SVN/Git (or even the simple `diff` command) for incremental versions to save

		Write/Existing	Write/New	Read
50MB file	Mean (MB/s)	43.93	41.69	32.17
	Standard deviation	3.40	0.31	0.09
500KB file	Mean (MB/s)	26.46	8.09	16.67
	Standard deviation	1.18	0.43	5.26

Table 3: File transfer performance in the trusted updater from 10 measurements. For small files (e.g., 500KB), other overhead predominates the transfer time.

disk space. However, in that case, each time new files/updates are committed, the updater must scan the whole of both files for differences and then perform the transfer. Moreover, deletion is supposed to be very quick with non-incremental versioning (just flagging the file); with diff-like versioning, for each file the updater has to reassemble from all previous versions to form the latest one to be kept. The overhead could be significant in our setting (considering batch-deletion of versions). Also, for common file types such as images, videos, and rich documents (e.g., PDF, Word), incremental versioning may not save much disk space.

5.6 Security Analysis

In this section, we list various potential attack vectors, and discuss how they are addressed, or why they do not pose a threat (see Section 5.2 for our assumptions).

Since we shift the defense from detection/prevention to data immunization, we avoid common attacks such as whether new ransomware can evade detection, whether it does privilege escalation, how the encryption keys are generated and so on. There are two basic questions in evaluating Inuksuk's effectiveness: 1) Outside the trusted environment, can ransomware update files in the protected partition? No, without the high-entropy key sealed in TPM, software on the host system cannot break the write-protection enforced by SED. 2) Inside the trusted environment (updater), can the ransomware trick the user or the updater to write arbitrary content of his choice? The updater does not synchronize any file deletion from the original partition but only adds files from it. With the updater's integrity ensured by TXT, user I/O cannot be influenced by any external software.

(a) Malicious termination, modification or removal of Inuksuk. A simple but effective attack against Inuksuk is terminating its kernel driver in the OS, or even completely removing it. Similar to rootkit malware's termination of host-based anti-malware defenses, rootkit ransomware can easily launch this attack against Inuksuk.[8] The pre-existing files in the protected partition remain immune to this attack; however, newly created or updated files thereafter are not protected. However, unlike anti-malware defenses, which run mostly transparent to users, users are expected to interact with Inuksuk, albeit infrequently (e.g., manually triggering Inuksuk for committing new/updated files, and file deletion), and thus may notice when Inuksuk cannot be launched. When the ransomware's presence is apparent, users can take other mitigating actions, e.g., reinstallation of OS/software, updating OS/anti-malware signatures. Note that modifying the Inuksuk updater's binary, which may reside in the unprotected partition, does not help the attacker; the SED unlock secret can only be accessed by the genuine Inuksuk updater (TPM unsealing).

(b) Known attacks against SEDs. Müller et al. [193] show that SEDs are also vulnerable to known attacks against software FDEs (e.g., cold boot, warm boot, DMA, and evail-maid). They also found a simple attack called *hot plug*, enabled by the fact that SEDs are always in a binary state of locked or unlocked. Once it becomes unlocked in a legitimate manner (e.g., user-supplied unlock passwords), the adversary can connect the disk to another attacker-controlled machine without cutting power, and can get access to protected data. In addition to these attacks, an adversary may also capture the cleartext SED secret/password from the SATA interface, e.g., by tapping the connection pins with a logic analyzers. Since all such attacks require physical access, they are not viable for a scalable ransomware attack. However, if a software-only attack can bypass SED protection (e.g., unlock a partition without supplying the corresponding secret), Inuksuk will be defeated. No such attacks exist thus far.

(c) Attacks on TXT/TPM. Although TPMs offer some physical tamper-resistance, TPMs and similar security chips have been successfully attacked in the past (e.g., [147, 266, 255, 164, 294]); see also Nemec et al. [203]. However, with physical access excluded, we do not need to consider these attacks; also note that tapping TPM pins and DMA attacks require a malicious device to be connected. Regarding known software-only attacks against TXT, most such attacks are ad-hoc (e.g., the SINIT module flaw [299]), or version-specific; Intel has purportedly patched them in the subsequent versions, or at least the user is motivated to choose one that has no known flaws.

[8]Malicious termination can be made difficult by registering Inuksuk as a Windows Early Launch Antimalware (ELAM) driver.

There are also attacks against TXT (e.g., [297]) that exploit the System Management Mode (SMM), an intrinsic part of the Intel x86 architecture, referred to as Ring -2. Normally if we assume anytime before entering TXT, the system (including BIOS) is compromised, and SMI (SMM interrupt) is left enabled when entering MLE, a malicious SMI handler can always preempt the TXT execution and intercept any trusted operations. However, there must be certain code triggering SMI (manipulating physical SMI# pin is out of scope), e.g., writing to port 0xB2. This could be from an OS, hypervisor or bootloader loaded in TXT, which must have SMI enabled but has compromised/vulnerable code. In the case of Inuksuk, neither of the two factors satisfy: 1) SMI is not needed in our custom code – the trusted updater. So we just do not trigger it. Because of TXT's exclusiveness, no other code can trigger it either. 2) We can also disable SMI upon entry, leaving no time for any triggering. Another possible (powerful) attack avenue similar to SMM is vulnerable Intel Management Engine firmware [75]. Unless there is a pressing need for ME, we suggest to disable it in a rigorous manner (for efforts and difficulties, see [236]).

(d) Delayed attacks after deletion. Persistent ransomware can stay hidden for a long period (ranging from weeks to months), during which it just transparently decrypts encrypted data when accessed [120]. This can trick the user to believe that her data is intact (when viewed from within the OS). At some point, if she removes older versions to save space, then the ransom can be demanded (i.e., no more showing the decrypted version).

The root cause of this problem is that OS-based file viewers (e.g., Microsoft Word), run outside the trusted environment and can be manipulated by rootkit ransomware arbitrarily, such as performing decryption before displaying a file to the user, or simply feeding a cached, unencrypted copy of the file. A straightforward countermeasure is to perform verification inside the updater before removing previous versions, e.g., by porting advanced file viewing tools in TXT, which can require significant effort. A simpler approach is to use (imperfect) encryption detectors and other ad-hoc mechanisms (e.g., [5]) as used in other solutions (e.g., [149, 150]). Another possible way would be to boot from a trusted OS image (e.g., USB) to check files before deletion (inefficient, but needed only occasionally). If auto-deletion of old versions is enabled, we suggest the duration should be long, e.g., a year or two, depending on the size of the protected partition. Note that, delayed attacks risk being discovered and mitigated by anti-malware vendors, and thus we do not consider them a serious threat.

(e) Hiding/locking access to files. Inuksuk defends against ransomware based on data inaccessibility (e.g., erase or encryption). However, another variant of ransomware, which is more prevalent in mobile platforms, is locking-based (non-crypto). It simply blocks the user's access to computer resources, e.g., by switching to a blank desktop using `CreateDesktop()`, or by showing a persistent HTML page [151]. Since it mainly targets non-computer-savvy users, simply showing a screen PIN lock on the mobile phone may lock the user out and suffice for demanding ransom. This can also be done on top of encryption, to make locating files non-trivial (e.g., overwriting the master boot record). Such ransomware, although still worrisome for non-computer-savvy users, is relatively easier to cope with. With proper tools, recovering data is highly possible (e.g., by reverting the system changes), as the original data is preserved.

(f) Forged user interface. Due to human users' inability to authenticate machines (cf. Stark [195]), a common means of attack by rootkit malware is to mimic the appearance of the intended program/tool, where the user can be tricked to leak secrets. However, the adversary will not benefit from it, as there is no UI in Inuksuk for prompting for the SED unlock secret (in fact, the unlock secret is unknown to users). Also, for manual deletion, there is no way to specify which files to delete from outside the trusted updater (files are selected in TXT right before they are deleted). In the end, without the genuine updater in TXT, the adversary cannot manipulate any file in the protected partition.

(g) Attacking auto-deletion. If auto-deletion is enabled, i.e., older file versions are automatically deleted after a preset threshold (e.g., 365 days), a straightforward threat is clock source manipulation. Rootkit ransomware can adjust the system time (to a far future date) to fool Inuksuk to believe the versions are already too old to be kept. To address this, Inuksuk can be configured to only trust a signed NTP time from a remote server, absence of which will stop auto-deletion (see Section 5.3.2).

5.7 Related Work

There are many solutions dealing with user-level ransomware; only FlashGuard [122] targets rootkit-level ransomware. However, some solutions against data manipulation by rootkit malware (not specific to rootkit-level ransomware) are close to Inuksuk in spirit. We discuss several examples from each category.

Rootkit-level solutions. S4 [261] is proposed as a self-securing storage entity behind a security perimeter, which records all file operations (like journaling or auditing) and retains old versions of user files. It is implemented as a network service (similar to NFS), and assumed to be resistant to compromise by a remote party (due to S4's limited outward interface). The usage scenario is focused on intrusion survival and forensics collection, in the case of an admin account compromise in a client machine. As S4 promptly stores all changes made to the client machine, as soon as possible, its storage overhead can be significant. To address this challenge, S4 makes use of novel compression and differential versioning techniques, which can benefit Inuksuk as well. Also, relying on a network service is problematic for various reasons; e.g., it can be made unreachable from the client machine by the rootkit, not easily deployable for home users, and involves a large TCB, including a full OS and network-reachable servers. Moreover, if the admin account of S4 (or any similar backup system) is compromised, large volumes of data may be lost at once.

FlashGuard [122] proposes to modify the garbage collection mechanism of SSD firmware (assuming vendor support), so that for *suspicious* overwrites (i.e., first read and then written in a quick succession), a copy of the original data block is kept for a preset amount of time (e.g., 20 days). FlashGuard leverages a unique *out-of-place write* feature of modern SSDs (in contrast to regular hard drives), which provides an implicit backup of recently overwritten data blocks. The user is expected to detect any attack before the preset time elapses and perform the recovery from a separate machine; otherwise the data will be lost. The detection of suspicious overwrites can be an issue; e.g., ransomware can read and encrypt the file, and at some later point (i.e., not immediately to avoid being flagged), delete the file. However, this can solved by retaining *all* deleted data blocks, at the expense of increased storage overhead. FlashGuard authors also do not specify the clock source to measure the preset time; SSDs do not offer any trusted clock, and relying on OS/BIOS could be fatal.

Rootkit-resistant disks (RRD [45]) are designed to resist rootkit infection of system binaries, which are labelled at installation time, and write operations to protected binaries are mediated by the disk controller. System binaries are updated by booting into a safe state in the presence of a security token. While effective against rootkit infection, RRD is infeasible against ransomware that targets regular user files (adding/updating will require reboot). Inuksuk's goals are complementary to RRD's and exclude protecting system binaries.

User-level solutions. Defenses are usually implemented as system services, kernel drivers (unprivileged adversary), or even user-land applications. For instance, Redemption [150] explicitly mentions that their TCB includes the display module, OS kernel, and underlying software. Redemption claims to provide real-time ransomware protection, by inspecting system-wide I/O request patterns. Its detection approach involves a comprehensive list of features, with both content-based (entropy, overwriting and deletion) and behavior-based (e.g., directory traversal). In the end, a malice score is calculated to facilitate decisions. Redemption creates a protected area, called *reflected file*, which caches the write requests during inspection; the file is periodically flushed to disk (if no anomaly is identified). This ensures data consistency in case of false positives, i.e., if the suspicious operations is confirmed by the user to be benign, there is still the chance to restore the discarded data.

In an effort to achieve better universality and robustness, some proposals are purely data-centric (i.e., agnostic to program execution, checking just the outcome). E.g., CryptoDrop [239] focuses on file transformation information for individual files, regardless of where those transformations come from. It also claims to achieve early detection. It employs three novel indicators to detect suspicious file operations. Low file similarity before and after may indicate encryption but legitimate operations can also cause it (e.g., a blurred JPG file). Shannon entropy can be used in detecting encryption although compression also leads to high entropy. Last, file type changes (through content parsing) might not be robust enough with format-preserving encryption [257].

Although most ransomware mitigation techniques aim to detect/prevent ransomware as the primary goal, very few also focus on recovery, e.g., PayBreak [159]. Symmetric keys used by ransomware to encrypt user data are captured through crypto function hooking before they are encrypted with the adversary's public key, and then stored in a secure key vault. When infection is detected or a ransom is demanded, the user can retrieve the keys for decryption without paying the ransom. PayBreak's crypto function hooking works for both statically and dynamically linked binaries, but only if the ransomware uses known third-party crypto libraries. Also, it is subject to evasion by obfuscation for statically linked ransomware. The key vault, even though encrypted with the user's public key and protected by the admin privilege, can still be easily erased by rootkit ransomware.

ShieldFS [60] is a copy-on-write shadowing filesystem reactive to ransomware detection, which is also based on I/O requests (I/O Request Packets - IRPs). Its methodology fits in the intersection of recovery-based solutions and immunization, and thus is similar to Inuksuk in positioning. The detection portion also makes use of numerous behavioral features reflected from the IRPs. Specifically, ShieldFS's cryptographic primitives detection, different from PayBreak's, does not rely on hooking known crypto libraries, but captures inevitable properties of crypto primitives, such as the key schedule pre-computation of block ciphers. To achieve the claimed self-healing, on the first write attempt, ShieldFS keeps a copy of the original file in a protected location (only from userland processes); once an anomaly is detected, the changes made can be reverted with this copy, or otherwise it can be deleted at any time.

Microsoft BitLocker [300] is a widely-used (enterprise) data protection tool integrated with the Windows OS. BitLocker provides strong confidentiality guarantees through TPM-bound encryption. However, when a BitLocker-protected partition is unlocked after a successful boot (i.e., accessible to the OS and applications), there is no way to distinguish a malicious write attempt from legitimate ones, and thus making the protected data vulnerable to even user-level malware/ransomware attacks.

For advanced data protection in iOS, Apple's secure enclave co-processor (SEP [175]) is also a form of hardware-enforced security feature, enabling memory encryption and credentials management (among other functions). The SEP communicates with the application processors (APs) via a mechanism called Secure Mailbox. From the limited public documentation, it appears that per-application access control is possible with SEP, therefore, decryption (and thus updates) can be only exposed to the right application.

5.8 Conclusions

In summary, we propose the notion of data immunization, in an effort to address rootkit-level data alteration as exemplified by ransomware, a significant threat that remains largely unaddressed in current state-of-the-art solutions. We leverage both trusted execution environments in modern CPUs and hardware-enforced write-protection in self-encrypting drives. Inuksuk leaves original user files in use with applications and exposes the protected copies as read-only all the time, and silently accepts creation/modification of the files by preserving previous versions. Users are only involved in file deletion occasionally in the trusted environment (e.g., once every year or so, in case the protected partition becomes full). Although our current prototypes are less than ideal (file transfer performance, Flicker and Windows specific issues), we believe Inuksuk is a solid step towards countering rootkit ransomware.

Chapter 6

COTS One-Time Programs

Protecting data (e.g., confidentiality and integrity) has been our primary goal. Using also trusted execution techniques, in this chapter we explore options for ensuring execution integrity and contribute to the practicality of one-time programs (referred to as OTP hereinafter).

Co-authorship. Parts of this chapter have been co-written with other students and professors from Concordia University, University of Florida, and Case Western Reserve University.

6.1 Introduction

Consider the well-studied scenario of two-party computation: Alice and Bob want a function computed that includes their own inputs, but they do not want to disclose these inputs to each other (only what can be inferred about them from the output of the computation). This is traditionally handled by an interactive protocol between Alice and Bob and much cryptographic literature has been devoted to its study (see [109] for a recent textbook). In this chapter, we study a variant of this protocol that is non-interactive. Alice prepares a device for Bob with the function and her input included. Once Bob obtains this device, he can supply his input and learn the outcome of the computation. Alice might be a company selling the device in a retail store, and Bob is the customer; the two never interact directly. Bob uses the device offline, and thus is assured that his input is private.

Some immediate issues arise. The first is: how can Alice be sure her input, which must be somehow encoded into the device, cannot be extracted by Bob? Second, even if Bob cannot directly extract it, he could indirectly infer her input by running the computation on many different inputs. We provide a practical implementation of a device that resists attempts at removing Alice's input and can only be executed by Bob on a single input. We also note that one-time programs can be realized with a much simpler primitive, called one-time memory or OTM (as pointed out in the cryptographic literature [92]), whose essentials are also rooted in one-timeness. We are inspired by the intuition that 1) one-timeness is still a form of execution logic; and 2) if we can make use of hardware primitives to enforce such execution logic, we will have one-timeness. Therefore, we shift from cryptography (where interaction is inevitable for OTP) to trusted computing technologies. Trusted Execution Environments (TEEs) are hardware-assisted secure modes on modern processors, where execution integrity and secrecy are ensured [182] (see Section 6.2.2 for further explanation). We propose two configurations for one-time programs built on TEEs: (1) deployed directly in the TEE and (2) deployed indirectly via TEE-backed one-time memory and garbled circuits (Frigate [190]) outside of the TEE.

Intuitively, the use of a TEE may appear to provide a trivial solution. TEEs provide desirable qualities for realizing a one-time program, including platform state binding and protection of secrets. However, OTP faces a stronger adversarial model (e.g., Bob's physical possession of the device), whereas TEE usually provides less protection against physical attacks. Therefore our design and implementation require substantial care. For example, in-memory secrets must be taken care of, as a naïve exposure in RAM can cause the system to be vulnerable to the cold-boot attack. An adversary may extract sensitive information through a successful attack with which one-timeness could be broken.

On an application-specific basis, performance needs to be taken into account. As the principle goes, security that is not usable is valueless. Our pragmatic methodology for building a more practical solution to OTP is expected to outperform purely cryptographic OTP and OTM solutions which may still remain prohibitively expensive.

Contributions In this chapter, we consider how to realize one-time programs in a practical way using trusted computing technologies.. Our system, built using Intel TXT and TPM, is available today (as opposed to custom OTP/OTM implementations using FPGA [143], PUF [153], quantum mechanisms [41] or online services [155]) and could be built for less than \$500.[1] We discuss in Section 6.2.2 other TEE possibilities and why we choose TXT instead of SGX or ARM TrustZone; nevertheless, if we were to choose TrustZone, which we do not consider for reasons discussed in Section 6.2.2, the cost will be even lower, given that the custom-built devices with minimal components are cheap, for as low as a few dollars. Our system is designed to defend against the cold-boot attack by carefully exposing a minimal amount of the input in RAM. To illustrate the generality of our solution, we also map the following application into our proposed OTP paradigm: a company selling devices that will perform a private genomic test on the customer's sequenced genome. The device contains certain proprietary algorithms as the company's private input and the customer's genome should also be kept secret from the company. For this use case in one of our two variants (TXT-only), the company can initialize the device in 5.6 seconds and the customer can perform a test in 34 seconds.

6.2 Preliminaries

We first state the design goals for the OTP system, especially requirements for a TEE to realize the general design. We then list any design assumptions and analyze several popular TEEs as candidates for the prototype implementation.

6.2.1 Design goals

A one-time program (OTP) is intended to run on a single input and disallow subsequent runs on any different input, as proposed by Goldwasser et al. [92]. The creation of an OTP is generally dependent on the instantiation of a one-time memory (OTM), although recent work [155] has shown how to achieve OTP using secret sharing without OTMs. OTMs allow one of two keys to be returned without ever revealing the other. More details on what constitutes a one-time program are given in Section 6.2.5, alongside previously proposed techniques for instantiating an OTM. Increasingly, processors are providing hardware-enforced isolation in the form of a trusted execution environment to ensure operation integrity. Intuitively, using a TEE to replace the OTM or more straightforwardly the whole OTP becomes an appealing solution for us.

[1]An example could be Intel STK2mv64CC, which is a Compute Stick that supports both TXT and TPM, priced at \$459.00 USD on Amazon.com (as of January, 2018).

One-time programs can be conceived of as a non-interactive version of a two party computation: $y = f(a, b)$ where a is Alice's private input, b is Bob's, f is a public function or program, and y is the output. Alice hands to Bob an implementation of $f_a(\cdot)$ which Bob can evaluate on any input of his choosing: $y_b = f_a(b)$. Once he executes on b, he cannot compute $f_a(\cdot)$ again on a different input.

Properties We informally consider an OTP to be secure if the following privacy conditions are met:

1. The privacy of Alice's input a with respect to Bob.
2. No more than one b can be executed in $f(a, b)$ per device.
3. The privacy of Bob's input b with respect to Alice.

We argue the security of our systems in Section 6.7 but provide a synopsis here first. We use property 2 in defining property 1, so we will start with property 2.

Enforcing property 2 is the key benefit by using TEEs, which is mostly about execution (logic) integrity. Upon checking a certain protected flag (as opposed to relying on cryptographic techniques), the program logic determines if it should execute or abort. This property of TEEs can either be applied directly to achieve OTP (see Section 6.3) or indirectly via making a one-time memory device (see Section 6.4) as per the Goldwasser et al. construction.

Given property 2, we consider property 1 to be satisfied if an adversary learns at most negligible information about a when choosing b and observing $\langle \mathsf{OTP}, f(a, b), b\rangle$ as opposed to simply $\langle f(a, b), b\rangle$, where OTP is the entire instantiation of the system, including all components of the device and system details. Note that since f is a public function (or program), learning a breaks property 2 at the same time (i.e., with both f and a, the adversary can compose $f_a(\cdot)$).

Property 3 concerns the privacy of Bob's input b. As Bob is provisioned a device that can compute $f_a(b)$ without online interaction with Alice, Bob's privacy is unconditionally secure. This specifically refers to Alice's lack of both physical possession of the device and network connectivity. There is, however, a possibility that the device surreptitiously stores Bob's input and tries to leak it back to Alice. We discuss this systems-level attack in Section 6.7.

Requirements for TEE Based on the aforementioned desired system properties, we now look at what requirements a candidate TEE needs to satisfy. To achieve property 2, isolated execution with integrity (**R1**) usually suffices, which is the fundamental feature of a TEE. However, particular to OTP, such one-timeness implies statefulness and must be determined by a non-volatile flag. Therefore, we also require non-volatile secure storage (formally termed as Secure Element, **R2**) for the TEE. Property 1 involves two aspects: I) secrecy of the stored a, and II) secrecy of the a in execution. For I, we need the capability to bind the secret to the exact machine state (program being executed) and hardware. Sealing (**R3**) achieves this with the assistance of R2. Moreover, II is trickier, as COTS processors have not supported encrypted execution (see XOM [169], and a comprehensive survey [113]), which means secrets in use are exposed in main memory. With that in mind, we consider two types of exposure: if the RAM content can be exposed to other code on the same device, and if the RAM content can be exposed physically to the outside. We require the TEE to have no same-device memory exposure (**R4**), and no physical leakage (**R5**). Even if not all the outlined requirements (**R1** - **R5**) can be satisfied by a particular TEE, we would like to see which is best-positioned to realize OTP and what additional steps can be taken to compensate for the missing.

A potential generalized construction with a TEE follows:

1. An existing program is converted into or a new program is written in plain C (or another designated language). To minimize the Trusted Computing Base (TCB), we do not intend to support a rich language environment.
2. A key is generated in the TEE and, together with an initial value of “0” as the flag, sealed in the Secure Element (SE).
3. Alice’s input, optionally with the program is then sealed, or encrypted (with the key), in the SE or other non-volatile storage, depending on its size.
4. When the system is shipped to Bob, the execution starts in the TEE, reading in Bob’s input either from file or the command line. The program unseals the flag to check if it is already “1”; if so, it aborts. Otherwise, the program sets the flag to “1” and seals it back. Then the key, Alice’s input, and optionally the program are unsealed/decrypted.
5. Taking the inputs of Bob and Alice, the program produces a certain output following execution. Thereafter, Alice’s input can be securely wiped from the SE (even if it is not, future unsealing attempts will be prevented).
6. Bob retains the ability of attesting to the integrity of this execution by either

contacting the IAS server (in the case of Intel SGX, see below) or matching with known good values (for other TEEs).

6.2.2 Trusted execution environments

Trusted computing (where TEEs belong) already has a history of more than a decade (cf. an earlier endeavor of Texas Instrument M-Shield [26] on OMAP). TEEs are usually architecture-shipped, with a primary focus on securing processor execution. They can be categorized as follows: 1) Exclusive. Exemplified by Intel TXT, this type of TEE suspends all other operations on the processor and owns all resources before it exits. The advantage is less attack vectors exposed; or 2) Concurrent. Represented by Intel SGX and ARM TrustZone, this type creates secure enclaves or worlds that exist alongside other processes. There might be multiple instances at the same time. These are more suitable for application-level logic. In the following, we discuss a few typical TEE options in the context of OTP, and see their suitability for matching each of our stated requirements. All TEEs satisfy **R1**, without mentioning.

Intel TXT and AMD SVM. TXT and SVM are simply counterparts on their respective vendor's platform, with nearly the same properties (slight differences). They are exclusive by nature and rely on a security chip called TPM (Trusted Platform Module), corresponding to **R2**. When the secure session is started anytime, TXT/SVM measures the loaded binaries and stores the results in the TPM. Two primitives are important: 1) Measured launch. TXT/SVM can compare the measurements with the "good" values in the TPM and aborts execution if mismatch occurs. 2) Sealing (platform binding, satisfying **R3**). Sealed data can only be accessed in the intact, genuine program and correct platform. Their exclusiveness naturally meets **R4**, as no other code can be run simultaneously. As desktop processors, detachable RAM modules are inevitable, so the cold-boot attack fails **R5**. We will discuss a workaround in Section 6.3.

ARM TrustZone [206]. TrustZone introduces the notion of secure world and normal world. The secure world coexists with the normal world, with everything (including I/O) separated. The two can communicate through a special monitor. This leaves it questionable for **R4**, as there might be potential side-channel attacks from code running in the normal world (cf. [166]). Since it is coupled with the ARM architecture, we can use it on mobile platforms or a dedicated device other than a desktop. This intrinsically satisfies **R5** as it should be difficult (if not impossible) to physically extract RAM secrets, e.g., probing or detaching memory modules. TrustZone also supports sealing satisfying **R3**. An obvious advantage of TrustZone is its secure peripheral communication (enabled by the AMBA3 AXI to APB Bridge). For example, if a small region of the screen is allocated to the secure world, user input there cannot be intercepted by the normal world OS. However, in our OTP, we have no need to involve a regular full-blown OS. Moreover, one of its disadvantages is that the essential secure element (where secrets are stored, like TPM) is not standardized and always vendor-specific [292], thus failing **R2**. This means for any OTP we develop, we have to collaborate closely with the device manufacturer, whereas for TXT, we can buy COTS devices. Nevertheless, if such collaboration became possible for a specific organization, making use of TrustZone on mobile platforms can significantly lower the cost (from approximately $500 to a few dollars) per device.

Intel SGX [16]. More recent than TXT, SGX (Software Guard Extensions) can also be utilized to achieve one-timeness. Intel SGX provides finer-grained isolated environment (measurement-based like TXT) where individual secured apps (called *enclaves*) coexist with the untrusted operating system (thus failing **R4**). The integrity of the program logic (e.g., refusing to run a second time) is guaranteed by the measurement of enclaves before loading. However, what was missing has been secure persistent storage for the flag (to ensure one-timeness) and Alice's input (SGX did not use TPM in the first place); without secure storage, Bob can simply make a copy of both before execution/evaluation, hence defeating one-timeness. To bring back freshness with SGX-sealed data, Intel recently added support for non-volatile on-chip monotonic counters (similar to TPM), see [179]. Therefore, SGX-sealing the flag and key pairs with replay attack resistance is feasible now (**R2** and **R3**). With respect to **R5**, unlike TXT, enclave memory remains encrypted and thus is not susceptible to the cold-boot attack.

There have been also other TEEs around not discussed here, for the reason that either they are less used or obsoleted (e.g., M-Shield [26]) or no sufficient public information is available to support development (e.g., Apple Secure Enclave Coprocessor [19]). We decide to implement our engineering prototype with Intel TXT with the following considerations (compared with SGX). Note that since TrustZone requires vendor collaboration, we skip it for now.

1. **Fewer known flaws.** TXT has been time-tested and known flaws are already stable public information (see Section 6.7). For SGX, there have

been multiple reports regarding various side-channel attacks mounted by malicious/compromised OS or even peer apps [302, 241]. What is worse, Intel admits it as a known flaw that will remain, leaving the closing of side-channels as a responsibility of enclave developers [144].

In other words, side-channels are explicitly out of the threat model of SGX. Such a flaw allows potential multiple or even unlimited number of executions of the protected program, which Bob is motivated to do. On the other hand, although TXT used to have a few system/hardware-level flaws [297, 299] (as no other software can coexist), there are no recent such reports, and previous ones have been patched or not-applicable any more with newer CPU versions. Note that certain attacks based on SMM (System Management Mode) have also been targeting TXT, but does not pose as much threat here as explained in Section 6.7, Item (i).

2. **Meltdown [170]/Spectre [158].** The lately identified flaws in modern processors make side-channel attacks potentially ubiquitous. An *exclusive* trusted environment (where no other OS/entities/processes exist) is more preferable in achieving one-timeness, which is the case for TXT.
3. **Dedicated environment.** SGX is positioned differently than TXT and does not replace it, in the sense that the former allows multiple user-space instances for cloud applications, whose attestation requires contacting Intel's IAS server each time. In contrast, TXT is a dedicated environment, with reduced attack vectors, that also allows local attestation.

6.2.3 Threat model

In the following, we list certain conditions and assumptions for our prototype OTP. As the design can be generalized, more assumptions can be relaxed in a specific OTP application.

- We assume that Alice is semi-honest and Bob is malicious. In addition, Alice can be computationally unbounded. Alice is curious about Bob's input and Bob is always interested in stealing Alice's input and executing the program with his different inputs multiple times.
- Both Alice and Bob have to trust the hardware manufacturer (in our case, Intel and the TPM vendor) for their own purposes: For Alice, the circuit can only be evaluated once on a given input from Bob; for Bob, the received circuit is

genuine and the output results are trustworthy.

- For the majority of the chapter, we assume no parties other than the aforementioned ones are involved/malicious. Specifically, the OTP machine delivered to Bob is not compromised. We add discussion of a third-party adversary in Section 6.7.
- Bob has only bounded computational power (e.g., cracking with a few PCs for days), and may go for some lab efforts, such as tapping pins on the motherboard and cloning a hard drive, but not as complicated as imaging a chip.
- More specifically, components on the motherboard cannot be manipulated easily (e.g., forwarding TPM traffic from a forged chip to a genuine one by desoldering). Even if it is possible, neither of the parties gain anything, as without tampering the TPM chip, changes to the genuine chip are still irreversible and one-timeness is enforced.

6.2.4 Terminology

Below, we define the parties and phases (in the context of their supporting technologies) presented in the remainder of the chapter.

- *Vendor* (Alice): The vendor is both the provider and owner of the one-time program to be evaluated.
- *Client* (Bob): The client will receive the one-time program from Alice, evaluating it on its own input.
- *Sealing*: This term, rather than referring to just the specific TPM or SGX operation, is used to refer to any environment-binding encryption or access-control. Any change to the loaded program causes the environment's measurement to change, thus preventing sealed data from being unsealed.
- *Trusted selection*: This refers to the key selection process that happens inside TEE, whose integrity and confidentiality are protected. This is the part of OTP that enforces one-timeness. It comprises two modes: provisioning mode and execution mode, as sealing must be done in the same measured program (i.e., if Alice uses a different program to seal the data, Bob will not be able to unseal it).

6.2.5 Additional background

We provide additional background helpful for understanding one-time programs, garbled circuits, and one-time memories.

One Time Programs A one-time program (OTP), as introduced by Goldwasser [92], is an implementation of a deterministic function which is provided by Alice to Bob. We describe it here with less generality than it was presented originally (but see Section 6.2.3 for a reconciliation of both approaches). Consider the implementation as containing the function itself (unprotected) and Alice's input to the function (cryptographically protected). Bob can choose a single input and evaluate the function (with Alice's input) on it. With the output, he may be able to infer something about Alice's input (depending on the exact function), but he cannot infer anything about her input beyond this. Since Bob is operating the device autonomously from Alice, his input is unconditionally private from Alice. The core requirement of OTP is that while Bob is able to receive the evaluation on a single input of his choosing, he is unable to obtain an evaluation of any other input. The mechanism to enforce this is the topic of this chapter.

The term 'one-time program' is a slight misnomer. One might equate it with a form of copy protection or digital rights management. It is worth illustrating the difference with a simple example. Consider a DRM scenario: Alice providing Bob with a media player (the function) and a movie (Alice's input). The movie will play if Bob inputs a correct access code. In this case, the stream of the movie is the output of the function and once Bob learns the output, he can replay it as many times as he wants. Therefore this is not a valid application of OTP; instead consider the following: Alice provides Bob with a game of Go (the function) programmed with the latest in artificial intelligence (Alice's input). Bob's moves are his input to function. He can 'replay' the game with the exact same moves (resulting in the exact same game and outcome) as many times as he wants (so it is not strictly 'one-time'), however as soon as he deviates with a different move, the program will not continue playing. In this sense, he can only play 'once'.

OTPs can be realized in a straightforward way with trusted execution. From here, we will describe the alternative approach [92] of realizing one-time programs via a simpler primitive called one-time memory (OTM), and composing OTM with garbled circuits.

Garbled Circuits Garbled circuits (GC) were first proposed by Yao [303] as a technique for achieving secure multiparty computation by at least two parties, a *generator* (Alice) and an *evaluator* (Bob). A program is first converted into its Boolean circuit representation. For each of the i wires in the circuit, Alice chooses encryption keys k_i^0 and k_i^1. Each gate of the circuit takes on the form of a truth table, and entries of the truth table are permuted to further conceal whether any particular entry holds a 0- or 1-value. The keys received on each input wire unlocks a single entry of the truth table, itself a key that is released on the output wire and fed into the next gate. Bob receives the garbled circuit from Alice, together with Alice's garbled inputs. Bob garbles its own inputs through oblivious transfer (OT) with Alice. During evaluation, an output key is iteratively unlocked, or decrypted, from each of the garbled gates until arriving at the final output, which is revealed to all participants.

One Time Memory In summary, one-time programs extend garbled circuits where the oblivious transfer phase is replaced with a special purpose physical device called one-time memory (OTM). The protocol proceeds as in garbled circuits with Bob given the circuit, encoded with Alice's input under encryption. Instead of interacting with Alice to learn the keys that correspond to his input, Alice provisions a device with all keys on it. However when Bob reads a key off the device (say for input bit 0) the corresponding key (for input bit 1) is erased. The end result is the same as oblivious transfer: Bob receives exactly one key for each input bit while not learning the other key, whereas Alice learns which keys Bob selected. The main difference is that the key-selection is non-interactive, meaning Alice can be completely offline.

6.3 System 1: TXT-only

The fundamental feature of most Trusted Execution Environments (e.g., TXT in our case) is isolation, integrity protection, and platform state binding. Therefore, the program logic and data secrecy can be guaranteed in TEE. We propose to achieve one-timeness by running the protected program in TEE only once (relying on logic integrity) and storing its persistent state (including both the payload secret and the one-time indicator) in a way that it is only accessible from within the TEE. During execution, all secrets are also protected because of the isolation. The platform state binding ensures that any modified logic (software) or attempt to run on a different device (hardware) will fail.

We name this design *TXT-only*, since we use Intel TXT as the TEE. To achieve minimal TCB and simplicity, we choose native C programming in TXT (as opposed to running an OS/VM). Therefore, per-application adaptation is required as the application logic might be existent in various programming languages (cf. similar porting effort is needed for the GC-based variant, see Section 6.4).

A one-time indicator (flag) is sealed into the TPM NVRAM to prevent replay attacks (i.e., neither readable nor writable outside the correct environment). The indicator is checked and then flipped upon entry of the OTP. Without network connection, the device shipped to the client can no longer leak any of the client's secrets to the vendor. For this reason, only the vendor's secret input has to be protected. We TPM-seal the vendor input on hard drive for better scalability, and there is no need to address replay attacks as sealed vendor input can only be unsealed in the correct environment where one-timeness is enforced.

Considering cold boot attacks (may reveal in-memory data to an adversary but only once), we expose the unsealed vendor input in very small chunks during execution. For example, if the vendor input has 100 records, we would unseal one record into RAM each iteration for processing the whole user input. This way, the destructive cold boot attack only learns one-hundredth of the vendor's secret, and no more attempts are possible (the indicator is already updated). There are two limitations: 1) Performance might be affected. 2) The protected operation must support iterative processing, or at least produce no intermediate sensitive output that cannot be pre-sealed but leaks information.

6.3.1 TXT-only provisioning at Alice's site

At first, Alice is tasked with setting up the box, which will be delivered to Bob. Alice performs the following: (1) Write the integrity-protected payload/logic in C adapted to the native TXT environment, e.g., static-linking any external libraries and reading input data in small chunks. We name it the TXT program. (2) In provisioning mode of the TXT program, initialize the flag to 0 and seal.[2] The one-timeness flag is stored in an NVRAM index with permissions AUTHREAD|AUTHWRITE and PCR selection (17, 18) for read and write. Instead of depending on a password and regular

[2] A flag is more straightforward to implement than a TPM monotonic counter, as the NVRAM index access can be directly bound to PCRs via TPM_PCR_SELECTION of TPM_NV_DATA_PUBLIC, whereas a counter would involve extra steps (such as attesting to the counterAuth password).

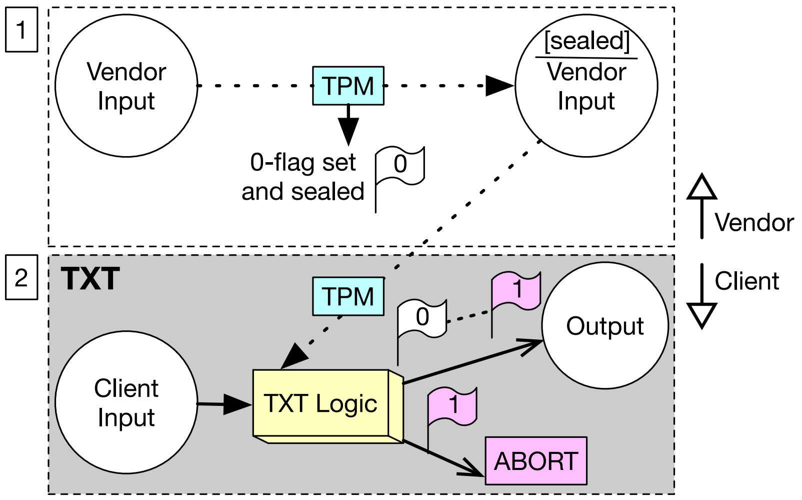


Figure 10: Our realization of One Time Programs spans two phases when relying on TXT alone for the entire computation. Alice is only active during phase 1, while Bob is only active during phase 2.

sealing, we rely on a stronger access-controlled ciphertext. (3) Seal Alice's input onto the hard-drive in the provisioning mode.

6.3.2 TXT-only evaluation at Bob's site

Once Bob receives the computation box from Alice, Bob performs the following: (1) Place the file with Bob's input on the hard drive. (2) Load the TXT program, which will read in Bob's input and unseal Alice's input to compute on. (3) Receive the evaluation result (e.g., from the screen or hard drive).

As long as it is Bob's first attempt to run the TXT program, the computation will be permitted and the result will be returned to Bob. Otherwise, the TXT program will abort upon loading in step (2), as shown in Figure 10.

6.3.3 Trusted execution

We design the program running inside TXT to be loaded by the Intel official project *tboot* [134] and GRUB. It complies with the Multiboot specification [91], and for accessing the TPM, we reuse part of the code from tboot and developed our own functions for commands that are unavailable elsewhere, e.g., reading/writing indices with PCR binding (sealing-equivalent access control). Since we do not load a whole OS into TXT with tboot (to minimize TCB), we cannot use OS services for disk I/O access; instead, we implement raw PATA logic and directly access disk sectors with

DMA.

Before running the provisioning mode of the TXT program, both tboot and our program must be measured and the resulted policy placed in TPM NVRAM indices. We choose to include PCRs 17 and 18 in the policy, corresponding to DRTM (i.e., measuring the CPU-specific Intel ACM module, tboot, and our trusted selection program). Then each time the system boots, Bob/Alice has the option of entering either the provisioning mode or normal execution mode. However, re-provisioning will always erase everything so that security is not undermined. Entering the provisioning mode (multiple times) is different from resetting TPM where all sealed data is automatically invalidated. Once the normal execution mode is entered, the program will refuse to run a second time.

6.3.4 Performance evaluation

We perform our evaluation on a machine with a 3.50 GHz i7-4771 CPU, Infineon TPM 1.2, 8 GB RAM, 320 GB primary hard-disk, additional 1 TB hard-disk [3] functioning as a one-time memory (dedicated to storing client input and sealed vendor input), running Ubuntu 14.04.5 LTS.

We perform experiments to determine the effects of varying either client or vendor input size. Our upcoming case study in Section 6.5 gives the vendor 880 bits and the client 22.4M bits of input, so we use 224 and 880 as the base numbers for our evaluation. We multiply by multiples of 10 to show the effect of order-of-magnitude changes on inputs. We start with 224 for client and 880 for vendor inputs. When varying client input, we fix vendor input at 880 bits. When varying vendor input, we fix client input at 224K bits.

Varying Client Input Table 4 shows the timing results for TXT-only provisioning and execution when keeping vendor input constant and varying only the size of client input. During provisioning, only the vendor input is sealed, so the provisioning time is constant in all cases. As client input size increases, so does execution time, but the change is moderate. Performance is insensitive to client input size up through the 224K case. Even for the largest (22M) test case, we see that increasing the client input size by two orders of magnitude results only in a slowdown by a factor of 3.5x.

[3]The additional hard-disk only needs to be large enough to store client input and sealed vendor input. It needs not be larger than the primary hard-disk.

Client Input (bits)	Prov. (ms)	Exec. (ms)
22M	5640.17	33427.50
2M	5640.17	11078.19
224K	5640.17	9426.56
22K	5640.17	9388.27
2K	5640.17	9393.88
224	5640.17	9394.58

Table 4: TXT-only results with fixed vendor input size (880 bits) and varying client input size, averaged over 10 runs. Though runtime increases with client input size, the change is gradual and suggests that TXT-only OTP is effective at supporting large client inputs.

Vendor Input (bits)	Prov. (ms)	Exec. (ms)
88000	527026.89	921338.53
8800	53515.75	92551.43
880	5640.17	9426.56

Table 5: TXT-only results with fixed client input size (224K bits) and varying vendor input size, averaged over 10 runs. The performance of TXT-only is linear and time taken is proportional to vendor input sizes.

Varying Vendor Input Table 5 shows the timing results when keeping varying vendor input size while keeping client input constant. Although we only tested against three configurations, we immediately see that an order-of-magnitude increase in vendor input size is accompanied by an order-of-magnitude increase in both provisioning and execution times.

6.4 System 2: GC-based

As our TXT-only approach to OTP (System 1) involves sealing Alice's input, and sealing is the most time-consuming operation, it is a good choice when Alice's input is relatively small and Bob's input is substantially larger. However the linear increase in execution time with increases in Alice's input size raises a new question. Is there a construction that complements TXT-only and is less sensitive to the size of Alice's input?

The answer may lie in garbled circuits. During garbled circuit execution, randomly generated strings (or keys) are used to iteratively unlock each gate until arriving at the final output. For more details regarding garbled circuits and their use, refer to Section 6.2.5.

To adapt garbled circuits for OTP, we separate out the key-generation and key-selection steps. As long as we limit key-selection to occur a single time, and the unchosen key of each keypair is never revealed, we can prevent the running of a particular circuit on a different input.

To prevent keys from being selected more than once, we need to instantiate a One Time Memory (OTM), which reveals the key corresponding to each input bit and effectively destroys the unchosen key in the keypair. OTM is left as a theoretical device in the original OTP paper [92]. We realize it using Intel TXT and the TPM. As in our original construction, we seal a one-time flag into the TPM NVRAM and minimize the TXT logic to just handle key-selection, in preparation for GC execution. Alice will seal (in advance) keypairs for garbling Bob's inputs. Bob may then boot into TXT to receive the keys corresponding to his input. When Bob reads a key off the device (say for input bit 0), the corresponding key (for input bit 1) is erased.[4] By instantiating an OTM in this manner, we can replace interactive oblivious transfer (OT) and perform the rest of the garbled circuit execution offline, passing in the keys output from trusted selection. By combining TXT and garbled circuits in this way, sealing complexity is now tied to Bob's inputs, as the key pairs (whose size is twice the size of his inputs) need to be sealed into (for provisioning) and unsealed from (for key selection) the TPM. We name this alternate construction *GC-based* (System 2).

Performance overhead with TPM sealing According to our measurement, each TPM sealing/unsealing operation takes about 500ms and therefore 1 GB of key pairs would need about 1000 hours, which is infeasible. So instead, we generate a random number as an encryption key (MK) at provisioning time and the GC key pairs are encrypted with MK. We only seal MK. This way, MK becomes per-deployment, and reprovisioning the system will not make the sealed key pairs reusable due to the change of MK (i.e., the old MK is replaced by the new key). Note that we could also apply the same approach to TXT-only (i.e., encrypting Alice's input with MK and sealing only MK), as needed by the application. In our chosen scenario, Alice's input is a few orders of magnitude smaller than Bob's key pairs in GC.

[4]Unselected keys remain sealed/encrypted and are still erased from storage for better security.

6.4.1 The Frigate GC compiler

Frigate *Frigate* [190] is a modern Boolean circuit compiler that outperforms several other garbled-circuit compilers (e.g., PCF [160], Kreuter et al. [161], CBMC [118]) by orders of magnitude. *Frigate* is also extensively validated and found to produce correct and functioning circuits where other compilers fail [190]. For these reasons, we decide to use *Frigate* for implementing the garbled circuit components of our GC-based OTP.[5]

Battleship *Battleship* is developed by the same group behind *Frigate* and separates out the interpreter and execution functionalities of *Frigate*. Battleship reads in and interprets the circuit file produced by *Frigate*. *Battleship* is originally designed to be run interactively by at least two parties, a generator and an evaluator. The generator is able to independently garble its own inputs, whereas the evaluator depends on OT to garble its inputs. At each gate of the garbled circuit, a single value is decrypted from the associated truth table containing encrypted entries. Garbled gates are iteratively decrypted until arriving at the final output, which is released to either party. Output need not be the same for both parties.

We make the following modifications to *Battleship* to support one-time programs:

- Split execution in *Battleship* into two standalone phases. In the first phase, a fresh set of random keys (0- and 1-keys for encoding each bit of the client's input) is generated and written out to a file. The keypairs contained in this file will be used during TXT provisioning, after which the file is discarded. The second phase reads in another file containing the subset of keys chosen (during trusted selection) according to the client's input bits and performs evaluation of the circuit. *Battleship* did not originally require these file operations since inputs were garbled and immediately usable without needing to interrupt the system, while we rely on Intel TXT.
- Remove the oblivious-transfer step. In our setting, vendor and client do not perform interactive computation in real time. Instead, the client receives the garbled representation of its input during the trusted selection process inside Intel TXT. The client's input chosen in this way is not exposed to the vendor, who no longer has access to the system after sending it to the client.

[5]Although we choose to go with *Frigate*, it is possible to instantiate our OTP system with other garbled circuit compilers.

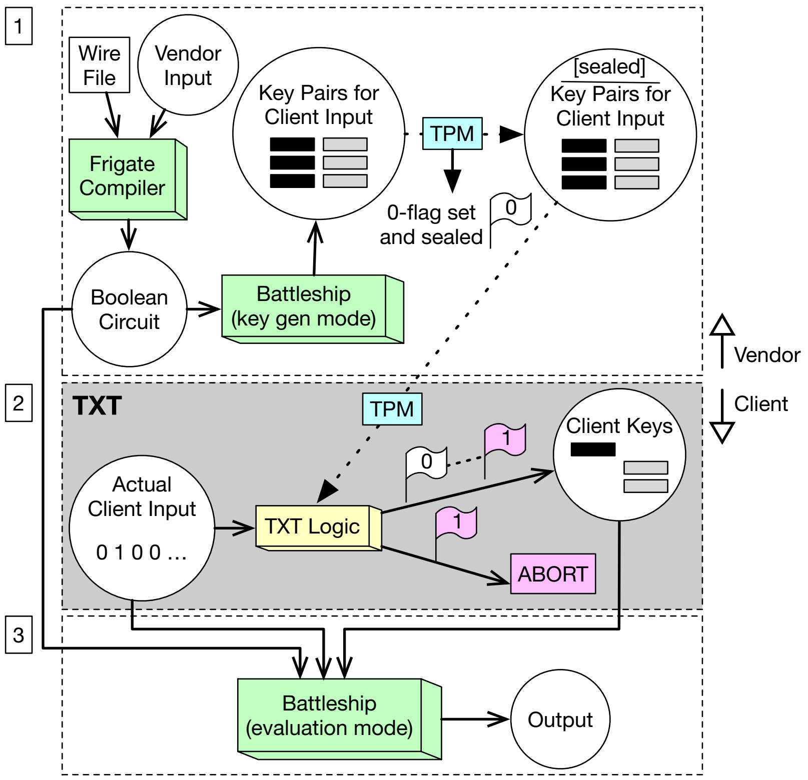


Figure 11: Our GC-based approach to OTP spans three phases. Alice is only active during phase 1, while Bob is only active during phases 2 and 3.

- Remove dependency on the full set of 0- and 1- keys in the second phase. In the original *Battleship* design, generator and evaluator would be separate parties, so the full set of keys were not visible to the evaluator even though it remained available to the generator over the course of evaluation. In our setting, both generator and evaluator functions run on the same machine, so it is imperative that we make the full keyset unavailable. We achieve this by instead supplying both the chosen subset of keys and the raw binary input of the client into the second phase of our modified *Battleship*. In this way, individual keys can be identified as either 0- or 1-keys without needing to examine the full keyset.

6.4.2 Execution steps

Our GC-based approach to OTP relies on TXT for trusted key selection and leaves the computation for garbled circuits, as shown in Figure 11. In our setting, Alice represents the vendor and Bob represents the client.

Provisioning at Alice's site Alice sets up the OTP box by performing the following:

1. Initialize flag to 0 and seal in the TXT program's provisioning mode.
2. Write and compile, using *Frigate*, the wire program (.wir), together with Alice's input, into the circuit.

The wire program may be written and compiled on a separate machine from that which will be shipped to Bob. If Alice chooses to use the same machine, the (no longer needed) raw wire code and *Frigate* executable should be removed from the box before provisioning continues.

3. Load the compiled .mfrig and .ffrig files, vendor's input, and the *Battleship* executable onto the box.
4. Write the TXT program (for key selection) in the same way as in TXT-only.
5. Run *Battleship* in key-generation mode to generate the k_i^0 and k_i^1 key-pairs corresponding to each of the i bits of Bob's input. These are saved to file.
6. Seal the newly generated keypairs onto the hard-drive in provisioning mode of the TXT program.

Alice is able to generate the correct number of key pairs, since garbled circuit programs take inputs of a predetermined size, meaning Alice knows the size of Bob's input. As opposed to our TXT-only construction, costly sealing of all key pairs could be switched out for sealing of the master key (MK) used to encrypt the key pairs.

Evaluation at Bob's site Bob, upon receiving the OTP box from Alice, performs the following:

1. Place the file with Bob's input bits on the hard drive.
2. Load the TXT program in normal (non-provisioning) mode for key selection.
3. Receive selected keys corresponding to Bob's input bits; these are output to disk in plaintext.

As long as it is Bob's first attempt to select keys, the TXT program will return the keys corresponding to Bob's input. Otherwise, the TXT program will abort upon loading in step (2), as shown in Figure 11. After Bob's inputs have been successfully garbled (or converted into keys) and saved on the disk, Bob can continue with the evaluation properly. TXT is no longer required.

4. Reboot the system into the OS (e.g., Ubuntu).
5. Launch *Battleship* in circuit-evaluation mode.
6. Receive the evaluation result from *Battleship*.

When *Battleship* is launched in circuit-evaluation mode, the saved keys corresponding to Bob's input are read in. *Battleship* also takes vendor input (if not compiled into the circuit) before processing the garbled circuit. The Boolean circuit is read in from the .mfrig and .ffrig files produced by *Frigate*. Evaluation is non-interactive and offline. The evaluation result is available only to Bob.

6.4.3 Enhanced security: GC-based Plus

Similarly to the TXT-only OTP, our GC-based approach is vulnerable to the cold-boot attack. Unlike TXT-only, where the protected vendor input is exposed in RAM in chunks, the use of MK in GC-based becomes a single point of failure in face of cold-boot attack. Once the MK inevitably shows up in RAM and is seen by the adversary, all key pairs can be decrypted and one-timeness lost.

To address this, for smaller-sized client input, we can apply the same approach as in TXT-only: seal the key pairs directly and only unseal into RAM in small chunks:

- When the key pairs are in kilo bytes, sealing into TPM NVRAM will ensure no ciphertext access is possible outside the correct environment.
- When the key pairs are larger (only constrained by a tolerable sealing time), we can seal all key pairs and store them on the hard drive. This way the adversary has access to the sealed data (ciphertext) but cannot unseal it. Keep in mind that sealing/unsealing of every 200 bytes takes about one second each.

6.4.4 Performance evaluation

We use the same experimental setup as used in Section 6.3.4 for evaluating System 1. The vendor and client input sizes are also kept the same.

Varying vendor input We are interested in whether our GC-based OTP is less sensitive to the size of Alice's input than TXT-only OTP. Our results are shown in Table 6. Since provisioning (Prov.) involves the sealing of a constant number of keypairs, and selection (Sel.) is dependent on the unsealing of these keypairs to output one key from each, there is no change. Both *Battleship* `gen` and `evl` mode timing is

Vendor Input (bits)	`gen` (ms)	Prov. (ms)	Sel. (ms)	`evl` (ms)
88000	3286.9	4244.03	2508.73	32000.9
8800	3198.7	4244.03	2508.73	32200.4
880	2323.7	4244.03	2508.73	31815.4

Table 6: GC-based results with fixed client input size, varying vendor input size, and encryption of keys by a sealed master key, averaged over 10 runs.

Client Input (bits)	`gen` (ms)	Prov. (ms)	Sel. (ms)	`evl` (ms)
22M	—	346606.87	283704.57	—
2M	16842.8	33934.54	19188.31	305362.8
224K	2323.7	4244.03	2508.73	31815.4
22K	1659.7	991.91	724.24	3643.7
2K	1318.9	906.70	688.62	1631.8
224	1503.7	843.64	600.55	1350.8

Table 7: GC-based results with fixed vendor input size, varying client input size, and encryption of keys by a sealed master key, averaged over 10 runs. Provisioning- and execution-mode times were measured separately. Dashes indicate tests not run due to insufficient memory on our testing setup.

largely invariant, as well. Whereas System 1 performance was linearly dependent on vendor input size, we observe that GC-based OTP (System 2) is indeed less sensitive to vendor input.

Varying client input For completeness, we also examine the effects of varying client input size on runtime. Our results are shown in Table 7. Prov. and sel. stages both increase as client input size increases, since more keypairs must be sealed/unsealed. `gen` and `evl` times are also affected by an increase in client input bits. Most notably, `evl` demonstrates a near order-of-magnitude slowdown from the 224K case to the 2M case. We indeed find that TXT-only OTP is complemented by GC-based OTP, where performance is sensitive to client input size.

6.5 Case Study

In this section, we implement our proposed system on a concrete use case based on genomic testing. We provide additional use cases in Section 6.6.

Background The genetic instructions that determine development, growth and certain functions are carried on Deoxyribonucleic acid (DNA) [256]. DNA is in the form of double helix, which means that DNA consists of two polymer chains that complement each other. These chains consist of four nucleotides: Adenine (A), Guanine (G), Cytosine (C), and Thymine (T). Genetic variations are the reason that approximately 0.5% of an individual's DNA is different from the reference genome. Single nucleotide polymorphism (SNP) is one of the most common form of these variations. SNP defines a position in the genome referring to a nucleotide that varies between individuals. Each person has approximately 4 million SNPs. Each SNP contains two alleles, which correspond to nucleotides. Certain sets of SNPs determine the susceptibility of an individual to specific diseases. If an individual's set of SNPs is analyzed, it may pose a threat to privacy, as this analysis may reveal what kind of diseases a person may have.

Indeed, advancements in genomics research have given rise to concerns about individual privacy and led to a number of related work in this space. Genomic data not only gives information about a person's association with diseases, but also about the individual's relatives [201]. Furthermore, genomic data can uniquely identify a person, hence the need for taking precautions regarding privacy emerges. Different studies in the literature address the privacy of the genomic and health related data. Canim et al. [47], propose a framework that utilizes a tamper-resistant hardware that provides secure storage and processing for clinical genomic data. Naveed et al. [200] introduce a cryptographic tool that is called controlled functional encryption, in which a service provider has to send a fresh key request to an authority whenever he wants to evaluate a function on an individual's encrypted genomic data. Their proposed scheme is used to determine patient similarity, paternity, and kinship. In [291], several private edit distance protocols, which provide high efficiency and precision, are proposed to determine similar patients across different hospitals.

There are several works on privacy-preserving genomic testing that are realized by using cryptographic tools. Baldi et al. worked on efficient techniques for privacy-preserving testing on fully sequenced genomes [28]. In [25], a privacy-preserving system for storing and processing genomic data is proposed. The proposed system is based on homomorphic encryption and privacy-preserving integer comparison. Fisch et al. create a functional encryption based system using Intel SGX [81] . They consider a scenario in which a genetics researcher collects public-key encrypted genomes from individuals and the researcher requests an analysis on these genomes from an authority.

While a number of different techniques have been proposed for privacy-preserving

	Non-interactive	One-timeness	Pattern-hiding
Our technique	✓	✓	✓
Homomorphic Encryption	-	✓	✓
Functional Encryption	-	✓	-

Table 8: Comparison of our technique with existing methods for genomic testing.

genomic testing, ours is the first work to address this using one-time programs grounded in secure hardware. Moreover, whereas the schemes based on homomorphic encryption lack non-interactivity and the schemes based on functional encryption lack non-interactivity and pattern-hiding, our technique provides all three properties, as shown in Table 6.5. We did not specifically implement these techniques and compare our solution with them. However, from the performance results that are reported in the original papers, we can argue that the proposed scheme provides comparable (if not better) efficiency compared to these techniques.

In this work, our aim is to prevent the adversary (the client or Bob) that uses the device for genomic testing, from learning which positions of his genome are checked and how they are checked, specifically for the genomic testing of the breast cancer (BRCA) gene. BRCA1 and BRCA2 are tumor suppressor genes. If certain mutations are observed in these genes, the person will have an increased probability of having breast and/or ovarian cancer [290]. Hence, genomic testing for BRCA1 and BRCA2 mutations is highly indicative of individuals' predisposition to develop breast and/or ovarian cancer throughout their lives.

We aim to protect the privacy of the vendor (company) that provides the genomic testing and prevent the case where the adversary extracts the test, learns how it works, and consequently, tests other people without having to purchase the test. We aim to protect both the locations that are checked on the genome and the magnitude of the risk factor corresponding to that position. Note that the client's input is secure, as Bob is provided the device and he does not have to interact with the vendor (Alice) to perform the genomic test.

6.5.1 Genomic test

In order to perform our genomic testing, we obtained the SNPs related with BRCA1[6] along with their risk factors from SNPedia [49], an open source wiki site that provides

[6]Similarly, we can also list the SNPs for BRCA2 and determine the contribution of the observed SNPs to the total risk factor.

SNP Reference Number	Position	Alleles	Risk Factor
rs41293463	43051071	AT	6
		GG	6
		GT	6
rs28897696	43063903	AA	7
		AC	6
rs55770810	43063931	CT	5
		TT	5
rs1799966	43071077	GG	2
		AG	1.1
rs41293455	43082434	CG	5
		CT	5
		TT	2
rs1799950	43094464	GG	2
		AG	1.5
rs4986850	43093454	AA	2
rs2227945	43092113	GG	2
rs16942	43091983	AG	2
		GG	2
rs1800709	43093010	TT	2
rs4986852	43092412	AA	2
rs28897672	43106487	GG	4
		GT	4

Table 9: SNPs on BRCA1 and their corresponding risk factors for breast cancer.

the list of these SNPs. The magnitude of risk factors ranges from 0 to 10 [253]. A risk factor greater than 3 indicates a significant contribution of that particular allele combination to the overall risk of contracting breast cancer. The SNPs that are observed on BRCA1 and their corresponding risk factors for breast cancer are listed in Table 9.

We obtain genotype files of different people from the openSNP website [100]. The genotype files contain the extracted SNPs from a person's genome. At a high level, for each SNP of the patient that is on a gene linked to BRCA1, we add the corresponding risk factor to the overall risk. For instance, assume the SNP with ID rs1799950 is observed in the patient's genotype file with alleles A and G. From Table 9, we observe that the contribution of this allele combination to the total risk factor is 1.5 and hence add this value to the overall risk of the patient.

The details of our genomic algorithm is shown in Algorithm 1, where RF corresponds to the total risk factor for developing breast cancer. The "risk factors" file contains the associations in Table 9 while the "patient SNPs" file contains a patient's extracted SNPs. If a SNP on BRCA1 is observed in the patient SNPs file, we check the allele combination and add the corresponding risk factor to the total amount. In order to prevent a malicious client from discovering which SNPs are checked, we check every line in the patient SNPs file. If a SNP related to breast cancer is not observed at a certain position, we add zero to the risk factor rather than skipping that SNP. By doing this, we prevent the client from inferring checked SNPs using side-channel.

Algorithm 1 Genetic Algorithm

```
Input: $RiskFactors, Patient\ SNPs$
Output: $RF$
procedure GENETIC ALGORITHM($RiskFactors, Patient\ SNPs$)
    $RF = 0$
    for line in Patient SNPs do
        SNP_ID = SNP ID in line
        ALLELES = two alleles in line
        for line_rf in Risk Factors do
            SNP_ID_rf = SNP ID in line_rf
            ALLELES_rf = two alleles in line_rf
            if SNP_ID = SNP_ID_rf then
                if ALLELES = ALLELES_rf then
                    RF = RF + risk factor in line_rf
                else
                    RF = RF + 0
                end if
            else
                RF = RF + 0
            end if
        end for
    end for
    return $RF$
end procedure
```

Let i denote the reference number of a SNP and s_i^j be the allele combination of SNP i for individual j. Also, S_i and C_i are two vectors keeping all observed allele combinations of SNP i and the corresponding risk factors, respectively. Then, the equation to calculate the total risk factor for individual j can be shown as follows:

$$RF_j = \sum_i f(s_i^j), \tag{1}$$

where

$$f(s_i^j) = \begin{cases} C_i(\ell) & \text{if } s_i^j = S_i(\ell) \text{ for } \ell = 0, 1, \ldots, |S_i| \\ 0 & \text{otherwise} \end{cases}$$

For instance, for the SNP with ID i = rs28897696, $S_i =< AA, AC >$ and $C_i =< 7, 6 >$. If the allele combination of SNP rs28897696 for individual j corresponds to one of the elements in S_i, we add the corresponding value from C_i to the total risk factor.

6.5.2 GC-Based OTP implementation

The garbled circuit program is written as wire (.wir) code accepted by the *Frigate* garbled circuit compiler. The program closely follows Algorithm 1. For each SNP of Bob's (client's) input, the SNP ID is compared to that of each entry of Alice's (vendor's) input. If the SNP IDs match, the allele-pairs are compared. If the allele-pairs also match, the overall risk factor increases by the associated value. If there is a mismatch at any step, a zero-value is added to the overall risk factor in order to not leak side information.

We choose Bob's input from AncestryDNA files available on the openSNP website [100]. We perform preprocessing on these files in order to arrive at a compact representation of the data contained within. Specifics are available in Section 6.5.3. Alice's input is hard-coded into the circuit at compile-time. This is done by initializing an unsigned int of vendor input size; we use *Frigate*'s wire operator to individually assign values to each bit of the unsigned int structure.

Final input representation Following the original design of *Battleship*, inputs are accepted as a single string of hex digits, with each digit represented by 4 bits. Each digit is treated separately, and the input is parsed byte by byte (e.g., 41_{16} is represented as 10000010_2).

We use 7 hex digits (28 unsigned bits) for the SNP reference number and a single hex digit (4 unsigned bits) to represent the allele pair out of 16 possible combinations of A/T/C/G. Alice's input additionally contains 2 hex digits (8 signed bits) of risk factor, allowing us to support individual risk factor values ranging from -128 to 127. We chose to keep risk factor a signed value, since some genetic mutations can result in a lower risk of disease. Although we did not observe any such mutations pertaining to BRCA1, our representation gives extensibility to tests for other diseases.

Output representation The output of the garbled circuit program is 16 signed bits, allowing us to support a cumulative risk factor ranging from -32,768 to 32,767. This can easily be adjusted for other applications, but is accompanied by substantial changes in the resulting circuit size. For example, for the same functionality, an 11 GB circuit that outputs 16 bits grows to 18 GB by doubling the output size to 32 bits.

OTP Type	Mode	Timing (ms)
TXT-only	Prov.	5640.17
	Exec.	33427.50
GC-based	`gen`	—
	Prov.	346606.87
	Sel.	283704.57
	`evl`	—

Table 10: Performance numbers for our TXT-only and GC-based OTP implementation of the BRCA1 genomic test, averaged over 10 runs. Vendor input is 880 bits, while client input is 22,447,296 bits. Dashes indicate modes not run due to insufficient memory.

6.5.3 GC-based case study setup

Client input To arrive at a compact representation of Bob's input, we employ a simple bash shell script to:

- Remove unused chromosome and position fields.
- Remove comment lines at the start of file and the line containing field headings.
- Remove the "rs" prefix from each SNP reference #.
- Remove all spaces between fields and line breaks between entries, making the entire input one line.
- Convert SNP reference numbers into hexadecimal format, and zero-pad the result to length 7 (hex format allows us to reduce 4 keys per entry for a more efficient representation).
- Merge Allele 1 and Allele 2 fields, and assign a 1-digit hex value to each possible allele pair.

The removal of all spaces and line breaks caters to the original *Battleship* design, which expects inputs to be read in as a single line. It is especially important that reference numbers be padded with zeroes (e.g., 0x3DE2 (15842), becomes 0x0003DE2), given that we merge all inputs into a single line, so entries can be parsed using fixed indices. 7 hex digits is sufficient to support all reference numbers, which have at most 8 decimal digits.

Vendor input If a particular SNP has more than one (*allele pair*, *risk factor*) mapping, then each of these is treated as a separate input (with SNP reference number repeated). Although this leads to increased circuit size, specifying Alice's input in this manner is necessary in order to avoid subtle timing disparities which may leak information about the test being performed. The alternative is to make the *if* condition at line 10 of Algorithm 1 iterate over the mappings associated with each SNP. While it would result in less I/O time, fixing the loop bound according to the maximum number of mappings decreases performance if the majority of SNPs have few associated mappings. This would also complicate distinguishing between entries in our compact representation.

6.5.4 TXT-only OTP implementation

In TXT-only, the same comparison (of the SNP ID and allele pairs) logic is ported in pure C. Alice's input is in the form of 7 hex digits of the SNP ID, 1 hex digit for the allele pair and 2 digits for the risk factor. Bob's input is 2 digits shorter without the risk factor.

Where we pay special attention is that, to minimize RAM exposure of Alice's input, we have to perform all operations per one single record of Alice's, and delete it before moving on to the next. We also seal Alice's one record (10 bytes) into one sealed chunk (322 bytes), which consumes more space. In each iteration, we unseal one Alice's record and compare with all Bob's records.

6.5.5 Evaluation

Our case study aligns with the 22M configuration in Sections 6.3 and 6.4. We pull from those results here.

We see in Table 10 the results for both our OTP systems. Even at first glance, we see that TXT-only OTP vastly outperforms GC-based OTP. Alice's input comprises the 22 SNPs associated with BRCA1, as shown in Table 9. Each SNP entry takes up 40 bits, so Alice's input takes up 880 bits. Bob's input comprises the 701478 SNPs drawn from his AncestryDNA file, each of which is represented with 32 bits, adding up to a total size of 22,447,296 bits.

Provisioning is two orders of magnitude slower in GC-based OTP, and trusted selection itself is an order of magnitude slower than the entire execution mode of TXT-only OTP.

Small Vendor + Small Client **TXT-only**	Small Vendor + Large Client **TXT-only**
Large Vendor + Small Client **GC-based**	Large Vendor + Large Client **TXT-only**

Table 11: Depending on the input sizes of vendor and client, one system may be preferred to the other. GC-based OTP is favorable when large vendor input is paired with small client input; TXT-only OTP otherwise.

Choosing one OTP We already saw in Section 6.3 that TXT-only OTP is less sensitive to client input, whereas we saw in Section 6.4 that GC-only OTP is less sensitive to vendor input. We illustrate the four cases in Table 11. In this specific use-case of genomic testing, we are in the upper-right quadrant and thus TXT-only OTP dominates. However, other use cases, like the Database Queries in Section 6.6, are in the lower-left quadrant where GC-based OTP will outperform TXT-only. What do we do if both inputs are "small" or both are "large"? A safe bet is to stick with TXT-only OTP. Even though GC technology continues to improve, garbled circuits will always be less efficient than running the code natively.

6.5.6 Porting effort

To get started with an application based on our OTP (either variant), the very first step is always creating the payload program, or if existent, porting it to a designated programming language. In the case of the GC-based, rewriting in the Frigate-specific limited-C language is necessary. For the TXT-only, the few technical tweaks such as PATA I/O with our added DMA support are seamlessly transparent to the application developer, since they are only exposed like POSIX-like file operations, similar to *fopen*, *fread* and *fwrite*. We argue that regarding the status quo of most existing OTP solutions, this process has to be (quasi-)manual in terms of programming language. Therefore, we hope, as future work, to either have an automated framework for OTP-specific conversions or (in the case of TXT-only) include a lightweight language environment.

6.6 Other Use Cases

Database Queries Another application in a medical setting can be the case where the protocol is between two parties, namely a company that owns a database consisting of patient data and a research center that wants to utilize patient data. The patient data held at the company contains both phenotypical and genotypical properties. The research center wants to perform a test to determine the relationship of a certain mutation (e.g., a SNP) with a given phenotype. There may be three approaches for this scenario:

1. **Private information retrieval [54]:** PIR allows a user to retrieve data from a database without revealing what is retrieved. Moreover, the user also does not learn about the rest of the data in the database (i.e., symmetric PIR [235]). However, it does not let the user compute over the database (such as calculating the relationship of a certain genetic variant with a phenotype among the people in the database).

2. **Database is public, query is private:** The company can keep its database public and the research center can query the database as much as it wants. However, with this approach the privacy of the database is not preserved. Moreover, there is no limit to the queries that the research center does.

   As an alternative to this, database may be kept encrypted and the research center can run its queries on the encrypted database (e.g., homomorphic encryption). The result of the query would then be decrypted by the data owner at the end of the computation [146]. However, this scheme introduces high computational overhead.

3. **Database is not public, query is exposed:** In this approach, the company keeps its database secret and the research center sends the query to the company. This time the query of the research center is revealed to the company and the privacy of the research center is compromised.

In order to address all of these challenges, we propose a system in which a one-time program is used on a device. The company stores its database into the device and the research center purchases the device to run its query on it. This system enables both parties' privacy. The device does not leak any information about the database and also the company does not learn about the query of the research center, as the research center purchases the device and gives the query as an input to it. In order to determine the relationship of a certain mutation to a phenotype, chi-squared test can be used to determine the p-value, that helps the research center to determine whether a mutation has a significant relation to a phenotype.

To demonstrate the workings of our OTP system, we designed and implemented a genomic test for BRCA1 genes. Our OTP construction can also be adapted to other uses for one-time programs; we provide the intuition below. We must also consider the monetary costs associated with adapting programs into OTP boxes according to our design. If non-interactivity is not required, interactive garbled circuit protocols may suffice.

Additional genomic tests Other tests are possible that operate on a sequenced genome. Further, Bob may have multiple inputs to evaluate on a single function. For instance, an individual may input two or more genomes for a paternity test or a disease predisposition test that may also involve other family members. This functionality can be easily added to the proposed scheme by treating multiple sets of test data as single input, although it does not provide privacy between family members (but provides privacy of the set from the vendor).

Temporary transfer of cryptographic ability OTP lends itself naturally to the situation when one party must delegate to another the ability to encrypt/decrypt or sign/verify messages [92]. In this case, individual OTP boxes must be provisioned and given in advance to the designee, with each box only capable of performing a single crypto-operation. The cost could easily add up, but it might be acceptable for time-sensitive or infrequent messages of high importance (such as military communications). If messages are more frequent, then it may be worthwhile to consider a k-time extension ($k > 1$) to OTP. In either case, the designee is never given access to the raw private key. Care must be taken to restrict the usable time of each box, which can be realized by sealing an end date in addition to the one-timeness flag.

One-time proofs As suggested by [92], OTP allows witness-owners to go offline after supplying a proof token to the prover. This proof token can be presented to a verifier only once, after which it is invalidated. We can certainly realize this functionality using our OTP boxes, since proofs produced by our OTP are invalidated by nature of interactive proof systems and may not be reused. Depending on the usage environment, using our OTP box may or may not be cost-effective. While our implemented system may be too costly to serve as subway tickets, the cost may be justified if our box is used as an access-control mechanism to a restricted area.

Digital cash As a one-time program, this was investigated by [155], which used Shamir's secret sharing in place of OTMs. We borrow their three-party scenario to reason about our own OTP system.

1. The bank supplies OTP boxes with set dollar values.
2. To make a payment, the user provides to the OTP box the shop's hash of a newly generated random number.
    - In TXT, the corresponding keys are selected.
    - After reboot, the selected keys are input into the garbled circuit program, which outputs a signature of the dollar-value concatenated with the shop's hash value.
3. The shop verifies the signature.
4. The shop requests cash from the bank using the signature.

Unlike [155], we have a proper OTM in the form of the TPM. A sealed flag value could enforce the one-timeness, preventing the user from giving valid signatures for more than one shop input. However, our scheme requires further modification to prevent double-spending, as it is possible for two independent shop hash-values to be the same, in which case the user can reuse the associated signature. Furthermore, OTP for digital cash would not be feasible if the held dollar value is less than the cost of the OTP box itself, unless the bank customer's goal is untraceability.

6.7 Security analysis

a) Replay attacks. A major threat for OTP is the reuse of data from a previous state (e.g., before execution). This is sometimes even possible without compromising the one-time logic in TXT. For instance, if there is no freshness in the key pairs, as they are large and have to reside on the hard drive, the adversary may be able to make a copy of the sealed/encrypted key pairs and re-deploy the OTP and evaluate on a different input of his choice. There are two possibilities of re-deployment: either the adversary resets TPM and sets his own owner password, or he enters the provisioning mode (as Alice does). For the first possibility, resetting the TPM causes all sealed data to be invalidated (due to changed SRK), which means the loss of the sealed MK/vendor input (or sealed key pairs in GC-based Plus). To counteract re-entering the provisioning mode, where the genuine TXT program is still in control, we simply use a per-deployment nonce appended to all sealed/encrypted items (or XOR'ing with it). For the MK-encryption variant, since MK itself is re-generated each time the provisioning mode is entered, all the encrypted key pairs have freshness and no replay is possible.

b) A third-party adversary. We prioritize the enforcement of one-timeness as the main goal. However, in light of comprehensiveness, we also discuss a threat that harms the genuineness/correctness of the evaluation results: system compromise by a third-party adversary, e.g., during shipping. If the OS is compromised so that the output from the Battleship evaluation is modified, Bob might be mislead (in favor of the adversary), e.g., showing a disease that Bob does not have.

Secure delivery of the OTP box might be infeasible (assuming in-person handover between Alice and Bob is also very unlikely). To ensure Battleship evaluation's code integrity we may refer to ROTI [56] for trusted installation. But in our case, since the TXT phase already guarantees one-timeness, Bob only needs to measure the shipped garbled circuit and resort to his own machine for execution. There is no necessity to ensure Battleship evaluation's process integrity (for which execution in TXT seems a straightforward option). This is because Bob has motivation not to interfere with this process for his own correct result.

c) Memory side-channel attacks. Despite the hardware-aided protection from TXT+TPM, the TXT program must, at certain points, operate on sensitive plaintext data. For instance, MK is needed for encrypting/decrypting key pairs and the key pairs when being selected must also be in plaintext. There are generally two categories of memory attacks: one relies on software/firmware vulnerabilities such as DMA attacks [246]; and one is purely physical exemplified by the cold-boot attack where the RAM modules are mounted to another machine to be accessed after the content is preserved using liquid nitrogen due to the remanence effect.

In TXT, DMA is properly disabled and there is no other software (hence the exclusiveness) running in parallel (no OS/hypervisor). Therefore, generally the first category of memory attacks can be excluded (cf. previous reports [297, 299]). We also assume that Alice has the motivation to select a hardware model with no known public flaws, to ensure one-timeness of her program.

However, the cold-boot attack is effective as long as plaintext content is in RAM. For small-sized secrets like MK, we reduce the time MK is exposed but due to its constant presence in RAM (for decryption/encryption) if an adversary gains access to memory contents his chance of getting MK is still high. As mitigations, existing academic/industrial solutions can be used (e.g., [194, 103, 285, 251]); especially, in our case MK is as small as a few bytes which fits perfectly into the alternative locations (other than RAM) such solutions propose to use, e.g., CPU registers, caches, GPU registers. Oblivious RAM (ORAM) is another potential countermeasure to hide memory access patterns. For larger secrets, like the key pairs/vendor input, we perform block-wise processing so that at any time during the execution, only a very small fraction is exposed. Also, as cold-boot attack is destructive (only one attempt), the adversary will not learn enough to reveal the algorithm or reuse the key pairs.

d) Input credibility/correctness. In genomics scenario, one may not want a third party to run a test on his genome without his consent. Similarly, an attacker should not run several tests using fake genomes to infer the protected function. As future work, we will add a mechanism to verify (i) credibility of the input data (i.e., that data indeed belongs to a real individual), and (ii) ownership of the input data (that data indeed belongs to the individual that is running the test, or the test is being conducted with his consent). To achieve this, we can utilize digital signatures and biometric attributes together. The input data to the device, which is the sequenced DNA, may be signed together with a biometric factor like fingerprint, by an authority (sequencing facility). Along with his input, Bob should also provide a fresh biometric input to the device so that it can be checked against the signed biometric attribute. While the signature over the input provides credibility, the biometric factor ensures ownership.

e) Attack cost. Similar to the above item, Bob may try to infer the protected function and vendor inputs by trying different inputs in multiple instances. Of course, this attack may incur a high cost as Bob will need to order the OTP from Alice several times (if he can afford and is willing to pay). This is a limitation of any offline OTP solution, which can only guarantee one query per box.

f) Inference attacks. Even though Bob is only allowed to run a single test on Alice's function, he may (probabilistically) infer outputs of other correlated tests by observing the output of the test on his selected input. For instance, there may be two genetic disorders that are highly correlated with each other. For instance, the SNP with ID rs429358 has influence on the risk of having both Alzheimer's disease and heart disease [254]. Moreover, different psychiatric disorders are also correlated [64]. This relation is determined according to the SNPs observed. The SNPs associated with schizophrenia and bipolar disorder are also highly correlated. Then, it means that the output of the first genomic test will enable a person to make inferences about the result of the second test. Bob might also infer details of Alice's input to the function depending on how the circuit is designed. Care should be taken to design the circuit with the same circuit depth, independent of Alice and Bob's input values.

g) Adaptive attacks. For one-time programs, the privacy guarantee of garbled circuits alone is too weak against anything beyond honest-but-curious adversaries, as highlighted in [92]. Unless a (projective) garbling scheme is carefully transformed into one that provides adaptive privacy, the resulting scheme used to realize an OTP opens up room for adaptive attacks [31].[7] In a coarse-grained adaptive attack, the adversary selects inputs after inspecting the circuit. Further, the adversary may choose some key pairs, decrypt part of the circuit, and use intermediate information to decide what keys to choose next in a fine-grained adaptive attack. There are several mechanisms for stopping these attacks:[8]

- Allow the adversary to decrypt the circuit but not learn the output of the circuit until all keys have been chosen [92]. The output is blinded with a random value, distributed into n shares, where n is the number of keys. Each time a key is chosen, it is returned together with a corresponding share of the random value. Only after choosing all keys is the random value revealed for unmasking the output.
- Encrypt the circuit using either a one-time-pad or random-oracle-based encryption and reveal the decryption key together with the garbled input in the online phase [31]. A somewhere-equivocal encryption scheme, where a small subset of message bits are equivocal, may also be used [112].
- Place a "holdoff" gate into each output wire that cannot be evaluated until all keys are learned [143].

For our approach, we draw inspiration from [31] and choose to seal the entire circuit

[7]For certain classes of circuits, [142] claims that garbled circuits are adaptively secure without further modification, with security loss tied to pebble complexity of the circuit.

[8]Note that this is not an issue in the fully-online, interactive garbled circuit setting where the circuit is sent over to the evaluator only after the evaluator's inputs have been garbled using OT.

and only unseal it after all keys have been selected, enforcing this using TXT. Alternatively, we may seal a master key used to encrypt the circuit, similarly to our ENC variant approach for protecting the keypairs. The security reduces to the security of TXT, which we already assume. Additionally, if we assume AES is an ideal cipher, then looking at an AES encryption of the circuit is the same as not being given the circuit (in any form) at all for a computationally bounded adversary.

h) Cryptographic attacks. The security of one-time programs (and garbled circuits) is proven in the original paper [92] (updated after caveat [31]), so we do not repeat the proofs here.

i) Clonability. Silicon attacks can reveal secrets (including the Endorsement-Key), but chip imaging/decapping requires high-tech equipment. Thus, cloning a TPM or extracting an original TPM's identity/data to populate a virtual TPM (vTPM) is considered unfeasible. Sealing achieves platform-state-binding without attestation, so non-genuine environments (including vTPM) will fail to unseal.

j) SMM attacks. The System Management Mode (SMM) is a special execution mode in modern x86 CPUs and considered having (informally) the Ring minus 2 privilege, preempting virtually any other modes. Therefore, although not recently, it was exploited [297] to interfere with TXT execution. This attack assumes the compromise of the SMI (SMM Interrupt) handler (which is difficult, but feasible in an ad-hoc manner), and during TXT execution an SMI is triggered and the compromised handler comes in to manipulate anything of the adversary's choice. However, in the case of our OTP, we do not load any standard code that needs SMI and has it enabled (like an OS or hypervisor); instead, our custom program for key selection or OTP execution has SMI disabled from the first line of code (not to mention containing any SMI trigger, e.g., writing to port 0xb2), and thus is not affected by such attacks. Since TXT is exclusive, no other code can run in parallel.

Note that Alice no longer benefits from any attacks (e.g., stealing Bob's input) due to loss of physical possession and network connectivity. We exclude, for now, any potential threats from Intel ME (Management Engine) which is referred to as Ring minus 3 and has a dedicated processor, in the consideration that all rely on ad-hoc vulnerabilities and this topic is still under open discussion [75]. We will follow up on this.

k) TPM relay attack [80]. A man-in-the-middle (MitM) attack specifically targeting TPM-like devices impersonates and forwards requests to a (remote) legitimate device, pretending its proximity or co-location on the same machine, to either learn the secrets or forge authentication/attestation results. In the case of our OTP, only Bob has physical possession and is motivated for such attacks. However, since he cannot clone the TPM chip, whatever real traffic directed to the legitimate one will cause irreversible effect (e.g., flipping the flag) Note that his intension is not merely mimicking, which does not help. Also, we do not send TPM commands in plaintext, except for ordinals and certain metadata. Our ultimate argument is that, regardless of the lab effort we already exclude in Section 6.2.3, the integration of TPM in other microchips (e.g., SuperIO) or an equivalent method will avoid exposing TPM pins for potential probing.

6.8 Related Work

OTP implementations In the original OTP paper [92], OTM is left as a theoretical device. In the ensuing years, there have been some design suggestions based on quantum mechanisms [41], physically unclonable functions, and FPGA circuits [143]. The latter is the closest to a practical design so we expand on it. The authors provide an FPGA-based implementation for GC/OTP, with a GC evaluation of AES, as an example of a complex OTP application. They conclude that although GC/OTP can be realized, their solution should be used only for "truly security-critical applications" due to high deployment and operational costs. They also provide a cryptographic mechanism for protecting against a certain adaptive attack with one-time programs (see Section 6.2.3); it is tailored for situations where the output size is larger than a typical security parameter. Kitamura et al. [155] realize OTP without OTM by proposing a distributed protocol, based on secret sharing, between non-colluding entities to realize the 'select one key; delete the other key' functionality. This introduce further interaction and entities. Our approach is in the opposite direction: removing all interaction (other than transfer of the device) from the protocol. Prior to OTP being proposed, Gunupudi and Tate [105] proposed count-limited private key usage for realizing non-interactive oblivious transfer using a TPM. Their solution requires changes in the TPM design. By contrast, we utilize unmodified TPM 1.2.

6.9 Concluding Remarks

Until now, one-time programs have been theoretical or required highly customized/expensive hardware. We shift away from crypto-intensive approaches to the emerging but time-tested trusted computing technologies, for a practical and affordable realization of OTPs. With our proposed techniques, which we will release publicly, anyone can build a one-time program today with off-the-shelf devices that will execute quickly at a moderate cost. The cost of our proposed hardware-based solution for a single genomic test can be further diluted by extension to support multiple tests and multiple clients on a single device (which our current construction already does). The general methodology we provide can be adapted to other trusted execution environments to satisfy various application scenarios and optimize the performance/suitability for existing applications.

Chapter 7

Explicit Authentication Response Considered Harmful

The aforementioned research topics mainly concern no online third parties, such as a service provider. In this chapter, based on similar defense philosophy (P2), we address guessing/dictionary attacks against passwords where an online trusted party is involved.

7.1 Introduction

Automated online password guessing is a long-standing problem for password-based authentication. Nowadays, this problem is possibly getting worse for reasons including the following. (a) The growth of underground market for stolen credentials; i.e., attackers can turn stolen passwords into tangible profits; see e.g., Holz et al. [117]. (b) The value of user accounts increases over time, e.g., long-standing Facebook profiles, Gmail accounts, highly-reputed Paypal accounts. In many cases, user accounts are not as readily replaceable as in the past—i.e., create a new account if the old one is compromised. (c) User chosen passwords are not getting better in terms of complexity. New services requiring passwords are emerging, causing password fatigue or sharing across sites. Also, the increasing number of online participants (e.g., see [125]) makes the use of common passwords more possible. (d) Attackers are getting more organized than before, and have access to better tools and crackers; for example, they now maintain more robust botnets, and can use better techniques than just brute-forcing, e.g., optimized dictionary attacks [198].

Common countermeasures include: rate-limiting the number of allowed login attempts in a given period of time; the use of captchas to restrict automated attacks, see e.g., Pinkas and Sander [219]; and triggering a two-step authentication, e.g., one-time PIN sent to a pre-registered mobile phone, and personal challenge questions. In most cases, attackers can bypass the countermeasures, at least to a limited extent. For example, assuming a three-strike account locking technique is used, an attacker can still employ a large botnet (e.g., million-node) to test the most common passwords and possibly compromise some accounts; here, the attacker is successful if her goal is to access a few accounts (e.g., to use as intermediate money-transfer accounts), instead of compromising a targeted account. Captchas are mostly detested by human users as they are becoming increasingly difficult to decipher (see e.g., [43]); as a side-effect, login times also increase as legitimate users sometimes need to try more than one captcha for an exact match. Several real-world captcha schemes have been defeated by improved image recognition algorithms (see e.g., [44]). As a result, service providers often leave with no option but to deploy more complex captchas. These limitations are known and several proposals in the past attempted to address the security-usability trade-off in captcha schemes (e.g., [219, 13]).

The fundamental problem here, as we see is that the attacker can learn the outcome of her guess with 100% certainty, using fully automated attacks or involving some trivial human help. Human-assisted captcha breaking services are available, for cheap (see e.g., [191]). As we are aware, verifiers in all known authentication schemes output a success or failure message after a trial, and we argue that such explicit messages aid online guessing attacks. *Explicit messages* may include return codes from an authentication API, protocol data from the verifier, text string, or even the continuation/discontinuation of the attempted session.

We introduce here *Uvauth* (user-verifiable authentication) to reduce the attacker's confidence on the outcome of her guessed password by granting her access for any password she enters. For a given userid, the correct password will lead to the real user account, and all other passwords will provide *fake sessions* (i.e., with fake user data). To avoid detection by re-logging into the same account, same userid-password pair will always result in the same session. Likewise, different userid-password pairs should also lead to different sessions in order that the legitimate session cannot be distinguished from fake ones. The underlying assumption is that real users will implicitly understand the outcome of their authentication attempt by the presented data; i.e., an unfamiliar account will indicate that the entered password is incorrect, and they need to try again. On the other hand, a random attacker may have little or no idea what to expect as user data after being logged in, even if she launches a human-assisted attack. Attackers can perform different operations to discover a fake session, and our goal is to raise the bar for such attacks to succeed—e.g., by requiring non-trivial efforts from the attacker beyond simply solving a captcha. By increasing the attack cost, we choose to tolerate the attacks, instead of addressing them head-on. Users are also freed from "solving" captchas, or going through other reverse Turing tests as part of their authentication.

Note that Uvauth is different than implicit authentication (see e.g., [249]), where a user is authenticated by her usual traits/actions. An explicit outcome is provided at the end of such an authentication attempt, which we would like to avoid. Our proposal is also independent of whatever secrets, features or tokens are used to verify a user; it is the outcome of an authentication attempt that we would like to protect, where online guessing is a concern.

Uvauth's fake sessions can be seen as a form of deception, which has been in use for centuries in traditional wars and conflicts; see e.g., "All warfare is based on deception" [90]. Deception as a cognitive defensive technology has been extensively studied by many researchers for years; see e.g., [233, 310, 59, 27]. In current computer security techniques, this methodology is well demonstrated in honeypots, where deception is used to influence the behavior of attackers, or to collect data for future use, e.g., to understand the attackers and their target systems and network resources (see e.g., [259, 53, 221, 202]).

Our use of deception is not to gain more insights into the attackers' behaviors, but simply to raise the difficulty of online guessing attacks against weak authentication secrets. The following analogy may further clarify the difference. Consider a virtual building with several locked rooms. Honeypots protect access to a room by generating a fake room on-the-fly or claiming that the room is unavailable. In contrast, we create a fake room to protect the lock of a room, assuming the lock is weak—i.e., given enough time, a lock-picker can easily open it. Our use of fake sessions can also be viewed as the no-information leakage property of a perfect one-time pad (OTP) encryption: attackers have no way of verifying a guessed key for an OTP scheme, as a valid key exists for every candidate plaintext, i.e., attackers do not know when they succeed.

Several challenges must be addressed for Uvauth. Generating fake sessions would require additional resources from the verifier, and non-trivial efforts to mimic a legitimate session. Protected accounts should have enough personal content so that legitimate users will easily learn whether they have logged in with the correct password. To address less-/non-personal accounts, we propose the use of distorted images / modified captchas as a communication channel for the authentication result from the verifier to a client. The crucial difference with existing captcha here is that: we do not require users to solve captchas verbatim (i.e., character-by-character) and type the result. Instead, users are expected to use the captcha messages as a second channel to verify their login (i.e., in addition to the content they can see). More challenging captcha schemes can be used in our setting, as users are not required to decipher each character in the exact form.

In summary, our contributions include:

1. In user-level authentication, we introduce the idea of programmatically leaving the result of authentication on the server (verifier). Such hiding of authentication results may enable effective protection against online guessing attacks.

2. We propose the use of adapted distorted image as a computer-cipher/human-decipher channel to communicate short messages in human-machine interaction.

3. Our proposal requires no changes on the client side software or existing password input UI, and can be used with any authentication scheme vulnerable to online guessing attacks.

7.2 Threat Model and Assumptions

In this section, we describe our goals, the conditions under which Uvauth works, and list situations that are considered out-of-scope.

Goals. The objective of our proposal is to make both machine-only and human-assisted attacks significantly more difficult than using the current state-of-the-art captchas. The level of difficulty can be set by the depth of deception in Uvauth's fake sessions.

Assumptions.

a) User-level authentication. We address authentication scenarios where a human user is the claimant and a computer is the verifier. We do not include machine-to-machine authentication, e.g., automated script for connecting to a database server.

b) Weak-secret-based, single-factor authentication. Uvauth can be used independent of any existing authentication technique, e.g., text or graphical password schemes, certificate-based schemes. However, Uvauth's protection is intended for situations where weak-secrets are used that can be efficiently guessed through online attacks (e.g., a human-chosen password vs. a random 128-bit key). Multi-factor schemes that use an additional token or biometrics also may not need protection against guessing attacks, assuming the additional factors provide enough entropy. However, single-factor multi-stage schemes (e.g., SiteKey or personal questions with passwords) may benefit from Uvauth; e.g., the fake session can start right at the end of first-stage of authentication. However, most of our discussion here considers only commonly-used single-stage password authentication.

c) Data-oriented sessions. We focus on accounts that mostly deal with user data (e.g., banking, email), instead of providing some generic services to the user (e.g., Internet access). Implementing fake sessions for service-oriented accounts could be quite challenging, if not impossible. For instance, if simply non-working Internet access is provided, the adversary can easily detect it; or otherwise he can remain using the Internet (if working) regardless of the authentication result.

d) Separate machines. The user/attacker software has no means of accessing the verifier's running environment other than via the network channel used for authentication. Otherwise, authentication results may leak from the verifier through side-channels (e.g., [42, 230]).

e) Random attacker. Attackers in our model are assumed to be random individuals, i.e., unrelated to a target user. If the user is known to the attacker, fake sessions in Uvauth may be detected by known information (e.g., Facebook profile information, email contacts). However, the attacker may know all valid userids of a target service.

f) No offline attacks. We assume that data at rest is safe, e.g., password databases are inaccessible to attackers. Otherwise, simpler offline attacks can be mounted to reveal the passwords (if hashed or encrypted under a weak key).

g) Other password-unrelated security issues. Our proposal only addresses online password guessing; so, if a website or application is vulnerable to other types of attacks such as SQL injections, Uvauth's protection may not help. We also do not address several other threats, including: phishing, malicious software on the client or verifier, and session hijacking attacks.

7.3 Uvauth: User-verifiable Authentication

In this section, we discuss Uvauth and the underlying self-evidence of authentication that may make the scheme feasible. By analyzing some account properties, we also provide a list of considerations for designing fake sessions, and discuss scenarios where Uvauth may be more applicable.

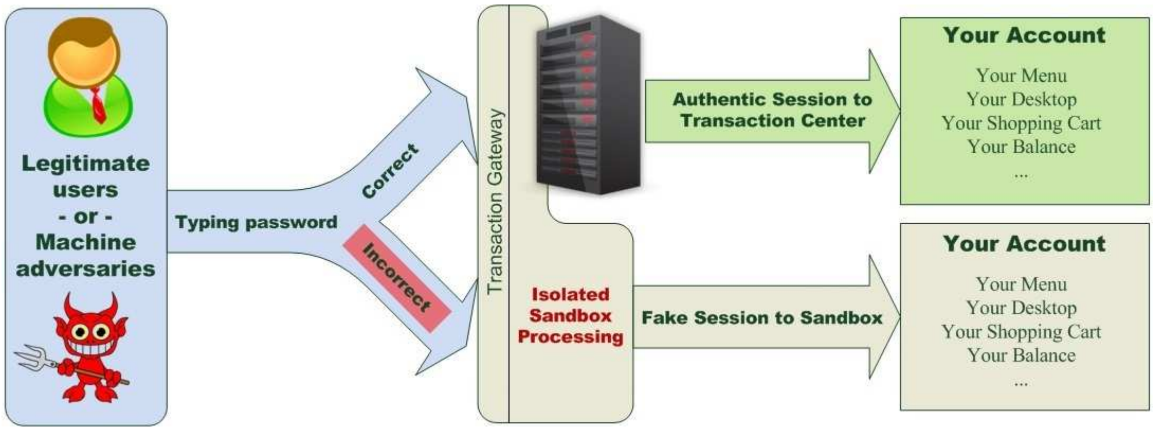


Figure 12: Overview of user-verifiable authentication

Overview. Figure 12 shows an overall architecture of Uvauth. Legitimate users and potential attackers are treated equally, in terms of authentication results. A transaction gateway accepts all incoming authentication requests; the gateway is also configured to authenticate users (e.g., it has access to user credentials). When a correct userid-password pair is received, processing is handed over to the transaction center and a legitimate session is established. Otherwise, when the given password is incorrect, the user/attacker is redirected to a sandbox-enabled environment that hosts fake user sessions. The established sessions in both cases appear to be (almost) the same to a machine. A random human attacker may also be unable to judge the content of the fake account without performing some non-trivial tasks.

7.3.1 Implicit detection of an authentication outcome

We first consider authentication sessions where users can distinguish success/failure without explicit messages from the verifier. This is the basic type of authentication considered in Uvauth, and requires user-knowledge of the target account. In Section 7.4, we discuss less-personal accounts where some explicit hints from the verifier are needed.

Self-verification

If the data fed to end users after a login request is personal and of relatively high-entropy, the presented data itself may be enough for a straightforward and effortless decision by the real data owner. In this case, the authentication result is implicit, i.e., requires no indication of failure or success. Consider the following as examples of this type of authentication. For most active users of a social networking site (e.g., Facebook), users can (possibly) easily identify their own accounts after a successful login—e.g., from the profile info, page layout, friends list and messages. The same is possibly also true for online banking login, identified by e.g., user info, account balance, transaction history and registered bills. These types of accounts are highly personal and quite unique to a user. More importantly, these accounts can be populated with fake information to make them indistinguishable even to *non-owner* human users (in addition to automated bots).

User-verifiability obviously requires that the same user experience is provided for a specific credential used. Therefore, to implement a user-verifiable authentication scheme that is both user-acceptable and attacker-indistinguishable, we must consider the following issues. First, each fake session generated for a specific userid-password pair (even if the userid is non-existent), must appear to be the *same* for a certain period of time. If randomness of fake sessions is distinguishable for login attempts with the same userid with different passwords, attackers can easily detect the difference, and then learn the authentication outcome. On the other hand, the fake session for a specific userid must change with time, as is the case for many user accounts (e.g., new messages and friends in a Facebook account; updated balance and new transactions in a banking account).

Additional login help for legitimate users

To aid users and help identify a successful login, a combination of the following methods can also be used.

a) Customized messages. A user customized welcome message may be used for the identification of a valid session. During account registration, a user can set up some personalized information so that when a correct password is entered, it will be displayed; otherwise, a random message is displayed. Such customized messages may be an image, or excerpts from a book. Note that, our use of customized message/image is different than existing anti-phishing solutions such as SiteKey [29], and Verified-by-Visa personal message [289]. We do not address phishing, and security of Uvauth is not dependent on users' noticing the messages correctly or all the time. If the user does not pay heed to the displayed image/message, they may be mislead into believing a successful login, which eventually will be detected when they check carefully their account information. In contrast to known vulnerabilities in SiteKey (e.g., [305]), no authentication secrets are leaked for the user's mistake in Uvauth.

b) Secondary channels. An out-of-band signalling, e.g., SMS/twitter/email messages can also be used to notify when a login is successful. Mobile SMS is widely used for user status indication in many businesses, such as successful credit card transactions (see e.g., MasterCard inControl [178]). We assume here that the secondary channels are not compromised; otherwise, an attacker can use such a channel for verification. Periodically, users may also be notified about failed login attempts through secondary channels (e.g., in the form of a daily digest).

c) Warning messages. A warning message may be displayed so that the user is reminded that Uvauth is in place, and verify whether they can access their data. An example message is as follows: "Please check your account data; in case you do not see your expected data, try again with the correct password."

d) Dynamic security skins. Anti-phishing techniques such as synchronized random dynamic boundary [304] and dynamic security skins [68] can be used as a means to identify an authentic server, and to communicate success/failure messages to a client browser. Note that, Uvauth's security does not require these visual cues to be 100% reliable, or always correctly matched by users; they simply provide an additional channel for session verification.

e) Limiting fake sessions for *known* devices. Authentication attempts from known devices with prior successful logins for a specific userid can be exempted from fake sessions when an incorrect password is entered, and given directly a traditional failure message (e.g., incorrect userid or password). User devices may be whitelisted by IP addresses, cookies, geolocation services as enabled in popular browsers including Google Chrome[1] and Mozilla Firefox,[2] or through other web-based device fingerprinting mechanisms (see e.g., [208]). Assuming that most legitimate users access their accounts from a relatively fixed set of devices (computers at home or office, or mobile devices), such exemptions from fake sessions may aid usability; similar mechanisms have been explored in prior work (see e.g., [219, 13]; more in Section 7.6). However, to counter guessing attacks from infected whitelisted devices and cookie theft, such exemptions must be limited (e.g., by the number of allowed attempts without fake sessions).

7.3.2 Designing fake sessions

Uvauth's effectiveness depends on attackers being unable to detect fake sessions *efficiently*. Below, we discuss few considerations and account properties for designing effective fake sessions.

Account properties

Here we list four factors that may be used to categorize account types. We also discuss how these factors may be considered during the generation of fake sessions.

a) Server-side data retention. Here we consider whether the user is allowed to make changes after logged in and to what extent the changes are kept and accessible when she logs back in at a later time. This feature of a user account could be resource-intensive, as fake sessions may also need to store attacker-initiated changes. If no changes are stored, inconsistent fake sessions may still be useful to some extent; cf. Neagoe and Bishop [202]. For read-only accounts (e.g., call logs of a pre-paid phone card), generating fake sessions could be much easier. However, most online accounts generally allow at least some changes (e.g., profile parameters). If the size of updateable data is small, the cost of consistent fake session generation may still remain affordable.

b) Client-side data representation. For most account types, users get access to some data after logged in. How much an attacker can understand the meaning of

[1] `https://support.google.com/chrome/answer/142065`

[2] `http://www.mozilla.org/en-US/firefox/geolocation/`

user data, determines how easily she can detect a fake session. For highly-personalized data (e.g., photos, blogs, and calendars), fake session detection would be significantly difficult for an attacker, even if human assistance is used; the attacker has no obvious means to distinguish between fake and real data. For impersonal, human-readable data (e.g., magazine subscriptions), fake sessions should be populated with context-aware, meaningful data. For impersonal data with machine semantics (e.g., protocol traffic or command responses), it may be more difficult to generate fake sessions, and sometimes specific restrictions should be applied to limit the cost of fake sessions (e.g., running processor-intensive jobs in a fake ssh session).

c) Update types. Some accounts are update-driven, i.e., frequently updated directly by both the account owner and others for the purpose of communication; examples include email and social networking accounts. Some accounts are activity-driven, i.e., indirectly updated by user transactions; examples include credit card accounts. Some accounts may be of mixed type; e.g., a seller's Paypal account is updated by Paypal (e.g., transaction logs) and other users (e.g., comments). These different account types should be modeled correctly to design realistic fake sessions.

d) Externally-modifiable data. If anyone can influence the content of a target account, the account is considered externally-modifiable; examples include email accounts (e.g., anyone can send an email), social networking accounts (e.g., public posts). These accounts are susceptible to the post-and-check attack as discussed in Section 7.5.

Considerations for fake session generation

a) Verisimilitude. There is a trade-off between the deployment of more realistic/consistent fake sessions with more functionality and resource consumption on the server. We define the depth of verisimilitude as the levels of operation a fake session would allow, before it may be detected by an automated attacker. Also, not all functions are equal in terms of costs–e.g., allowing the update of a profile parameter vs. searching for a transaction. As an example, consider a fake session at an online banking portal; an automated attempt can be performed by an attacker to transfer a certain amount of money to an account that directly/indirectly belongs to him, as such tasks are not far down in the operations hierarchy. A countermeasure is to output deceptive statements such as "Transfer-out is not activated for this account", "USB token is required for this transaction". See e.g., Rowe [233] for an in-depth discussion on how to design good deceptions for intruders with a probabilistic model of belief and suspicion. Moreover, text strings (e.g., names and messages) used in fake sessions should meet certain criteria; existing work on generating (somewhat meaningful) random words/phrases may be used (see e.g., [61, 24]). Note that, for Uvauth to be effective, detection of fake sessions must be non-trivial, but it is non-essential to deploy highly complex fake sessions to make detection very difficult.

b) Timing characteristics. Sometimes due to network delay or processing on the server side, logging in or operations on a website are subject to different levels of responsiveness. Fake sessions should insert lags when required to simulate timing characteristics of different operations in the operations hierarchy, hours of the day, or even seasons in a year. This may also help confuse intruders as they cannot detect fake sessions by profiling timing characteristics. The freed time slots can be used for scheduling more fake sessions.

c) Data sanitization. Data sanitization (also known as redaction for printed documents) is to hide or transform confidential information before publishing. Examples include erasing customer names, randomizing figures, or disrupting the order of user behaviors. In some scenarios, it may be necessary to reuse parts of the real production/user data for generating fake sessions, especially accounts with a lot of user data. Up-to-date operating data from a real system may be sanitized by removing all privacy/security-sensitive parts, while retaining interrelated rationality (see e.g., [34, 211]). For instance, in the case of a web portal of a mobile phone subscriber, the prefix of a login phone number may indicate some regional information; therefore, the presented information, such as, the numbers in the call log and the address of residence must also appear legitimate after sanitization. The account balance can be randomized to some extent, but the call/message logs could be pulled, sanitized and mixed from a group of real users (i.e., individual identifiers are removed but group characteristics are preserved). However, special care must be taken to sanitize data to avoid exposure of sensitive data (see e.g., [199, 33]). For Uvauth, a significant amount of fake data can be mixed with user data before applying sanitization, which may reduce the risk of privacy exposure.

d) Virtualization. As Uvauth may need to deal with a large number of fake sessions (e.g., when under guessing attack from a botnet), virtualization technologies can be used for creating and hosting those sessions efficiently. We have not tested generating such large-scale VM deployment for evaluating Uvauth; cf. CLAMP [214]. Virtualization may also help limit resources allocated to fake sessions, especially when under heavy-load (e.g., due to DoS attack).

7.4 Distorted Image as a Communication Channel

In this section, we discuss the possibility of using captchas as a one-way communication channel (server-to-user), and propose few variations of existing captchas for this purpose. These captcha variants may be considered when techniques in Section 7.3.1 are not preferred (e.g., for deployability or usability reasons). Less personalized accounts (e.g., movie streaming websites), and managed-systems in batch (e.g., remote administration), may benefit from the proposed methods. We assume that these accounts would be attacked primarily by bots (i.e., no human assistance), as they may be less valuable compared to personal/financial accounts.

7.4.1 Captchas as a cipher

Most current captcha techniques are based on the use of distorted images (or similar methods), and are used before authentication, to verify the presence of a human user. In contrast, we propose a post-authentication use of captchas. The idea is to utilize the generation and recognition of distorted images to communicate the authentication result back to end users. End users will not be *tested* with our schemes below, and no user response is needed; users simply become recipients of the *ciphered* information. Note that, similar use of captchas has been proposed earlier for different purposes, e.g., verification of message integrity in an untrusted terminal [152], and NSA-proof fonts [197].

Using captchas to communicate messages is relatively immune to relay attacks (as compared with regular captchas). A machine adversary can still make use of exploited popular websites, and have a large number of innocent users to solve the distorted images. However, for Uvauth captchas, only recognizing all characters is not enough, and semantic interpretation is required to learn whether the feedback is positive or negative. We discuss few captcha variants in Section 7.4.2 that may make regular captchas more difficult for machine attackers.

7.4.2 Adaptation of regular captchas

For regular captchas, the content can be arbitrary and randomized, without carrying any meaningful information, e.g., an irrelevant mix of letters and numbers. However, for Uvauth, we need to transmit messages in natural languages with predefined meanings for conveying authentication results. Existing captcha breaking techniques (e.g., [85]) would perform even better against Uvauth captchas due to the limited entropy of our messages (resulting mostly from the fixed nature of the messages). To address this, the captcha generation may be adapted as follows.

a) Randomized padding. Humans have the ability to semantically interpret a message even if the message is garbled to some extent. Most people do not read all the characters in a word, or even all the words in a sentence (see e.g., [225]). As an example, consider the following sentences: "hke It uu is qKd k9l2 fine vMab weather.", "If You Can Raed Tihs, You Msut Be Raelly Smrat"; in most cases, humans can understand the meaning without much difficulty, but for machines it is not straightforward to extract the meaning from these sentences, especially when such messages appear in a distorted image. As an example, see Figure 13.

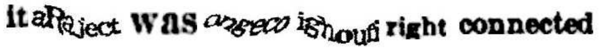


Figure 13: Distorted image with random padding

b) Indirect expression. Emotional tones in indirect positive or negative expressions such as "Everything goes well!" (correct password entry), or "Your password makes me angry!" (incorrect password entry) are quite self-evident for humans, but not so straightforward for machines. Existing work shows that machines can also learn to identify emotions in text (e.g., [260]), but requires non-trivial resources (e.g., a large knowledge database).

c) Display anywhere. Automated attacks on a captcha somewhat depends on the ability to locate the captcha on a screen. In regular usage, captchas are generally placed in a deterministic location, to facilitate the ease of processing by human users. As Uvauth's communication channel is one-way (i.e., no response back from the user), the distorted image can be placed anywhere on the screen (as long as it is visible to the user). It can also be embedded into a larger bitmap (e.g., banners, ads, backgrounds) to make automated identification difficult. Random delays (e.g., few seconds) may also be added before displaying the message (after the authentication phase), to frustrate the attacker even further.

7.4.3 An example with VNC

We now discuss how adapted distorted images may be used with a Virtual Network Computing (VNC [229]) application for remote desktop management.[3] When the remote machine is not personal to the user (e.g., accessed as a sysadmin), login feedback via distorted images may be used. Figure 14 shows a VNC session when an adapted distorted image (with "display anywhere") is used for authentication feedback. Here a legitimate user may expect such a string to be displayed anywhere on the screen. In contrast, for a machine attacker, it may be difficult to identify the distorted message from a screen-capture, specifically, when the message is blended with the background. Additionally, there is no need to display the distorted image right after login; e.g., a short, random delay can be added to confuse the attacker even further. The attacker may need to forward a video clip to a human solver to perform a relay attack, which would increase the cost of such an attack.

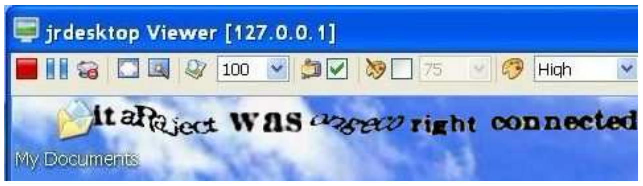

jrdesktop Viewer [127.0.0.1]
100
75
High
it a Reject was aagent right connected
My Documents

Figure 14: A VNC session with an adapted distorted image

7.5 Limitations and Attacks

In this section, we evaluate Uvauth from an attacker's perspective and list possible attacks. Some of these attacks can be mitigated if special care is taken, while others are limitations of our current design.

a) Post-and-check attacks. For certain accounts, attackers can first post a message to the target account, and then check for the posted message when launching a guessing attack on that account. For example, an attacker can post a comment on the target's Facebook account, and by checking whether this specific post is seen, the detection of a fake session becomes easier. Similarly, an email can be sent to a victim's Gmail account for the same purpose, and the attacker then just checks whether the email has been received when in the fake session. We term these attacks as post-and-check attacks, which can be automated and can make designing fake sessions

[3] `jrDesktop`, see: `http://jrdesktop.sourceforge.net`

significantly difficult. Application-specific defenses can be designed. For example, for a fake Facebook session, the target user's publicly-visible content, including posts from non-friends should be used. Assuming the attacker is not socially-connected to the user (i.e., not a Facebook friend), post-and-check attacks can be restricted.

Designing a similar mechanism for email is less straightforward, as no explicit social connections exist in email. However, email services (e.g., Gmail) are currently quite effective against spam email accounts; recently-received spam emails for a targeted account can be used in fake sessions (albeit with the risk of some information leakage, as sometimes legitimate emails are labelled as spam). Attackers also must send an email to the target account immediately before launching the guessing attack; otherwise, they would not know whether the target user has deleted the unwanted email, or they are in a fake session. Emails received from first-time contacts in a recent period (e.g., in the last five minutes) may be included in fake sessions. This can restrict post-and-check attacks for email accounts, at the expense of occasional information leakage. Email contacts as displayed in a fake session could also be problematic. If fake email addresses are used, by sending emails to these addresses, an attacker may identify the fake session (e.g., if an immediate delivery failure message is received). On the other hand, the use of real email addresses would cause obvious privacy exposure (e.g., harvesting of emails).

b) Targeted attacks. If an attacker knows a victim in person (real-world or online-only), she may also know one or more contacts in the victim's Facebook friend list, or the account number / address for online banking. When such information can be expected by the attacker, a fake session can be easily detected. We focus on restricting large-scale automated guessing attacks, and exclude targeted attacks (although these attacks may also be significant in some scenarios; see e.g., [157]).

c) New denial-of-service attacks. Uvauth fake sessions may be exploited to launch algorithmic/complexity-based DoS attacks (e.g., [63]). An adversary can initiate many fake sessions with resource-intensive operations on the server-side to overload the server, e.g., text search in an email account. So fake sessions must be designed carefully, and the allowed activities therein should not consume too much resources; i.e., the trade-off between verisimilitude and resource consumption must be chosen with care.

d) Adapted relay attacks. Paid human solver services (e.g., as discussed in [191]) can be used to attack Uvauth messages that rely on adapted captchas. We can alleviate such risk by applying "Display Anywhere", so that the attacker has to forward the whole screen or even a video clip to the human solvers which incurs more effort.

e) Inconsistency attacks. If states in fake sessions are not saved, then the attacker may detect a fake session by making some changes to it, and checking for those changes after a re-login. This is a known problem in deception, and referred as inconsistency of deception. Neagoe and Bishop [202] argue that even inconsistent deception can still effectively confuse an attacker.

f) Acquired targeted attacks.[4] Assume that a random attacker wants to guess the password for a specific account A and the attacker has already compromised another account B from the same user (on the same or a different website). Also assume that the password for A is different than that of B. Now, similar to the targeted attacks discussed above, the attacker can use extracted information from B to detect fake sessions for A. Note that, the attacker may need only temporary / one-time access to B. If the attacker can successfully guess the password for A, she can now use information from both accounts to brute-force other accounts from the same user (even when password reuse is avoided). As users generally maintain several password-protected accounts, this attack may be quite realistic—e.g., through the compromise of a large-scale, popular service provider (for some recent incidents, see e.g., [205]).

g) Legitimate users in a honeypot.[4] If an attacker succeeds in compromising an account (e.g., through password guessing), she could then (maliciously) change the password, e.g., to keep the account in her control and deny access to the legitimate user. Now, when the user tries to log in with the old password, he will be confused; by not seeing his data, the user might assume that he has mistyped the password, and keep trying several times before realizing the attack. Without Uvauth, the user will be denied access, and possibly try account recovery methods immediately.

A similar issue arises even when the account password remains uncompromised. If an incorrect password is tried (e.g., due to typos), users must detect the resulting fake session, and then log out for another attempt. Such wrong password entries would cost more time for users due to the additional step of detecting fake sessions. This usability issue is a side-effect of Uvauth, and does not happen with an explicit feedback, as in regular authentication. Note that typos can be avoided by displaying the password in cleartext (cf. [207]), specifically when shoulder-surfing is not an issue (e.g., the user sitting alone in her office). However, misremembered passwords may not be readily detected by such password unmasking, and the user may still be delayed in discovering the situation, partly due to Uvauth's fake sessions.

[4]An anonymous NSPW2013 reviewer pointed us this attack.

Other limitations include: we have not evaluated the server-side load for generating and running a large number of fake sessions. We also have not tested how effectively users can detect implicit results from an authentication attempt, or whether messages via adapted distorted images can be used in practice.

7.6 Related Work

Uvauth falls in the intersection of password security and deception techniques. Here we highlight a few related projects from both areas.

Pinkas and Sander [219] first proposed the use of Reverse Turing Tests (RTTs, e.g., captchas) to restrict large-scale online password dictionary attacks. The protocol challenges users with RTTs for a small fraction (e.g., 5%) of all possible userid-password pairs to reduce the server-load (of generating RTTs) and usability impact (of answering RTTs), while keeping the cost of launching a large-scale guessing attack significantly high. Correct passwords always require an RTT, unless a valid cookie from past successful login is found. In Uvauth, deploying fake sessions only for a small fraction of all login attempts, will also significantly reduce server-side load. However, if attackers use a small password dictionary (e.g., top 500 words), the number of fake sessions they must process may be too small to provide any significant protection. Assuming many users use common/weak passwords that may be found in small dictionaries, we recommend the use of fake sessions for all failed login attempts.

Later RTT-based proposals further improved security and usability aspects of the original Pinkas and Sander [219] scheme. For example, the password guessing resistant protocol (PGRP [13]), where more RTTs are imposed on *unknown* (possibly attack) machines than *known* (possibly legitimate) ones; machines are categorized using source IP addresses and cookies. As discussed in Section 7.3.1, item (e), the use of known devices may reduce the number of fake sessions for legitimate users. Unlike RTT-based schemes, Uvauth does not provide explicit authentication feedback, and avoids challenging users with RTTs. Recall that, even for our use of distorted images as a communication channel, we do not require a response from the user.

Goyal et al. [99] extend the *pricing via processing* paradigm (introduced by Dwork and Naor [73]) to address online password guessing; the proposed protocol (*CompChall*) imposes a significant amount of computation for the client on each authentication attempt. CompChall would not adversely affect legitimate users since their authentication attempts are expected to be limited. In contrast, the scheme may negatively impact an attacker when a large number of attempts are made from a single machine. However, CompChall may not be effective against attacks from a botnet.

The idea of closely monitored network decoys (honeypots), to distract/deceive adversaries from real targets and to collect analytical information about an attack, has more than two decades of history (see e.g., [259, 53]). Our methodology resembles honeypots in the sense that the attacker is also given deliberate access, and fed with false information. However, in contrast to honeypots, our use of deception focuses on hiding the result of an authentication attempt, instead of detecting/analyzing malicious activities. Similar to the generation of fake sessions in Uvauth, the deployment of a honeypot is also time-consuming and resource-intensive. Provos designed Honeyd [221], a framework for virtual honeypots that simulates virtual computer systems at the network level. It saves physical resources in terms of resource consolidation and tolerance of high destructiveness. Additionally, it is more flexible to configuration changes, and thus allows more complicated behaviors to be implemented. Uvauth's fake session generation may benefit from such existing honeypot work.

Herley and Florêncio [114] propose the use of honeypot credentials to restrict brute-force guessing attacks on online banking accounts. During account creation, for each userid, a large number of honeypot passwords (n, a subset of all possible passwords) are also registered along with the correct password. The userid with honeypot passwords are considered honeypot credentials, and all such credentials will lead to honeypot sessions, which are especially tracked by the bank server for money transfer attempts. To reduce the probability of mistyping by a real user, all honeypot passwords are chosen to be more than two characters apart from the correct password; however, a brute-force attacker is still n times more likely to try a honeypot password. Honeypot sessions are created from real user data (e.g., attributes, transactions) with fake identification information such as names and addresses. In comparison, Uvauth's scope is broader, and it considers the use of small password dictionary with known userids (instead of trying all possible entries from the userid-password space).

Pavlovic [216] re-visits the idea of security by obscurity, assuming attackers, like defenders, also have limited logical or programming resources. It is argued that the behaviour of defenders can also be hidden to gain tangible security advantages. Uvauth's use of deception is limited to hiding only the defenders' verification outcome from attackers.

Most work on deception focuses on maintaining consistency of the false reality as presented to attackers. Neagoe and Bishop [202] explore inconsistent deception for defending computer resources, and argue that these techniques may still be effectively used to track and monitor attackers. Forgoing consistency may also make the design of deception techniques simpler and less resource-intensive. Such techniques may significantly reduce the cost of deploying fake sessions in Uvauth.

Clark and Hengartner introduced panic password [57], where a separate password is used to indicate a duress situation to the server without soliciting an authentication failure; the primary goal is to protect both the victim's safety and sensitive information residing on the server. On the entry of a panic password, the observable response is to deceive the adversary with panic responses that are indistinguishable from the real response. While panic passwords are proposed to be used by a legitimate user under duress, Uvauth is targeted towards protecting passwords from being guessed using a botnet, or by (random) human-assisted attackers.

Juels and Rivest recently proposed *honeywords* [145] (false passwords) to address offline attacks against hashed password databases. For each account, the legitimate password is mixed with several honeywords; thus, when an attacker cracks a hashed password, she cannot be sure if it is the real password or a honeyword. Also, the use of a honeyword will trigger an alarm on the server-side (cf. panic password).

7.7 Conclusion

We propose Uvauth to hide authentication results from attackers to mitigate the risk of online password guessing. It can effectively deceive an attacker assuming fake sessions can be efficiently generated (as an attacker may launch many authentication attempts from a large-scale botnet). Most current authentication schemes would fail to an adversary who is willing to use human help to break into existing techniques that are designed to limit only automated attacks. As user accounts generally become more and more valuable with the duration of use, it may be worthwhile for attackers to invest in cheap human labor as a means to compromise user credentials. In designing Uvauth, we explicitly consider such threats and provide limited protection (possibly significantly more than existing technologies). Implementing Uvauth fake sessions would require server-side support, but no changes are needed on the client-side software or existing password input UI (including browser mechanisms such as "keep me logged in" and cookies). However, Uvauth, as presented, has not been fully evaluated, and has a number of limitations. Our goal is to attract attention to an important drawback of existing authentication schemes that enables large-scale guessing attacks.

Chapter 8

Further Discussion

8.1 Onto Mobile Platforms

Following up on Gracewipe and Hypnoguard and in considering an increasing amount of invaluable personal/business data is now being stored on mobile devices, we have also started porting Gracewipe over to mobile platforms, i.e., ARM-based devices, which we name Gracewipe Mobile. Likewise, selected user data is protected with full-disk encryption or file-based encryption (starting from Android 7.0) [94], and at boot-time or when a specific file is accessed, the user should be brought to a secure interface where she can type the (deletion/unlocking) password as in Gracewipe. Any (system) processes or even the firmware should not be able to see/access the protected files outside the secure world (i.e., of ARM TrustZone [206]) or without the user passwords.

In practice, there does not exist an actual distinction of data-at-rest and data-in-sleep, as many services require constant execution (e.g., as with Push Mail) so that the Android wake locks [95] are intensively used (or abused). In consequence, a mobile device is very rarely in a real sleep state. Moreover, the ARM processor has multiple sleep levels (e.g., WFI or WFE [165]), with SoC-specific implementation, causing user data protection to be more complicated. Therefore, in the end it will be a mix of Gracewipe and Hypnoguard with more adaptations for categorized use scenarios.

We have verified our primitive design with certain experimentation. The effort mainly involves two aspects:

1. Porting the core Gracewipe logic over to the secure world of the ARM processor. Different from TXT on the x86 platform, ARM TrustZone merely provides the processor support with certain interfacing specifications (defined by GlobalPlatform, e.g., `TEE_OpenPersistentObject`). We need a special OS running in the secure world eventually provding access to the SE for sealing secrets (e.g., KH). There are very few available secure TEE OSes in the open community (we used OP-TEE [292]). Many commercial ones are proprietary and unavailable to the public developers/researchers, such as Trustonic [32], TrustKernel (Pingbo) [218], BeanPod ISEE (Beanpodtech) [267], TEE-WatchTrust (WatchData) [101], TCore (Nutlet Technology) [268], and iTrustee (Huawei). See the discussion on SE below. With OP-TEE, we are currently only able to verify the functionality by emulation.

2. Porting the Gracewipe interface to the Android OS (normal world). With FDE, we opt to instrument `CryptKeeper.java` by replacing `DecryptTask().execute(password)` with our `Gracewipe.UI.logic`. Upon successful authentication, we continue with `DecryptTask().execute(KH)`. In this way, the original user password (strength needed) is replaced with Gracewipe's platform binding with a relative weak password. This partial prototype implementation can serve as a starting point for future work.

8.2 Open Problems

Among other potential paradigms for addressing our identified unconventional threats, we pursue hardware-software orchestration, where trusted computing is the primary focus (but not alone). In a broader sense, it is like a privilege race, e.g., a hypervisor-level (-1) mechanism should be effective against guest kernel-level (0) threats and so forth. Therefore, undoubtedly hardware can attain the lowest protection level (highest privilege) in the battle with various adversaries. Existing hardware-enforced mechanisms still have room for improvements in the following aspects: 1) Trust anchor. In trusted computing, the first link of the trust chain is usually an immutable and protected secret that binds to the hardware (the place storing the unique secrets, which is called Secure Element in certain terminology). On mobile platforms (e.g., TrustZone), SE is always individual vendor's proprietary design and implementation [292] (for example, Huawei has their unique inSE solution [148] on the Kirin 960 SoC, with

HiSEC V100 as its security core and an OS there called HiCOS), which is inaccessible to the open community and lacks auditability. In reality, a bigger issue is that the root of trust is often a combination, e.g., firmware, multiple levels of bootloader, and other vendor components (e.g., Intel's ACM) depending on how the trusted execution is launched. 2) Ecosystem. For both desktop and mobile platforms, the application of trusted computing still remains rare or only by certain manufacturers (e.g., Samsung). Intel SGX is an exception as mentioned above; but its attestation process is highly dependent on Intel. In this direction, closer collaboration with device manufacturers is unavoidable. This becomes a significant barrier for the academic research due to lack of access to hardware primitives or public documentation. The aforementioned status quo might not be all research problems, but the academic community can help with further formalization and standardization of TEEs, in terms of at least public validation (if open-source hardware is still yet to come) of hardware security primitives.

8.3 Concluding Remarks

In this thesis, we propose the notion of unconventional threats (or a strong adversarial model) to represent a series of realistic attack vectors which have not been thoroughly addressed in the literature or industry. To demonstrate the feasibility and actuality thereof, a number of representative scenarios are selected, distributed in different aspects of the cyber-world. We admit that it is nontrivial to address them and there might not be even quasi-ideal solutions. More specifically for example, we have to make unconventional assumptions sometimes to come up with a solution, whereas "normal" assumptions are usually enough for addressing conventional threats.

We introduce two major principles, i.e., hardware assistance and passive defense, in the battle with such threats; design and implement several solution prototypes corresponding to the selected scenarios. Although certain limitations are still persistent (as discussed in individual chapters), the prototypes serve as a showcase of the effectiveness of hardware primitives and the philosophy of passive defense. Our research points to a direction where the improvements can be made based on the two principles for coping with more unconventional threats in the future.

Bibliography

[1] FAT16/32 file system library. http://ultra-embedded.com/releases/fat_io_lib.zip.

[2] The history of Cryptowall: a large scale cryptographic ransomware threat. https://www.cryptowalltracker.org/.

[3] Library and utilities for manipulating TCG Opal compliant self-encrypting hard drives. https://github.com/tparys/topaz-alpha.

[4] List of USB flash drives with hardware write protection. https://www.fencepost.net/2010/03/usb-flash-drives-with-hardware-write-protection/.

[5] magic - file command's magic pattern file. https://linux.die.net/man/5/magic.

[6] The no more ransom project. https://www.nomoreransom.org/en/index.html.

[7] Programmed I/O (PIO) modes. http://www.pcguide.com/ref/hdd/if/ide/modes_PIO.htm.

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

[9] 16s.us. TCHunt. Tool for detecting encrypted hidden volumes (version: 1.6, release date: Jan. 29, 2014). `http://16s.us/software/TCHunt/`.

[10] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel virtualization technology for directed i/o. *Intel technology journal*, 10(3), 2006.

[11] ACPI.info. Advanced configuration and power interface specification. Revision 5.0a (Nov. 13, 2013). `http://www.acpi.info/spec.htm`.

[12] G. Alendal, C. Kison, and modg. got hw crypto? on the (in) security of a self-encrypting drive series. *IACR Cryptology ePrint Archive*, 2015:1002, 2015.

[13] M. Alsaleh, M. Mannan, and P. van Oorschot. Revisiting defenses against large-scale online password guessing attacks. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 9(1):128–141, 2012.

[14] AMD. AMD64 architecture programmer's manual volume 2: System programming. Technical article (May 2013). `http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf`.

[15] AMD.com. AMD64 architecture programmer's manual volume 2: System programming. Revision 3.25 (June 2015). `http://support.amd.com/TechDocs/24593.pdf`.

[16] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for CPU based attestation and sealing. In *Hardware and Architectural Support for Security and Privacy (HASP'13)*, Tel-Aviv, Israel, June 2013.

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

[18] Apple.com. iOS remote wipe. Available at `https://www.apple.com/business/docs/iOS_Security_Guide.pdf`. Accessed: 2016-11-01.

[19] Apple.com. ios security guide, 2018. White Paper. Available at `https://www.apple.com/business/docs/iOS_Security_Guide.pdf`.

[20] T. W. Arnold and L. P. V. Doorn. The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer. *IBM J. RES. & DEV*, 48(3/4), MAY/JULY 2004.

[21] ArsTechnica. New and improved CryptXXX ransomware rakes in $45,000 in 3 weeks. News article (June 27, 2016). `https://arstechnica.com/information-technology/2016/06/new-and-improved-cryptxxx-ransomware-rakes-in-45000-in-3-weeks/`.

[22] ArsTechnica.com. Drug dealer: Cops leaned me over 18th floor balcony to get my password. News article (Apr. 22, 2015).

[23] ArsTechnica.com. Microsoft may have your encryption key; here's how to take it back. News article (Dec. 29, 2015).

[24] J. Aycock. Transformitt. *Leonardo*, 46(5):482–483, Oct. 2013.

[25] E. Ayday, J. L. Raisaro, M. Laren, P. Jack, J. Fellay, and J.-P. Hubaux. Privacy-preserving computation of disease risk by using genomic, clinical, and environmental data. In *Proceedings of USENIX Security Workshop on Health Information Technologies (HealthTech" 13)*, number EPFL-CONF-187118, 2013.

[26] J. Azema and G. Fayad. M-Shield mobile security technology: making wireless secure. Technical report, Texas Instruments, 2008.

[27] G. Bakos and S. Bratus. Ubiquitous redirection as access control response. In *Annual Conference on Privacy, Security and Trust (PST'05)*, St. Andrews, NB, Canada, October 2005.

[28] P. Baldi, R. Baronio, E. De Cristofaro, P. Gasti, and G. Tsudik. Countering gattaca: efficient and secure testing of fully-sequenced human genomes. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 691–702. ACM, 2011.

[29] Bank of America. SiteKey authentication: An additional layer of online and mobile banking security. `https://www.bankofamerica.com/privacy/online-mobile-banking-privacy/sitekey.go`.

[30] G. V. Bard. Spelling-error tolerant, order-independent pass-phrases via the damerau-levenshtein string-edit distance metric. In *Australasian Information Security Workshop (AISW'07)*, Ballarat, Australia, 2007.

[31] M. Bellare, V. T. Hoang, and P. Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In *ASIACRYPT*, 2012.

[32] J. Bennett. Devices with trustonic tee, 2015.

[33] M. Bishop, R. Crawford, B. Bhumiratana, L. Clark, and K. Levitt. Some problems in sanitizing network data. In *IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2006)*, Manchester, UK, June 2006.

[34] M. Bishop, J. Cummins, S. Peisert, A. Singh, B. Bhumiratana, D. Agarwal, D. Frincke, and M. Hogarth. Relationships and data sanitization: A study

in Scarlet. In *New Security Paradigms Workshop (NSPW'10)*, Concord, MA, USA, Sept. 2010.

[35] E.-O. Blass and W. Robertson. TRESOR-HUNT: Attacking CPU-bound encryption. In *ACSAC'12*, Orlando, FL, USA, Dec. 2012.

[36] B. Böck. Firewire-based physical security attacks on windows 7, EFS and BitLocker. Secure Business Austria Research Lab. Technical report (Aug. 13, 2009). `https://www.helpnetsecurity.com/dl/articles/windows7_firewire_physical_attacks.pdf`.

[37] A. Boileau. Hit by a bus: Physical access attacks with Firewire. Ruxcon 2006. `http://www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf`.

[38] 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.

[39] 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.

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

[41] A. Broadbent, G. Gutoski, and D. Stebila. Quantum one-time programs. In *CRYPTO*, pages 344–360, 2013.

[42] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, Aug. 2005.

[43] E. Bursztein, S. Bethard, J. C. Mitchell, D. Jurafsky, and C. Fabry. How good are humans at solving CAPTCHAs? A large scale evaluation. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2010.

[44] E. Bursztein, M. Martin, and J. C. Mitchell. Text-based CAPTCHA strengths and weaknesses. In *ACM Computer and Communications Security (CCS'11)*, Chicago, IL, USA, Oct. 2011.

[45] K. R. B. Butler, S. McLaughlin, and P. D. McDaniel. Rootkit-resistant disks. In *ACM Computer and Communications Security (CCS'08)*, Alexandria, Virginia, USA, 2008.

[46] Calomel.org. AES-NI SSL performance: A study of AES-NI acceleration using LibreSSL, OpenSSL. Online article (Feb. 23, 2016). `https://calomel.org/aesni_ssl_performance.html`.

[47] M. Canim, M. Kantarcioglu, and B. Malin. Secure management of biomedical data with cryptographic hardware. volume 16, pages 166–175, 2012.

[48] R. Carbone, C. Bean, and M. Salois. An in-depth analysis of the cold boot attack: Can it be used for sound forensic memory acquisition?, Jan. 2011. Technical Memorandum (TM 2010-296), Defence Research and Development Canada (DRDC), Valcartier.

[49] M. Cariaso and G. Lennon. Snpedia, 2010.

[50] G. Chappell. The x86 BIOS emulator, 2010. `http://www.geoffchappell.com/studies/windows/km/hal/api/x86bios/index.htm?tx=7`.

[51] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova. Evaluation of the Intel® core$^{\text{TM}}$ i7 Turbo Boost feature. In *IEEE International Symposium on Workload Characterization (IISWC'09)*, Austin, TX, USA, Oct. 2009.

[52] R. Chatterjee, J. Bonneau, A. Juels, and T. Ristenpart. Cracking-resistant password vaults using natural language encoders. In *IEEE Symposium on Security and Privacy*, San Jose, CA, USA, May 2015.

[53] B. Cheswick. An evening with Berferd, in which a cracker is lured, endured, and studied. In *Winter USENIX Conference*, San Francisco, CA, USA, Jan. 1992.

[54] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 41–50. IEEE, 1995.

[55] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *USENIX Security Symposium*, Baltimore, MD, USA, Aug. 2005.

[56] L. S. Clair, J. Schiffman, T. Jaeger, and P. McDaniel. Establishing and sustaining system integrity via root of trust installation. In *Proceedings of the 2007 Annual Computer Security Applications Conference*, pages 19–29, Dec. 2007.

[57] 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.

[58] 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`.

[59] F. Cohen. The use of deception techniques: Honeypots and decoys. *The Handbook of Information Security*, 3:646–655, 2006.

[60] A. Continella, A. Guagnelli, G. Zingaro, G. De Pasquale, A. Barenghi, S. Zanero, and F. Maggi. ShieldFS: A self-healing, ransomware-aware filesystem. In *Annual Conference on Computer Security Applications (ACSAC'16)*, Los Angeles, CA, USA, 2016.

[61] H. Crawford and J. Aycock. Kwyjibo: automatic domain name generation. *Software: Practice and Experience*, 38(14):1561–1567, 2008.

[62] 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.

[63] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *USENIX Security Symposium*, Washington, DC, USA, Aug. 2003.

[64] Cross-Disorder Group of the Psychiatric Genomics Consortium et al. Genetic relationship between five psychiatric disorders estimated from genome-wide snps. *Nature genetics*, 45(9):984–994, 2013.

[65] 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.

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

[67] Dell.com. Dell remote data delete service. Available at `http://www.dell.com/downloads/global/services/Dell_ProSupport_Laptop_Tracking_and_Recovery_and_Remote_Data_Delete_ABU-EMEA_FINAL.pdf`. Accessed: 2016-11-01.

[68] R. Dhamija and J. D. Tygar. The battle against phishing: Dynamic security skins. In *Symposium on Usable Privacy and Security (SOUPS'05)*, Pittsburgh, PA, USA, July 2005.

[69] S. M. Diesburg and A.-I. A. Wang. A survey of confidential data storage and deletion methods. *ACM Computing Surveys (CSUR)*, 43(1):2:1–2:37, 2010.

[70] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2), Mar. 1983.

[71] Drive Trust Alliance. DTA sedutil self encrypting drive software. `https://github.com/Drive-Trust-Alliance/sedutil`.

[72] Dropbox.com. Dropbox remote wipe. Available at `https://www.dropbox.com/help/4227`. Accessed: 2016-11-01.

[73] C. Dwork and M. Naor. Pricing via processing or combating junk mail. In *Advances in Cryptology - CRYPTO'92*, Santa Barbara, CA, USA, August 1992.

[74] Elcomsoft.com. Elcomsoft forensic disk decryptor: Forensic access to encrypted BitLocker, PGP and TrueCrypt disks and containers. `https://www.elcomsoft.com/efdd.html`.

[75] M. Ermolov and M. Goryachy. How to hack a turned-off computer, or running unsigned code in Intel management engine. To be presented at Blackhat Europe 2017 in December.

[76] F-Secure. 2017 F-Secure state of cyber security. `https://www.f-secure.com/documents/996508/1030743/cyber-security-report-2017`.

[77] F-Secure. The state of cyber security 2017. Technical report (Feb. 16, 2017). https://www.f-secure.com/documents/996508/1030743/cyber-security-report-2017.

[78] S. Feaster. FAT32-implementation. https://github.com/ShamariFeaster/FAT32-Implementation.

[79] A. Filyanov, J. M. McCuney, A.-R. Sadeghiz, and M. Winandy. Uni-directional trusted path: Transaction confirmation on just one device. In *IEEE/IFIP Dependable Systems and Networks (DSN'11)*, Hong Kong, June 2011.

[80] R. A. Fink, A. T. Sherman, A. O. Mitchell, and D. C. Challener. Catching the cuckoo: Verifying tpm proximity using a quote timing side-channel. In J. M. McCune, B. Balacheff, A. Perrig, A.-R. Sadeghi, A. Sasse, and Y. Beres, editors, *Trust and Trustworthy Computing*, pages 294–301, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[81] B. A. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov. Iron: Functional encryption using intel sgx. 2016.

[82] Forensicswiki.org. Tools:memory imaging. http://www.forensicswiki.org/wiki/Tools:Memory_Imaging.

[83] M. Frank, R. Biedert, E. Ma, I. Martinovic, and D. Song. Touchalytics: On the applicability of touchscreen input as a behavioral biometric for continuous authentication. *IEEE TIFS*, 8(1):136–148, Jan. 2013.

[84] 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.

[85] Futurity.org. Gotcha! captcha security flaws revealed. Available at: `http://www.futurity.org/science-technology/gotcha-captcha-security-flaws-revealed/`.

[86] B. Garmany and T. Müller. PRIME: Private RSA infrastructure for memory-less encryption. In *ACSAC'13*, New Orleans, LA, USA, 2013.

[87] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: an auditing file system for theft-prone devices. In *EuroSys'11*, Salzburg, Austria, 2011.

[88] 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.

[89] D. Genkin, I. Pipman, and E. Tromer. Get your hands off my laptop: physical side-channel key-extraction attacks on PCs. *Journal of Cryptographic Engineering*, 5(2):95–112, Jun 2015.

[90] L. Giles. *Sun Tzu on the Art of War: The Oldest Military Treatise in the World.* London Luzac, 1910. Chapter 1: verse 18.

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

[92] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. One-Time Programs. In *CRYPTO*, pages 39–56, 2008.

[93] Google.com. Android remote wipe. Available at `https://support.google.com/a/answer/173390?hl=en`. Accessed: 2016-11-01.

[94] Google.com. Android full disk encryption, 2018. Available at `https://source.android.com/security/encryption/`.

[95] Google.com. Keep the device awake, 2018. Available at `https://developer.android.com/training/scheduling/wakelock`.

[96] J. Götzfried and T. Müller. Mutual authentication and trust bootstrapping towards secure disk encryption. *ACM Transactions on Information and System Security (TISSEC)*, 17(2):6:1–6:23, 2014.

[97] J. Götzfried and T. Müller. Mutual authentication and trust bootstrapping towards secure disk encryption. *ACM TISSEC*, 17(2):6:1–6:23, Nov. 2014.

[98] Gov1.info. NSA ANT product catalog. `https://nsa.gov1.info/dni/nsa-ant-catalog/`.

[99] V. Goyal, V. Kumar, M. Singh, A. Abraham, and S. Sanyal. Compchall: Addressing password guessing attacks. In *International Symposium on Information Technology: Coding and Computing (ITCC'05)*, Las Vegas, NV, USA, April 2005.

[100] B. Greshake, P. E. Bayer, H. Rausch, and J. Reda. Opensnp–a crowdsourced web resource for personal genomics. *PLoS One*, 9(3):1–9, 2014.

[101] W. Group. Watchdata presents its highlight watchtrust and sharkey solutions at mwc 2015, 2015. Available at `http://www.watchdata.com/newsshow.php?cid=336&id=76`.

[102] M. Gruhn and T. Müller. On the practicability of cold boot attacks. In *Conference on Availability, Reliability and Security (ARES'13)*, Regensburg, Germany, Sept. 2013.

[103] L. Guan, J. Lin, B. Luo, and J. Jing. Copker: Computing with private keys without RAM. In *NDSS'14*, San Diego, CA, USA, Feb. 2014.

[104] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *IEEE Symposium on Security and Privacy*, San Jose, CA, USA, May 2015.

[105] V. Gunupudi and S. R. Tate. Generalized non-interactive oblivious transfer using count-limited objects with applications to secure mobile agents. In *Financial Cryptography and Data Security*, FC'08, pages 98–112, 2008.

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

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

[108] 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.

[109] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols*. Springer, 2010.

[110] HealthcareITNews. Ransomware: See the 14 hospitals attacked so far in 2016. News article (Oct. 5, 2016). `http://www.healthcareitnews.com/slideshow/ransomware-see-hospitals-hit-2016`.

[111] C. Helfmeier, D. Nedospasov, C. Tarnovsky, J. Krissler, C. Boit, and J.-P. Seifert. Breaking and entering through the silicon. In *ACM CCS'13*, Berlin, Germany, Nov. 2013.

[112] B. Hemenway, Z. Jafargholi, R. Ostrovsky, A. Scafuro, and D. Wichs. Adaptively Secure Garbled Circuits from One-Way Functions. In *CRYPTO*, pages 149–178, 2016.

[113] M. Henson and S. Taylor. Memory encryption: A survey of existing techniques. *ACM Computing Surveys (CSUR)*, 46(4):53:1–53:26, Mar. 2014.

[114] C. Herley and D. Florencio. Protecting financial institutions from brute-force attacks. In *Proceedings of The Ifip Tc 11 23rd International Information Security Conference*, volume 278, pages 681–685. Springer US, 2008.

[115] HGST.com. Data center and enterprise storage solutions. `https://www.hgst.com/sites/default/files/resources/DC-Ent-StorageSolutions-BR.pdf`.

[116] G. Hoffmann. Intel AMT SoL client + tools. Available at `https://www.kraxel.org/cgit/amtterm`. Accessed: 2016-11-01.

[117] T. Holz, M. Engelberth, and F. Freiling. Learning more about the underground economy: A case-study of keyloggers and dropzones. In *European Symposium on Research in Computer Security (ESORICS'09)*, Saint Malo, France, Sept. 2009.

[118] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure Two-party Computations in ANSI C. In *CCS*, pages 772–783, 2012.

[119] A. Horvath and J. M. Slocum. Memory bandwidth benchmark. Open source project. `https://github.com/raas/mbw`.

[120] HTBridge.com. RansomWeb: emerging website threat that may outshine DDoS, data theft and defacements? News article (Jan. 28, 2015). `https://www.htbridge.com/blog/ransomweb_emerging_website_threat.html`.

[121] A. Huang. Keeping secrets in hardware: The Microsoft Xbox^{TM} case study. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES'02)*, San Francisco, CA, USA, Aug. 2002.

[122] J. Huang, J. Xu, X. Xing, P. Liu, and M. K. Qureshi. FlashGuard: Leveraging intrinsic flash properties to defend against encryption ransomware. In *ACM Computer and Communications Security (CCS'17)*, Dallas, TX, USA, 2017.

[123] J. Ibsen. Simple C vector. `https://github.com/jibsen/scv`.

[124] IEEE.org. 1667-2015 - IEEE standard for discovery, authentication, and authorization in host attachments of storage devices. `https://standards.ieee.org/findstds/standard/1667-2015.html`.

[125] Imod Digital. Social network numbers, May 2012. `http://www.imoddigital.com/iD-Social-Network-Statistics-2012-eBook.pdf`.

[126] Intel. Intel Trusted Execution Technology (Intel TXT): Measured launched environment developer's guide. Technical article (July 2015). `http://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf`.

[127] Intel. The MultiProcessor specification (MP spec), May 1997. `http://www.intel.com/design/archives/processors/pro/docs/242016.htm`.

[128] Intel.com. Intel active management technology start here guide (intel AMT 9.0). Available at `https://software.intel.com/sites/default/files/article/393789/amt-9-start-here-guide.pdf`. Accessed: 2016-11-01.

[129] Intel.com. Intel manageability commander. Available at `https://downloadcenter.intel.com/download/26375/Intel-Manageability-Commander`. Accessed: 2016-11-03.

[130] Intel.com. Intel TXT software development guide, measured launched environment developer's guide. https://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf.

[131] Intel.com. Intel vPro use case reference design - remote drive erase. Available at https://downloadcenter.intel.com/download/20971/Intel-vPro-Use-Case-Reference-Design-Remote-Drive-Erase. Accessed: 2016-11-03.

[132] Intel.com. Remote secure erase with intel AMT. Available at https://software.intel.com/en-us/blogs/2016/04/18/intel-amt-remote-secure-erase. Accessed: 2016-11-03.

[133] Intel.com. SMI transfer monitor (STM) user guide. Revision 1.00 (Aug. 2015). https://firmware.intel.com/sites/default/files/STM_User_Guide-001.pdf.

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

[135] Intel.com. Using Intel AMT serial-over-LAN to the fullest. Available at https://software.intel.com/en-us/articles/using-intel-amt-serial-over-lan-to-the-fullest. Accessed: 2016-11-01.

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

[137] Intel.com. *Intel TXT Software Development Guide - Measured Launched Environment Developer's Guide*, May 2014.

[138] IntelSecurity.com. Technical details of the S3 resume boot script vulnerability. Technical report (July 2015). http://www.intelsecurity.com/advanced-threat-research/content/WP_Intel_ATR_S3_ResBS_Vuln.pdf.

[139] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Fine grain cross-vm attacks on Xen and VMware. In *IEEE Conference on Big Data and Cloud Computing*, Washington, DC, USA, 2014.

[140] iSECPartners. YoNTMA (you'll never take me alive!). `https://github.com/iSECPartners/yontma`.

[141] 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`.

[142] Z. Jafargholi and D. Wichs. Adaptive Security of Yao's Garbled Circuits. In *TCC*, pages 433–458, 2016.

[143] K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs. In *CHES*, CHES'10, pages 383–397, 2010.

[144] S. Johnson. Intel® SGX and Side-Channels. `https://software.intel.com/en-us/articles/intel-sgx-and-side-channels`, 2017.

[145] A. Juels and R. L. Rivest. Honeywords: Making password-cracking detectable. Technical report (May 2, 2013). `http://people.csail.mit.edu/rivest/pubs/JR13.pdf`.

[146] M. Kantarcioglu, W. Jiang, Y. Liu, and B. Malin. A cryptographic approach to securely share and query genomic sequences. *IEEE Transactions on information technology in biomedicine*, 12(5):606–617, 2008.

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

[148] K. Kee. HuaweiâĂŹs kirin 960 summarized: What you need to know, 2016. Available at `https://nasilemaktech.com/huaweis-kirin-960-summarized-need-know/`.

[149] A. Kharaz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda. UNVEIL: A large-scale, automated approach to detecting ransomware. In *USENIX Security Symposium*, Austin, TX, 2016.

[150] A. Kharraz and E. Kirda. Redemption: Real-time protection against ransomware at end-hosts. In *Research in Attacks, Intrusions and Defenses (RAID'17)*, Atlanta, GA, USA, 2017.

[151] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, and E. Kirda. Cutting the Gordian knot: A look under the hood of ransomware attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Milan, Italy, 2015.

[152] J. King and A. dos Santos. A user-friendly approach to human authentication of messages. In *Financial Cryptography and Data Security (FC'05)*, Roseau, Dominica, February 2005.

[153] M. S. Kirkpatrick, S. Kerr, and E. Bertino. Puf roks: A hardware approach to read-once keys. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 155–164, Hong Kong, China, 2011.

[154] W. K. Kit. The complete book of tai chi chuan: A comprehensive guide to the principles and practice, 1998.

[155] T. Kitamura, K. Shinagawa, T. Nishide, and E. Okamoto. One-time Programs with Cloud Storage and Its Application to Electronic Money. In *APKC*, 2017.

[156] P. Kleissner. Stoned bootkit. Black Hat USA (July 2009). `http://www.blackhat.com/presentations/bh-usa-09/KLEISSNER/BHUSA09-Kleissner-StonedBootkit-PAPER.pdf`.

[157] Knowthenet.org.uk. More teenagers are being hacked by friends online but did you know it could be illegal? News article (Jan. 6, 2012). `http://www.knowthenet.org.uk/`.

[158] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, 2018.

[159] E. Kolodenker, W. Koch, G. Stringhini, and M. Egele. PayBreak: Defense against cryptographic ransomware. In *ACM Asia Conference on Computer and Communications Security (ASIACCS'17)*, pages 599–611, Abu Dhabi, UAE, 2017.

[160] B. Kreuter, A. Shelat, B. Mood, and K. Butler. PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation. In *USENIX Security Symposium*, pages 321–336, 2013.

[161] B. Kreuter, A. Shelat, and C. Shen. Billion-Gate Secure Computation with Malicious Adversaries. In *USENIX Security Symposium*, pages 285–300, 2012.

[162] A. Kumar, M. Patel, K. Tseng, R. Thomas, M. Tallam, A. Chopra, N. Smith, D. Grawrock, and D. Champagne. Method and apparatus to re-create trust model after sleep state, 2011. US Patent 7,945,786.

[163] K. S. Kuppusamy, S. R, and G. Aghila. A model for remote access and protection of smartphones using short message service. *International Journal of*

Computer Science, Engineering and Information Technology (IJCSEIT), 2(1), 2012.

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

[165] T. Lanier. Exploring the design of the cortex-a15 processor. *Available at* ***http://www.arm.com/files/pdf/atexploring the design of the cortex-a15.pdf***, 2013.

[166] B. Lapid and A. Wool. Cache-attacks on the ARM TrustZone implementations of AES-256 and AES-256-GCM via GPU-based analysis. In *Selected Areas in Cryptography (SAC) 2018*, Edmonton, Alberta, Canada, 2018.

[167] A. Laszka, S. Farhang, and J. Grossklags. On the economics of ransomware. *CoRR*, abs/1707.06247, 2017.

[168] M. D. Leom, K. K. R. Choo, and R. Hunt. Remote Wiping and Secure Deletion on Mobile Devices: A Review. *Journal of Forensic Sciences*, 61(6):1473–1492, nov 2016.

[169] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Cambridge, MA, USA, Nov. 2000.

[170] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *CoRR*, 2018.

[171] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, San Jose, CA, USA, May 2015.

[172] LogRhythm Labs. A technical analysis of WannaCry ransomware. Technical report (May 16, 2017). `https://logrhythm.com/blog/a-technical-analysis-of-wannacry-ransomware/`.

[173] T. Longstaff. Information about the PC CYBORG (AIDS) trojan horse. US DOE Computer Incident Advisory Capability (CIAC) information bulletin (Dec. 19, 1989). Wayback link: `https://web.archive.org/web/20060610040400/http://ciac.org/ciac/bulletins/a-10.shtml`.

[174] C. Maartmann-Moe. Inception. PCI-based DMA attack tool. `https://github.com/carmaa/inception`.

[175] T. Mandt, M. Solnik, and D. Wang. Demystifying the secure enclave processor. Technical report, Azimuth Security and OffCell Research, 2016. Black Hat Las Vegas.

[176] M. Mannan, B. H. Kim, A. Ganjali, and D. Lie. Unicorn: Two-factor attestation for data security. In *ACM CCS'11*, Chicago, IL, USA, Oct. 2011.

[177] N. Marketing. Departing employees and data theft, 2010.

[178] MasterCard. MasterCard inControl service now available from Barclaycard. News release (Jan. 21, 2010). `http://www.mastercard.com/us/company/en/newsroom/pr_mc_incontrol_service.html`.

[179] S. Matetic, K. Kostiainen, A. Dhar, D. Sommer, M. Ahmed, A. Gervais, A. Juels, and S. Capkun. Rote: Rollback protection for trusted execution. Technical report, ETH Zurich, 2017.

[180] Maximintegrated.com. Switching between battery and external power sources, 2002. `http://pdfserv.maximintegrated.com/en/an/AN1136.pdf`.

[181] McAfee Labs Threat Advisory. Ransomware-SAMAS. Technical article (Mar. 17, 2017). `https://kc.mcafee.com/resources/sites/MCAFEE/content/live/PRODUCT_DOCUMENTATION/26000/PD26873/en_US/McAfee_Labs_Threat_Advisory-Ransomware-SAMAS_v3.pdf`.

[182] J. M. McCune. *Reducing the trusted computing base for applications on commodity systems.* PhD thesis, Carnegie Mellon University, 2009.

[183] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys'08*, Glasgow, Scotland, Apr. 2008.

[184] 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.

[185] R. C. Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(4):294–299, 1978.

[186] Microsoft. Enable controlled folder access. Online documentation (Aug. 25, 2017). `https://docs.microsoft.com/en-us/windows/threat-protection/windows-defender-exploit-guard/enable-controlled-folders-exploit-guard`.

[187] Microsoft.com. BitLocker frequently asked questions (FAQ). Online article (June 10, 2014). `https://technet.microsoft.com/en-ca/library/hh831507.aspx`.

[188] Microsoft.com. Intune data wipe. Available at `https://docs.microsoft.com/en-us/intune/deploy-use/use-remote-wipe-to-help-protect-data-using-microsoft-intune`. Accessed: 2016-11-01.

[189] Microsoft.com. ProtectKeyWithTPM method of the Win32_EncryptableVolume class. Online reference. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa376470(v=vs.85).aspx`.

[190] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor. Frigate: A Validated, Extensible, and Efficient Compiler and Interpreter for Secure Computation. In *Euro-SP*, 2016.

[191] M. Motoyama, K. Levchenko, C. Kanich, D. McCoy, G. M. Voelker, and S. Savage. Re: CAPTCHAs – understanding CAPTCHA-solving services in an economic context. In *USENIX Security Symposium*, Washington, DC, USA, August 2010.

[192] T. Müller, A. Dewald, and F. C. Freiling. AESSE: A cold-boot resistant implementation of AES. In *European Workshop on System Security (EuroSec'10)*, Paris, France, Apr. 2010.

[193] T. Müller and F. C. Freiling. A systematic assessment of the security of full disk encryption. 12(5):491–503, September/October 2015.

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

[195] 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.

[196] T. Müller, B. Taubmann, and F. C. Freiling. TreVisor: OS-independent software-based full disk encryption secure against main memory attacks. In *Applied Cryptography and Network Security (ACNS'12)*, Singapore, June 2012.

[197] S. Mun. Making democracy legible: A defiant typeface. Blog post (June 20, 2013). `http://blogs.walkerart.org/design/2013/06/20/sang-mun-defiant-typeface-nsa-privacy/`. Project website: `http://z-x-x.org`.

[198] A. Narayanan and V. Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *ACM Computer and Communications Security (CCS'05)*, Alexandria, VA, USA, November 2005.

[199] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2008.

[200] M. Naveed, S. Agrawal, M. Prabhakaran, X. Wang, E. Ayday, J.-P. Hubaux, and C. Gunter. Controlled functional encryption. In *CCS*, pages 1280–1291. ACM, 2014.

[201] M. Naveed, E. Ayday, E. W. Clayton, J. Fellay, C. A. Gunter, J.-P. Hubaux, B. Malin, X. Wang, et al. Privacy and security in the genomic era. In *CCS*, 2014.

[202] V. Neagoe and M. Bishop. Inconsistency in deception for defense. In *New Security Paradigms Workshop (NSPW'06)*, Dagstuhl, Germany, Sept. 2006.

[203] M. Nemec, M. Sys, P. Svenda, D. Klinec, and V. Matyas. The return of Coppersmith's attack: Practical factorization of widely used RSA moduli. In *(to appear) ACM Computer and Communications Security (CCS'17)*, Dallas, TX, USA, 2017. `https://crocs.fi.muni.cz/public/papers/rsa_ccs17`.

[204] NetworkWorld.com. The latest ransomware threat: Doxware. News article (Feb. 27, 2017). `https://www.networkworld.com/article/3174678/security/the-latest-ransomware-threat-doxware.html`.

[205] NewYorker.com. The inevitable downfall of your password. News article (July 17, 2013). `http://www.newyorker.com/online/blogs/elements/2013/07/tumblr-vulnerability-how-to-secure-your-passwords.html`.

[206] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin. Trustzone explained: Architectural features and use cases. In *Collaboration and Internet Computing (CIC), 2016 IEEE 2nd International Conference on*, pages 445–451. IEEE, 2016.

[207] J. Nielsen. Stop password masking. Online article (June 23, 2009). `http://www.nngroup.com/articles/stop-password-masking/`.

[208] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, May 2013.

[209] not1337. amtterm patches. Available at `http://senseless.info/downloads.html`. Accessed: 2016-11-01.

[210] P. Oberoi and S. Mittal. Review of CIDS and techniques of detection of malicious insiders in cloud-based environment. *Cyber Security: Proceedings of CSI 2015*, 729:101, 2018.

[211] S. R. M. Oliveira and O. R. Zaiane. Protecting sensitive knowledge by data sanitization. In *IEEE International Conference on Data Mining (ICDM 2003)*, Melbourne, FL, USA, November 2003.

[212] E. T. Pancoast, J. N. Curnew, and S. M. Sawyer. Tamper-protected DRAM memory module, December 2012. US Patent 8,331,189.

[213] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011.

[214] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. Clamp: Practical prevention of large-scale data leaks. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2009.

[215] M. Patyal, S. Sampalli, Q. Ye, and M. Rahman. Multi-layered defense architecture against ransomware. *International Journal of Business & Cyber Security*, 1(2):52–64, Jan. 2017.

[216] D. Pavlovic. Gaming security by obscurity. In *New Security Paradigms Workshop (NSPW'11)*, Marin County, CA, USA, Sept. 2011.

[217] PCWorld. Alleged ransomware gang investigated by Moscow police. News article (Aug. 31, 2010). `http://www.pcworld.com/article/204577/article.html`.

[218] S. Pingbo. vtz: Trustzone and tee virtualization, 2017. Available at `https://www.trustkernel.com/en/products/hypervisor/vtz.html`.

[219] B. Pinkas and T. Sander. Securing passwords against dictionary attacks. In *ACM Computer and Communications Security (CCS'02)*, Washington, DC, USA, November 2002.

[220] J. E. Pixley, S. A. Ross, A. Raturi, and A. C. Downs. A survey of computer power modes usage in a university population, 2014. California Plug Load Research Center and University of California, Irvine. `http://www.energy.ca.gov/2014publications/CEC-500-2014-093/CEC-500-2014-093.pdf`.

[221] N. Provos. A virtual honeypot framework. In *USENIX Security Symposium*, San Diego, CA, USA, Aug. 2004.

[222] r0m30. msed - manage self encrypting drives. `https://github.com/NP-Hardass/msed/tree/master/msed`.

[223] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten. fTPM: a firmware-based TPM 2.0 implementation. Technical Report MSR-TR-2015-84, Microsoft Research, Nov. 2015.

[224] 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`.

[225] K. Rayner, S. J. White, R. L. Johnson, and S. P. Liversedge. Raeding wrods with jubmled lettres: There is a cost. *Psychological Science*, 17(3):192–193, Mar. 2006.

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

[227] 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.

[228] 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`.

[229] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[230] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM Computer and Communications Security (CCS'09)*, Chicago, IL, USA, November 2009.

[231] F. Rodriguez and R. Duda. System and method for providing secure authentication of devices awakened from powered sleep state, 2008. US Patent 20080222423.

[232] J. Rott. Intel AESNI sample library. Source code (May 11, 2011), available at: `https://software.intel.com/en-us/articles/download-the-intel-aesni-sample-library`.

[233] N. C. Rowe. Designing good deceptions in defense of information systems. In *the Annual Computer Security Applications Conferencen (ACSAC'04)*, Tucson, AZ, USA, December 2004.

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

[235] F. Saint-Jean. Java Implementation of a Single-Database Computationally Symmetric Private Information Retrieval (cSPIR) Protocol. Technical report, Yale University Department of Computer Science, 2005.

[236] Sakaki. Sakaki's EFI install guide/disabling the intel management engine. Wiki post (April 4, 2018). `https://wiki.gentoo.org/wiki/Sakaki%27s_EFI_Install_Guide/Disabling_the_Intel_Management_Engine`.

[237] 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/`.

[238] A. Saxena and M. PAUL. System and method for deletion of data in a remote computing platform.

[239] N. Scaife, H. Carter, P. Traynor, and K. R. B. Butler. Cryptolock (and drop it): Stopping ransomware attacks on user data. In *International Conference on Distributed Computing Systems (ICDCS'16)*, pages 303–312, Nara, Japan, June 2016.

[240] G. W. Scales, J. Elliott, and J. Norton. Systems and methods for the remote deletion of pre-flagged data.

[241] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[242] Seagate. DriveTrust technology: A technical overview. `http://www.seagate.com/docs/pdf/whitepaper/TP564_DriveTrust_Oct06.pdf`.

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

[244] SecurityWeek. LeChiffre ransomware hits Indian banks, pharma company. News article (Jan. 26, 2016). `http://www.securityweek.com/lechiffre-ransomware-hits-indian-banks-pharma-company`.

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

[246] R. Sevinsky. Funderbolt: Adventures in Thunderbolt DMA attacks. Black Hat USA, 2013.

[247] SFGate.com. Stockton mayor was briefly detained on return flight from China. News article (Oct. 2, 2015). `http://www.sfgate.com/bayarea/article/Stockton-mayor-was-briefly-detained-on-return-6546419.php`.

[248] J. Sharkey. Breaking hardware-enforced security with hypervisors. Black Hat USA, 2016.

[249] E. Shi, Y. Niu, M. Jakobsson, and R. Chow. Implicit authentication through learning user behavior. In *Information Security Conference (ISC'10)*, Boca Raton, FL, USA, Oct. 2010.

[250] T. Sim, S. Zhang, R. Janakiraman, and S. Kumar. Continuous verification using multimodal biometrics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(4):687–700, Apr. 2007.

[251] P. Simmons. Security through Amnesia: A software-based solution to the cold boot attack on disk encryption. In *ACSAC'11*, Orlando, FL, USA, 2011.

[252] 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`.

[253] SNPedia. Magnitude. https://www.snpedia.com/index.php/Magnitude, 2014.

[254] SNPedia. rs429358. https://www.snpedia.com/index.php/Rs429358, 2017.

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

[256] H. C. C. D. M. K. S. C. N. A. Spivey and R. Smith. *Essentials of Genetics*. NPG Education, 2009.

[257] W. Stallings. Format-preserving encryption: Overview and NIST specification. *Cryptologia*, 41(2):137–152, 2017.

[258] P. Stewin. *Detecting Peripheral-based Attacks on the Host Memory*. PhD thesis, Technischen Universität Berlin, July 2014.

[259] C. Stoll. *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage*. Doubleday, 1989.

[260] C. Strapparava and R. Mihalcea. Learning to identify emotions in text. In *ACM Symposium on Applied Computing (SAC 2008)*, Fortaleza, Brazil, March 2008.

[261] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: protecting data in compromised systems. In *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, pages 195–209, Los Alamitos, CA, USA, 2003.

[262] R. Sturmer. A lightweight implementation of the FAT32 filesystem for embedded systems. https://github.com/ryansturmer/thinfat32.

[263] T13 Technical Committee. *ATA Security feature Set Clarifications.* May 24, 2006.

[264] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. CleanOS: Limiting mobile data exposure with idle eviction. In *USENIX Operating Systems Design and Implementation (OSDI'12)*, Hollywood, CA, USA, Oct. 2012.

[265] C. Tarnovsky. Hacking the smartcard chip. Black Hat DC, 2010.

[266] 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`.

[267] B. Technology. ISEE trusted execution environment platform (secure os), 2016. Available at `http://www.beanpodtech.com/en/products/`.

[268] N. Technology. Nutlet technology passed tee authentication via gp, 2016. Available at `http://www.whty.com/news220.html`.

[269] The Atlantic. The computer virus that haunted early AIDS researchers. News article (May 10, 2016). `https://www.theatlantic.com/technology/archive/2016/05/the-computer-virus-that-haunted-early-aids-researchers/481965/`.

[270] The T13 Technical Committee. Information technology - ATA command set - 4 (ACS-4). `http://www.t13.org/Documents/UploadedDocuments/docs2016/di529r14-ATAATAPI_Command_Set_-_4.pdf`.

[271] TheGuardian.com. Revealed: how us and uk spy agencies defeat internet privacy and security, 2013. Available at `http://www.theguardian.com/world/2013/sep/05/nsa-gchq-encryption-codessecurity/print`.

[272] TheRegister.co.uk. Computing student jailed after failing to hand over crypto keys. News article (July 8, 2014).

[273] TheRegister.co.uk. Ex-Microsoft bug bounty dev forced to decrypt laptop for Paris airport official. News article (Jan. 6, 2015).

[274] TheRegister.co.uk. South Korean hosting co. pays $1m ransom to end eight-day outage. News article (June 20, 2017). `https://www.theregister.co.uk/2017/06/20/south_korean_webhost_nayana_pays_ransom/`.

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

[276] Trusted Computing Group. TCG storage architecture core specification. `https://trustedcomputinggroup.org/wp-content/uploads/TCG_Storage_Architecture_Core_Spec_v2.01_r1.00.pdf`.

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

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

[279] E. TS. At command set for user equipment, 2011. Available at `http://www.etsi.org/deliver/etsi_ts/127000_127099/127007/10.03.00_60/ts_127007v100300p.pdf`.

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

[281] USB.org. Enhanced host controller interface specification for universal serial bus. `http://www.usb.org/developers/resources/`.

[282] Usb.org. Universal serial bus (USB), device class definition for human interface devices (HID). Firmware Specification (June 27, 2001). `http://www.usb.org/developers/hidpage/HID1_11.pdf`.

[283] A. S. Uz. The effectiveness of remote wipe as a valid defense for enterprises implementing a BYOD policy. Master's thesis, University of Ottawa, 2014.

[284] J. van Hoboken, A. Arnbak, and N. van Eijk. Cloud computing in higher education and research institutions and the usa patriot act, 2012. Available at SSRN: `https://ssrn.com/abstract=2181534`.

[285] G. Vasiliadis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. PixelVault: Using GPUs for securing cryptographic operations. In *ACM CCS'14*, Scottsdale, AZ, USA, Nov. 2014.

[286] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an eXtensible and modular hypervisor framework. In *IEEE Symposium on Security and Privacy*, Berkeley, CA, USA, 2013.

[287] T. Vidas. AfterLife: USB based memory acquisition tool targeting "warm boot" machines with 4GB of RAM or less. `http://sourceforge.net/projects/aftrlife/`.

[288] T. Vidas. Volatile memory acquisition via warm boot memory survivability. In *Hawaii International Conference on System Sciences (HICSS'10)*, Honolulu, HI, USA, Jan. 2010.

[289] Visa. Verified by Visa FAQ & credit card security. Online FAQ. `http://usa.visa.com/personal/security/visa_security_program/vbv/verified_by_visa_faq.html#anchor_15`.

[290] T. Walsh, M. K. Lee, S. Casadei, A. M. Thornton, S. M. Stray, C. Pennil, A. S. Nord, J. B. Mandell, E. M. Swisher, and M.-C. King. Detection of inherited mutations for breast and ovarian cancer using genomic capture and massively parallel sequencing. *Proceedings of the National Academy of Sciences*, 107(28):12629–12633, 2010.

[291] X. S. Wang, Y. Huang, Y. Zhao, H. Tang, X. Wang, and D. Bu. Efficient genome-wide, privacy-preserving similar patient query based on private edit distance. In *CCS*, pages 492–503. ACM, 2015.

[292] J. Wiklander. Secure storage in OP-TEE. Available at `https://github.com/OP-TEE/optee_os/blob/master/documentation/secure_storage.md`.

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

[294] 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.

[295] Wired. How an accidental 'kill switch' slowed Friday's massive ransomware attack. News article (May 13, 2017). `https://www.wired.com/2017/05/accidental-kill-switch-slowed-fridays-massive-ransomware-attack/`.

[296] R. Wojtczuk and C. Kallenberg. Attacking UEFI boot script, 2014. `http://events.ccc.de/congress/2014/Fahrplan/system/attachments/2566/original/venamis_whitepaper.pdf`.

[297] 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.

[298] R. Wojtczuk, J. Rutkowska, and A. Tereshkin. Another way to circumvent Intel Trusted Execution Technology: Tricking SENTER into misconfiguring VT-d via SINIT bug exploitation. Technical article (Dec., 2009). http://theinvisiblethings.blogspot.com/2009/12/another-txt-attack.html.

[299] 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.

[300] A. Woodward. Bitlocker - the end of digital forensics? In *Australian Digital Forensics Conference*, page 38, Perth, Western Australia, 2006.

[301] Y. Xia, Y. Liu, C. Tan, M. Ma, H. Guan, B. Zang, and H. Chen. *TinMan*: Eliminating confidential mobile data exposure with security oriented offloading. In *EuroSys'15*, Bordeaux, France, Apr. 2015.

[302] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy*, 2015.

[303] A. C. Yao. Protocols for secure computations. In *FOCS*, 1982.

[304] Z. E. Ye, S. Smith, and D. Anthony. Trusted paths for browsers. *ACM Transactions on Information and System Security (TISSEC)*, 8(2):153–186, 2005.

[305] J. Youll. Fraud vulnerabilities in SiteKey security at Bank of America. Technical article (July 18, 2006). `http://www.cr-labs.com/publications/SiteKey-20060718.pdf`.

[306] A. Young and M. Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 1996.

[307] A. Young and M. Yung. Cryptovirology: The birth, neglect, and explosion of ransomware. *Communications of the ACM*, 60(7):24–26, July 2017.

[308] L. Ysboodt and M. D. Nil. Embedded filesystems library. `https://sourceforge.net/projects/efsl/?source=typ_redirect`.

[309] X. Yu, Z. Wang, K. Sun, W. T. Zhu, N. Gao, and J. Jing. Remotely wiping sensitive data on stolen smartphones. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS'14, pages 537–542, Kyoto, Japan, 2014.

[310] J. Yuill, D. Denning, and F. Feer. Using deception to hide things from hackers: Processes, principles, and techniques. *J. Information Warfare*, 5(3):26–40, 2006.

[311] L. Zhao and M. Mannan. Gracewipe: Secure and verifiable deletion under coercion. In *NDSS'15*, San Diego, CA, USA, Feb. 2015.

[312] L. Zhao and M. Mannan. Gracewipe: Secure and verifiable deletion under coercion. In *NDSS'15*, San Diego, CA, USA, Feb. 2015.

[313] L. Zhao and M. Mannan. Hypnoguard: Protecting secrets across sleep-wake cycles. Technical Report 981477, Concordia University, Aug. 2016. `http://spectrum.library.concordia.ca/981477/`.

Appendix A

Glossary and Additional Information

In this appendix, we provide the definition or explanation of certain technical terms that are used across chapters to facilitate understanding of the implementation of individual prototype systems.

Trusted Execution Environment (TEE). Modern CPUs usually support a special secure mode of execution, which ensures that only pre-configured unaltered code can be executed, with integrity, secrecy and attestability; and provides a form of isolation from both other software/firmware and physical tampering. TEE can be exclusive, preempting and suspending other code (as in Intel TXT), or concurrent, co-existing with other processes (as in Intel SGX and ARM TrustZone).

Technically, TEEs cannot function alone. For the purpose of storing measurements (to be matched with that of the code being loaded) and secure storage of execution secrets, a secure element is used in conjunction. It can be part of the processor die, an integrated chip, or a discrete module.

Intel TXT. Trust Execution Technology is Intel's first "late launch" technique, aiming at establishing trusted execution any time after system recycle, without relying on what has been already loaded (e.g., BIOS). It is exclusive, removing software side-channel attack vectors and with the help of VT-d [10], largely defending against violations from the I/O space. The TXT session must be first bootstrapped by an Intel authorized code module (`ACM` or `SINIT`), which performs the actual loading of a user-deployed program. TXT works with `TPM` (Trust Platform Module) as the secure element.

The volatile secure storage on the TPM chip includes `PCRs` (Platform Configuration Registers) where the run-time measurement can be stored. They can not be directly accessed but only extended (i.e., replaced with the cryptographic hash value of its original value concatenated with the new measurement). The non-volatile secure storage on the TPM chip is called `NVRAM`, which is accessible in the form of *index* (a numeric identifier). NVRAM indices can be allocated and deallocated and there can be multiple of them. They can be configured in different protection modes, e.g., by a password (called AuthData), or only when specific PCR values are present, or a combination thereof.

Sealing. Short for cryptographic sealing, it is a special mode of encryption, provided by TEEs, where the key is derived (in many ways) largely from the machine state, in the form of *measurement*. Measurement is the chaining of the loaded programs in sequence, e.g., concatenation of hashed values (for the TPM, residing in PCRs). Any single bit of change in loaded programs will cause a mismatch of measurement, making the derived key different, and thus render the decryption (unsealing) to fail. In this way, platform binding is achieved.

Tboot. Tboot [134] 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 the ACM, 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 (see below).

Multiboot. The multiboot specification [91] is an open standard for multi-stage/coexistent booting of different operating systems or virtual machine monitors (VMMs); it has been implemented in several tools, e.g., GRUB,[1] kexec tools,[2] and tboot [134]. 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. Chainloading [3] 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). Windows does not support the multiboot specification, so it is chainloaded by Gracewipe. We use GRUB as the bootloader for Gracewipe, as GRUB supports both multiboot and chainloading.

Flicker [183]. Before the advent of Flicker, Intel TXT was mostly applied with the pilot project tboot, which deals with boot-time trusted execution. The ability to switch between the regular OS environment and the trusted execution had always

[1] `http://www.gnu.org/software/grub/`

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

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

been desired. Flicker enables such transitions, e.g., interrupting and saving states for the OS, initiating the TXT session, performing trusted operations and resuming the OS. The trusted operations are encapsulated in what is called a PAL (piece of application logic) and thus OS-agnostic. It satisfies what is needed in Inuksuk.

TrueCrypt. The TrueCrypt on-the-fly full-disk encryption (FDE) utility is possibly the most popular choice in its kind at the time. 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 [12] 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 [278] 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. This is the interface we choose to use for Gracewipe. 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.

Also, SEDs come with certain additional vendor-specific interfaces for richer functionalities (e.g., TCG Opal and Seagate DriveTrust). With such interfaces, most SEDs offer fine-grained protection, such as dividing media space into ranges and splitting read/write accesses. This is the interface we choose to use for Inuksuk. The several design flaws or firmware bugs identified by researchers mostly rely on physical access, i.e., desoldering a microchip, manipulating the connector or evil maid attacks. Inuksuk is not affected by such attacks, as only software adversaries are considered. Also what we need is merely the write-protection enforcement, which is a lower requirement than ensuring data secrecy.