Hypnoguard: Protecting Secrets across Sleep-wake Cycles*

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

Abstract

Attackers can get physical control of a computer in sleep (S3/suspend-to-RAM), if it is lost, stolen, or the owner is being coerced. High-value memory-resident secrets, including disk encryption keys, and private signature/encryption keys for PGP, may be extracted (e.g., via cold-boot or DMA attacks), by physically accessing such a computer. Our goal is to alleviate threats of extracting secrets from a computer in sleep, without relying on an Internet-facing service.

We propose *Hypnoguard* to protect all memory-resident OS/user data across S3 suspensions, by first performing an in-place full memory encryption before entering sleep, and then restoring the plaintext content at wakeup-time through an environment-bound, password-based authentication process. The memory encryption key is effectively "sealed" in a Trusted Platform Module (TPM) chip with the measurement of the execution environment supported by CPU's trusted execution mode (e.g., Intel TXT, AMD-V/SVM). Password guessing within Hypnoguard may cause the memory content to be permanently inaccessible, while guessing without Hypnoguard is equivalent to brute-forcing a high-entropy key (due to TPM protection). We achieved full memory encryption/decryption in less than a second on a mainstream computer (Intel i7-4771 CPU with 8GB RAM, taking advantage of multi-core processing and AES-NI), an apparently acceptable delay for sleep-wake transitions. To the best of our knowledge, Hypnoguard provides the first wakeup-time secure environment for authentication and key unlocking, without requiring per-application changes.

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., [50]. 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 [38, 54, 6, 58] by exploiting vulnerable device drivers; common mitigations include: bug fixes, IOMMU (Intel VT-d/AMD Vi), and disabling (FireWire) DMA when screen is locked (e.g., Mac OS X 10.7.2 and later, Windows 8.1 [38]). A sophisticated attacker can also resort to cold-boot attacks by exploiting DRAM memory remanence effect [26, 22]. Simpler techniques also exist for memory extraction (e.g., [16]); some tools (e.g., [14]) may bypass 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., [48, 46, 57, 23, 64, 24]; see Section 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. [5]). 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 [37], and a comprehensive survey [28]). 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 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

*This is the tech-report version (August 11, 2016) of a CCS 2016 paper [72]; new additions: the custom GCM implementation and other minor details in the Appendix.

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 [31]) and AMD Virtualization (AMD-V/SVM [2]). 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. [47]).

Several boot-time solutions (e.g., [32, 65, 71]) 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. [13, 63]).

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 defense against memory attacks when the computer is in the wrong hands, and severely restrict guessing of weak authentication secrets (cf. [71]). 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.

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. [49], Intel [31], and AMD [2].

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 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 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.

2.2 Goals

We primarily consider attacks targeting extraction of secrets through physical access to 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 [71]): (*AG1*) when deletion is successful, there should be a cryptographic evidence that convinces the adversary that the RAM secrets are permanently inaccessible.

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., [26, 22]), 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., [60, 69, 55]); see also Item (f) in Section 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., [20] for wakeup-time. However, no known effective defense exists for more advanced evil-maid attacks, including hardware modifications as in NSA's ANT catalog [21]. 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. [45, 12]). A secure boot-time solution should be used to enforce strong authentication (cf. [71]). 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 [1]) 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 [68], 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}. Coercion has been considered recently during boot-time [71],

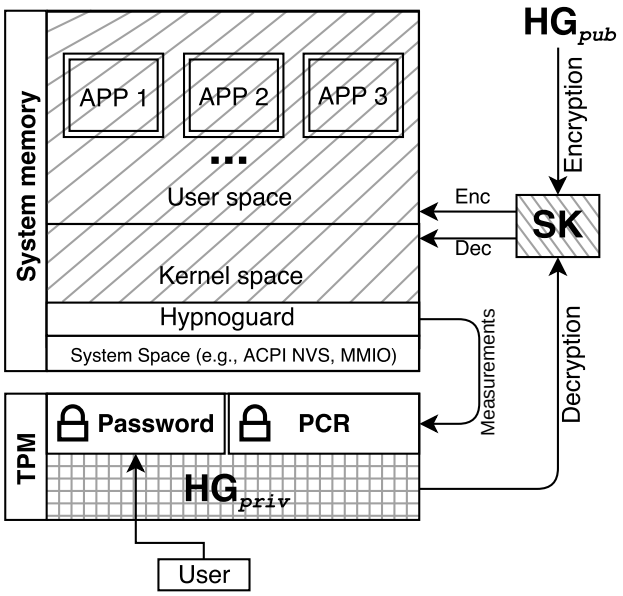

Figure 1: Memory layout and key usage of Hypnoguard. Shaded areas represent encrypted/protected data; different patterns refer to using different schemes/key types.

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.

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.

3 Design

In this section, we detail the architecture of Hypnoguard, and demonstrate how it achieves the design goals stated in Section 2.2. Technical considerations not specific to our current implementation are also discussed.

Overview. Figure 1 shows the memory layout and key usage of Hypnoguard across sleep-wake cycles; the transition and execution flows are described in Section 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.

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 [39]), 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 [41]). 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 [61] (and optionally, mix them with the output from the RDRAND instruction in modern CPUs).

Necessity of HG_{priv} and HG_{pub}. Although we use 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).

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. [11]). 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 [71]). 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 4.1.

3.3 How goals are achieved

Hypnoguard's goals are defined in Section 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 per-

formed 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. [31, 61]); 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 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.

4.1 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.

Execution steps. Figure 2 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 4.3). 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 run-

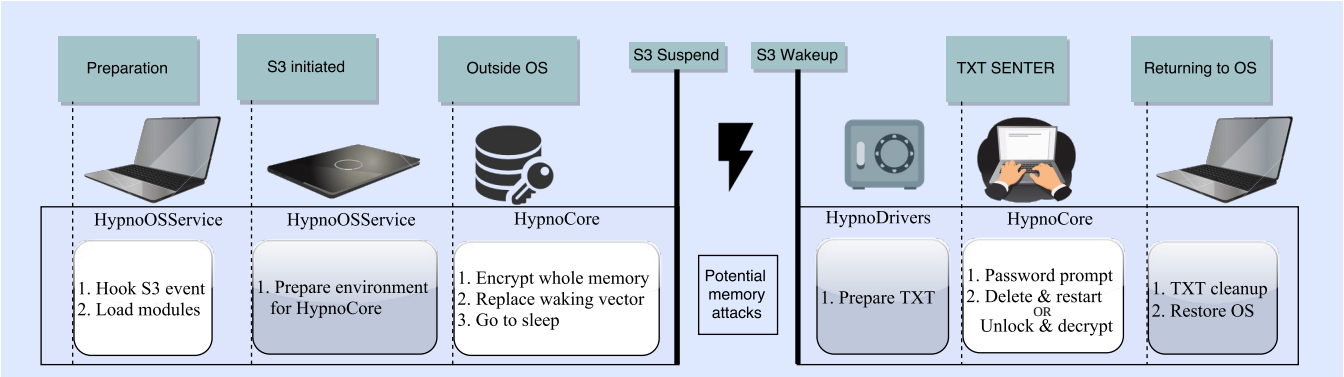

Figure 2: Simplified execution steps of Hypnoguard

ning 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.

4.2 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 [32]. 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.

4.3 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, so that Linux will not touch it afterwards. This is done by appending a 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 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 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 dur-

ing 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 [47] 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" [62].
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., [47, 15].

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 [41] 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.

4.6 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. [34]), 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 (see Appendix B for an example of our e820 table).

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.

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 [8]). 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 [53], with minor but non-trivial modifications (see Appendix C).

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 [33]. 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 Appendix A.

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 [9] 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 [29], 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, see Appendix A). The encryption function in assembly provided by Intel already works satisfactorily, which we do not change further. The performance numbers are listed in Table 1.

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 1: 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 Appendix A); — means not evaluated.

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 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 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 multicore 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.

7 Security analysis

Below, we discuss potential attacks against Hypnoguard; see also Sections 2.3 and 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 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 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., [10, 67, 66]). Several such tools exist, some of which are applicable to locked computers, see e.g., [16]. 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 [71] 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 [34, 68] (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., [71]), 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., [7]) are generally inapplicable against Hypnoguard, as no secrets are processed before the correct password is entered. For such an example attack on Xbox, see [30], 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., [60], see also [27] 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 [51] for discrete vs. firmware TPMs). Hypnoguard should be easily adapted to TPM v2.0.

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., [56, 17]) 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 [26]. There have been proposals based on relocation of secret keys from RAM to other "safer" places, such as SSE registers (AESSE [44]), debug registers (TRESOR [46]), MSR registers (Amnesia [57]), AVX registers (PRIME [18]), CPU cache and debug registers (Copker [23]), GPU registers (PixelVault [64]), and debug registers and Intel TSX (Mimosa [24]).

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 reauthentication at wakeup-time (e.g., [18, 23]). Several of them (re)derive their master secret, or its equivalent, from the user password, e.g., [44, 46]; 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 [37] for an early proposal in this domain, and Henson and Taylor [28] 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 [35] 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., [40]). 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 [43]). 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 [42]. In this setting, unattended computers would always resume from a powered-off state (cf. YoNTMA [35]), 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., [4]).

Recreating trust after S3 sleep. To re-establish a secure state when the system wakes up from S3, Kumar et al. [36] 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 [52] 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. [10] 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 [59] 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 [70] also relies on a trusted server, but does not decrypt confidential data in the device memory to avoid physical attacks. Keypad [19], 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 [71], 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 [3]) 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).

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 [71]) 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.

Acknowledgements

This paper was significantly improved by the insightful comments and suggestions from the anonymous reviewers of CCS 2016, USENIX Security 2016 and EuroSys 2016, as well as Jonathan McCune. We also appreciate the help we received from the members of Concordia's Madiba Security Research Group. The second author is supported in part by an NSERC Discovery Grant.

References

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

[2] 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`.

[3] 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.

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

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

[6] 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`.

[7] 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`.

[8] 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.

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

[10] 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.

[11] J. Clark and U. Hengartner. Panic passwords: Authenticating under duress. In *USENIX HotSec'08*, San Jose, CA, USA, July 2008.

[12] 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.

[13] 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.

[14] Elcomsoft.com. Elcomsoft forensic disk decryptor: Forensic access to encrypted BitLocker, PGP and TrueCrypt disks and containers. https://www.elcomsoft.com/efdd.html.

[15] 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.

[16] Forensicswiki.org. Tools:memory imaging. http://www.forensicswiki.org/wiki/Tools:Memory_Imaging.

[17] 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.

[18] B. Garmany and T. Müller. PRIME: Private RSA infrastructure for memory-less encryption. In *ACSAC'13*, New Orleans, LA, USA, 2013.

[19] 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.

[20] J. Götzfried and T. Müller. Mutual authentication and trust bootstrapping towards secure disk encryption. *ACM TISSEC*, 17(2):6:1–6:23, Nov. 2014.

[21] Gov1.info. NSA ANT product catalog. https://nsa.gov1.info/dni/nsa-ant-catalog/.

[22] 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.

[23] 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.

[24] 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.

[25] S. Gueron and M. E. Kounavis. Intel®carry-less multiplication instruction and its usage for computing the GCM mode. Intel whitepaper (Apr. 2014).

[26] 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 Symp.*, Boston, MA, USA, 2008.

[27] 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.

[28] M. Henson and S. Taylor. Memory encryption: A survey of existing techniques. *ACM Computing Surveys (CSUR)*, 46(4):53:1–53:26, Mar. 2014.

[29] A. Horvath and J. M. Slocum. Memory bandwidth benchmark. Open source project. https://github.com/raas/mbw.

[30] A. Huang. Keeping secrets in hardware: The Microsoft Xbox™ case study. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES'02)*, San Francisco, CA, USA, Aug. 2002.

[31] 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.

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

[33] Intel. The MultiProcessor specification (MP spec), May 1997. http://www.intel.com/design/archives/processors/pro/docs/242016.htm.

[34] 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.

[35] iSECPartners. YoNTMA (you'll never take me alive!). https://github.com/iSECPartners/yontma.

[36] 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.

[37] 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.

[38] C. Maartmann-Moe. Inception. PCI-based DMA attack tool. `https://github.com/carmaa/inception`.

[39] 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.

[40] Maximintegrated.com. Switching between battery and external power sources, 2002. `http://pdfserv.maximintegrated.com/en/an/AN1136.pdf`.

[41] 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.

[42] Microsoft.com. BitLocker frequently asked questions (FAQ). Online article (June 10, 2014). `https://technet.microsoft.com/en-ca/library/hh831507.aspx`.

[43] 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`.

[44] 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.

[45] T. Müller and F. C. Freiling. A systematic assessment of the security of full disk encryption. *IEEE TDSC*, 12(5):491–503, September/October 2015.

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

[47] 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.

[48] E. T. Pancoast, J. N. Curnew, and S. M. Sawyer. Tamper-protected DRAM memory module, December 2012. US Patent 8,331,189.

[49] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011.

[50] 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`.

[51] 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. Microsoft techreport, MSR-TR-2015-84 (Nov. 5, 2015).

[52] F. Rodriguez and R. Duda. System and method for providing secure authentication of devices awakened from powered sleep state, 2008. US Patent 20080222423.

[53] 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`.

[54] R. Sevinsky. Funderbolt: Adventures in Thunderbolt DMA attacks. Black Hat USA, 2013.

[55] J. Sharkey. Breaking hardware-enforced security with hypervisors. Black Hat USA, 2016.

[56] 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.

[57] P. Simmons. Security through Amnesia: A software-based solution to the cold boot attack on disk encryption. In *ACSAC'11*, Orlando, FL, USA, 2011.

[58] P. Stewin. *Detecting Peripheral-based Attacks on the Host Memory*. PhD thesis, Technischen Universität Berlin, July 2014.

[59] 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.

[60] C. Tarnovsky. Hacking the smartcard chip. Black Hat DC, 2010.

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

[62] 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`.

[63] 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.

[64] 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.

[65] 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.

[66] T. Vidas. AfterLife: USB based memory acquisition tool targeting "warm boot" machines with 4GB of RAM or less. `http://sourceforge.net/projects/aftrlife/`.

[67] T. Vidas. Volatile memory acquisition via warm boot memory survivability. In *Hawaii International Conference on System Sciences (HICSS'10)*, Honolulu, HI, USA, Jan. 2010.

[68] R. Wojtczuk and C. Kallenberg. Attacking UEFI boot script, 2014. `http://events.ccc.de/congress/2014/Fahrplan/system/attachments/2566/original/venamis_whitepaper.pdf`.

[69] 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`.

[70] 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.

[71] L. Zhao and M. Mannan. Gracewipe: Secure and verifiable deletion under coercion. In *NDSS'15*, San Diego, CA, USA, Feb. 2015.

[72] L. Zhao and M. Mannan. Hypnoguard: Protecting secrets across sleep-wake cycles. In *ACM Computer and Communications Security (CCS'16)*, Vienna, Austria, Oct. 2016.

A Adaptation for AES-GCM

A.1 Implementing regular GCM

GCM mode is unavailable in the current Intel AES-NI library. So we make use of a set of sample code published by Intel [25] as a starting point. Of interest to us, the sample code comes with, among other functions, encryption function (single block at a time) in C, encryption function (4 blocks at a time) in both C and assembly, key expansion function in C, and decryption function (single block at a time) in C. While adapting AES-GCM for Hypnoguard, we encountered several challenges, as discussed below. We did not evaluate the encryption function as it is already in assembly with 4-block processing (yielding over 7GB/s).

Stack alignment. All Intel SSE instructions (involving registers `XMM0` through `XMM7`, and in x64 `XMM8` through `XMM15`) mandate the memory operand to be 16-byte aligned; otherwise, the general-protection exception (#GP) will crash the system. For GCM implementations (exemplified by Intel's sample code), such instructions are used intensively. In assembly, this can be adjusted manually, while in C it must be enforced at the compiler level, e.g., by specifying `__attribute__((aligned (16)))` when defining such variables. However, for Hypnoguard we perform "hybrid compilation", where the 32-bit components (for all core functions like TXT processing) and the 64-bit component (for AES encryption) are compiled separately, and the 64-bit code is only invoked as binary located at a fixed address with an arbitrary stack position. Therefore, the compiler for the 64-bit code no longer has the global control over stack usage. This causes indeterministic behavior, such as, random system crashes when the incoming stack is aligned differently from what the compiler assumed.

A way around it is to force the compiler to insert a prologue that always realigns the local variables on the stack by specifying `-mstackrealign` (forcing to realign when necessary) and `-mincoming-stack-boundary=3` (assuming the worst case: 8-byte aligned). However, this is disallowed in a 64-bit system with current GCC versions (producing error message "*-mincoming-stack-boundary=3 is not between 4 and 12*"). We follow a non-mainline patch [1] (backported to gcc-5), and manually applied the changes to allow the value 3. After compilation, a prologue (a few lines of assembly code) is inserted to the beginning of each C function. This prologue always contains "`and $0xfffffffffffffff0,%rsp`" that removes the last significant nibble so that it is guaranteed to be 16-byte aligned. This way, Hypnoguard can run with GCM mode without crashes.

Parallelized decryption. Intel's sample code mostly focus on encryption performance, and the same applies to its evaluation, i.e., no mention of decryption. The decryption function is single-threaded and only available in C. Therefore, even if all the four cores are used, decryption speed was always under 3.3GB/s, which takes nearly 2.5 seconds to resume for an 8GB-memory computer. The first step we take is adapting it to process 4 blocks at a time, following the already parallelized encryption function.

We observe that even with all physical cores enabled, processing 4 blocks at a time only improves the performance a little (from 3.22GB/s to 3.3GB/s). We realize that the decryption function being in C, and that there is a prologue at the entry of each function invocation are possibly causing significant latency. We proceed to adapt the encryption function in assembly for decryption according to the construction of the existing decryption function in C. During this process, we have encountered several challenges, mainly related to the lack of deep understanding of the undocumented code, e.g., implicit and condensed use of the limited number of registers.

For now, we are unable to tell if the prologues alone are the root cause for the performance degradation, rather than the code being in C, as the program will crash without them. The new decryption function in assembly (processing 4 blocks at a time) achieves a performance of

[1] `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=66697`

5GB/s. This is much better than our initial attempts; however, considering wakeup-time latency (decryption) is more critical to user experience, 1.6 seconds for an 8GB-memory computer may not be ideal.

A.2 GCM with deferred tag verification

With the current GCM construction, we have already customized the implementation to a large extent, i.e., code written in assembly and processing 4 blocks at a time. However, encryption is still much faster than decryption (1.55 times) even when all other factors are the same. Note that in GCM, decryption is composed of a tag verification phase and the actual decryption. Our analysis shows that the tag verification accounts for a significant portion of the latency (a complete separate loop through all the data blocks). But this sequential processing is apparently necessary: the tag verification should be done before the (more) processing-intensive decryption operation is performed.

We propose a significant change in GCM implementation for boosting GCM decryption performance that fits certain circumstances, e.g., in our case where it does not matter if the plaintext content is already decrypted in RAM when an anomaly is detected (Hypnoguard can just halt/reboot the machine when tag verification fails). To this end, we try to remove the upfront tag verification, and integrate it into the existing decryption loop. Recall that during encryption the tag is generated alongside, and the logic is partially reusable and helpful for adapting the decryption function. The new construction only contains one loop through the ciphertext blocks, and GHASH is calculated for each iteration and the tag value is updated. We eventually managed to make run this construction, and have verified that at the end of the decryption, Hypnoguard is able to notify the user if authenticity of the RAM data is compromised; the user can then reboot or shut down the machine. Our GCM implementation with deferred tag verification achieves 6.8GB/s, taking only 1.18 seconds to wake up.

B An example e820 memory map

Our e820 table is shown below as an example; note that the reserved spaces are scattered across the RAM, and remain unused by the OS or user applications. We therefore exclude encrypting such regions.

```
[mem 0x0000000000000000-0x000000000009ebff] usable
[mem 0x000000000009ec00-0x000000000009ffff] reserved
[mem 0x00000000000e0000-0x00000000000fffff] reserved
[mem 0x0000000000100000-0x00000000007fffff] usable
[mem 0x0000000000800000-0x0000000000afffff] reserved
[mem 0x0000000000b00000-0x00000000b4fabfff] usable
[mem 0x00000000b4fac000-0x00000000b4fb2fff] ACPI NVS
[mem 0x00000000b4fb3000-0x00000000b5403fff] usable
[mem 0x00000000b5404000-0x00000000b586afff] reserved
[mem 0x00000000b586b000-0x00000000ca22dfff] usable
[mem 0x00000000ca22e000-0x00000000ca436fff] reserved
[mem 0x00000000ca437000-0x00000000ca44ffff] ACPI data
[mem 0x00000000ca450000-0x00000000ca9a4fff] ACPI NVS
[mem 0x00000000ca9a5000-0x00000000cbbfefff] reserved
[mem 0x00000000cbbff000-0x00000000cbbfffff] usable
[mem 0x00000000ce000000-0x00000000de1fffff] reserved
[mem 0x00000000f8000000-0x00000000fbffffff] reserved
[mem 0x00000000fec00000-0x00000000fec00fff] reserved
[mem 0x00000000fed00000-0x00000000fed03fff] reserved
[mem 0x00000000fed1c000-0x00000000fed1ffff] reserved
[mem 0x00000000fee00000-0x00000000fee00fff] reserved
[mem 0x00000000ff000000-0x00000000ffffffff] reserved
[mem 0x0000000100000000-0x000000021fdfffff] usable
```

C A memory alignment bug in Intel AES-NI library

During the process of the implementation, we identified a potential issue in the assembly module of Intel's 64-bit AES-NI library (only CTR mode). In *iEnc128_CTR*, there is one instruction that requires the first argument to be 16-byte aligned: `movdqa [rsp+16*16], xmm6`.

However, we have no control over where the stack pointer (*rsp*) is. Especially, even if we always assume rsp is 16-byte aligned on its entry, the instruction before it breaks this situation (`sub rsp,16*16+8+16`). In consequence, the system crashes with a general protection error (depending on the calling context). We ended up having to patch it as follows:

```
mov rax, rsp ;stack pointer into rax for processing
and eax, 15  ;take the least significant nibble
sub rsp, rax ;subtract the nibble from the stack pointer
             ;so that it is guaranteed to be 16-byte aligned
push rax     ;save what we have done for later reversion
...          ;the actual CTR function in AES-NI library
pop rax      ;load what was subtracted
add rsp, rax ;reverse what we have done
```