

DOI: 10.37943/ZHXT2203

Zhaksylyk Kozhakhmet

Master's Degree, Junior Researcher, CyberTech Research Center
zh.kozhakhmet@astanait.edu.kz; orcid.org/0009-0002-5449-3317
Astana IT University, Kazakhstan

Ali Myrzatay

PhD, Senior Researcher, CyberTech Research Center
mirzataitegiali@gmail.com; orcid.org/0000-0002-5339-2437
Astana IT University, Kazakhstan

Alissa Ryzhova

Master's Student, Department of Environmental Sciences, Informatics and Statistics
alissaryzh@gmail.com; orcid.org/0009-0006-3781-1773
Ca' Foscari University of Venice, Italy

Gul Jussupova

PhD, Department of Digitalization and Public Services
g.jussupova@sci.gov.kz; orcid.org/0000-0002-3456-810X
Ministry of Science and Higher Education of the Republic of Kazakhstan,
Kazakhstan

A HARDWARE-BOUND SPLIT-KEY FRAMEWORK FOR SOFTWARE PROTECTION AGAINST REVERSE ENGINEERING

Abstract: Software piracy and reverse engineering cause significant damage to intellectual property: the commercial value of unlicensed software installed worldwide has been estimated at over USD 46 billion. Traditional protection methods rely on static encryption keys or on obfuscation, and they are not resistant to modern analysis techniques such as differential computation analysis (DCA). This study addresses the single point of failure in software protection systems by developing a new key management architecture. The paper introduces a distributed split-key software-hardware framework. A deterministic decryption key is divided into three parts: a local key derived from the unique properties of the machine's hardware (hardware fingerprint), a server key issued after authenticated online activation, and a key sealed in a Trusted Platform Module (TPM) and bound to the platform's Platform Configuration Registers (PCRs). The final key is assembled exclusively in RAM during program execution through an XOR operation and is erased immediately after use. A C#/ .NET 8 reference implementation was evaluated on a single Windows platform with a discrete TPM 2.0 module. Changing any of the five hardware identifiers or the PCR state prevented decryption, the assembled key remained in memory for a median of 15.6 μ s, key assembly took 0.3 μ s, and the one-time TPM sealing took about 67 ms. The runtime overhead stayed below 5% for typical protected functions. The analysis also identifies the remaining limitations of the design: code harvesting through legitimate decryption, binary patching that does not affect PCR 0 and PCR 7, clock rollback during the offline grace period, and loss of access after legitimate firmware or Secure Boot updates. The framework binds execution to authorized hardware and raises the cost of reverse engineering.

Keywords: cryptography; data security; key generation; trusted platform module; platform's configuration registers;

Introduction (Literary review)

Even though the techniques of preventing software from being pirated are constantly changing, the question of whether or not it is possible to prevent illegal access to software through reverse engineering is not yet resolved. The most recent BSA Global Software Survey estimated that 37% of the software installed on personal computers worldwide was unlicensed, with a

commercial value of USD 46.3 billion [1], and firm-level evidence shows that piracy measurably reduces the innovation output of software companies [2]. Apart from the obvious financial loss to the enterprise, reverse engineering can be used to gain access to algorithms, making companies less competitive, and to bypass any licensing scheme to make unauthorised copies of software. Traditional software protection methods such as checking serial numbers, hardware dongles and software obfuscation have failed to protect software from modern attackers who combine debuggers, disassemblers and decompilers with automated deobfuscation based on machine learning and symbolic execution [3].

A major limitation of conventional protection solutions is that they rely on a single point of secrets. The attacker's goal is simply to discover and retrieve this single value, whether this is an embedded encryption key, license file, or a serial number. After obtaining the protection, it can never be restored and the key can be shared for an infinite number of illegal copies. Even the more complex approach of white-box cryptography, which seeks to hide the key behind a lookup table, is vulnerable to differential computation attacks and algebraic attacks which can be reasonably performed to recover the underlying key [4].

The paper proposes a software protection framework that fundamentally alters the security model by guaranteeing that there is never a complete decryption key as an extractable value. It presents the approach where the key is divided into three separate elements: one element is derived from the machine characteristics and hardcoded into the hardware, another element is issued by the server and is obtained through authenticated online activation, and the last one is sealed in the TPM (Trusted Platform Modules) and can be accessed only if the proper credentials are present on the authorized machine. The full key is only created at run-time by XORing it together, and is immediately used for decryption without being stored in memory.

The main contributions of this paper are:

- A novel split-key architecture to remove the single point of failure in software protection;
- A combination of hardware fingerprinting, server authentication and TPM sealing in one package;
- A thorough security analysis that shows resistance to known attack vectors;
- Guidance on how to implement these in the Windows platform.

Main Text:

Protection of software has been a topic of ongoing research for many years and there are many different methods for software protection, such as legal, code obfuscation, encryption, and hardware-based solutions. Commercial products like VMProtect, Themida and Denuvo use virtualization-based obfuscation where code segments are turned into virtual machine code of custom virtual machines, making analysis much harder. But, these solutions have been repeatedly bypassed using automated devirtualization tools and time spent on manual analysis, showing that obfuscation is not a long-term solution. For instance, it is believed that Denuvo uses periodic activation as one of its many security measures, so buyers have to log in every few days to authenticate their games. But the activation servers of Denuvo negatively impact the legitimate user experience for the legitimate users. This is what happened when Denuvo's servers prevented legitimate buyers of Batman Arkham Knight from accessing the authentication server to activate their legitimate copies. Users, who were using cracked versions of the game, were unaffected by the outages, as the game had been cracked by a threat actor who managed to emulate these activations offline using their cracks. It turns out that if Denuvo's servers shut down permanently, users who purchased the game will lose the ability to play it [5]. Empirical evidence confirms that such protection is short-lived: Denuvo-protected games that are cracked soon after launch lose about 20% of their revenue [6].

In the wake of Microsoft's mandate for TPM 2.0 to certify Windows 11, hardware-based protection has been adopted through the use of Trusted Platform Modules (TPM). TPMs can store keys securely, measure the integrity of the platform using Platform Configuration Registers (PCRs) and seal operations that tie secrets to particular states in the platform [7-8]. TPM sealing

is used in BitLocker and other full-disk encryption solutions to safeguard volume encryption keys. But TPM-only schemes are susceptible to hardware attacks such as bus sniffing; it has been shown that BitLocker keys can be extracted by observing the LPC or SPI bus between a discrete TPM and the CPU at the moment of unsealing [9].

Alternatively, software can be executed with greater security assurances, such as confidentiality and integrity, using a Trusted Execution Environment (TEE) without a trusted operating system. Remote Attestation is a process where specific software can be trusted and its authenticity and integrity is verified. For instance, end users can work around digital rights management systems with video streaming software. Moreover, virtual machines are obviously susceptible to malicious actions of their untrusted cloud hosts. The presence of Intel SGX, AMD SEV, RISC-V and Arm TrustZone-A/M TEE in common processors is a great help in the construction of secure applications. With the remote attestation, one can be confident that remote communication is taking place with a specific, trusted program. TEEs can provide protection and support for the attestation process, and isolate the attested software from the external system while also protecting it from powerful attacks. TEEs are used for attestation in a number of ways, however. Due to weak integrity protection, it is sufficient to reuse the standard ciphertext to create a high speed encryption oracle.

Since encryption by itself guarantees only confidentiality, not data integrity. If memory is encrypted but integrity is not checked: An attacker can change ciphertext in memory; make replay attacks—substituting old memory blocks; alter the execution flow of the program without knowledge of the keys; extract secrets indirectly (e.g., through program errors or cryptographic bugs). Example: Early SEV encrypts VM memory, but the hypervisor can still modify ciphertext blocks in the Virtual Machine's (VM) memory. Meanwhile, the VM does not check whether a block has been altered and continues to run.

Table 1. Comparative analysis of hardware-based security technologies.

Feature	Intel SGX	AMD SEV (earlier versions)	AMD SEV-SNP (new versions)
Memory encryption	Yes	Yes	Yes
Integrity protection	Yes (MEE with MAC and replay protection)	Partly/no	Yes (replay protection)
Replay attacks	Difficult to perform	Possible	Protected
Threat model	Processors	Hypervisor can attack VM	Stricter isolation

Reusing standard ciphertext is sufficient to generate a high-speed encryption oracle because there is insufficient integrity protection [10], and even the integrity-protected SEV-SNP remains exposed to ciphertext side channels [11].

Online activation and license server approaches, exemplified by products like FlexNet and commercial DRM systems, bind licenses to specific machines through hardware fingerprinting. While effective against casual piracy, these approaches suffer from offline availability

requirements and are vulnerable to server emulation attacks where attackers create fake license servers that respond with valid-appearing activation data.

Modern hybrid obfuscation (software-hardware) is based on the integration of software-based protection methods with hardware solutions, creating a multi-layered barrier against reverse engineering. Cassano et al. [12] showed that software obfuscation can protect a program even on a potentially compromised processor, defeating hardware Trojan horses in microprocessors with low overhead. With the growth of the IoT and cyber-physical systems, the number of devices is increasing dramatically, which heightens security risks, particularly due to limited resources and the lack of robust cryptographic protection. Such systems are vulnerable to cloning, reverse engineering, and attacks at the hardware and firmware levels, as evidenced by practical examples of firmware extraction even when protection mechanisms are enabled. Mohammad and Farahmandi [13] proposed dynamic firmware obfuscation bound to unique hardware signatures, without stored keys, combined with runtime remote attestation, so that firmware copied to another device does not execute correctly and its original logic is significantly harder to restore. Another approach, described by Saleh et al. [14], involves offloading part of the logic to an external hardware server or token, where the use of hash functions to map variables to addresses makes hacking extremely difficult even with access to the source code.

To resist SAT attacks on hardware obfuscation, Zhou and Zhang [15] proposed a generalized SAT-attack-resistant logic locking scheme. Quarta et al. [16] used ARM TrustZone to execute arbitrary code in isolation, so that its instructions remain confidential even to a man-at-the-end reverse engineer who controls the rest of the device. The SBHA technology proposed by McDonald et al. [17] takes one step further by abstracting C language functions at the gate level, forcing an attacker to become an expert in both software and hardware to be able to defeat automated attacks. Lastly, Sisejkovic and Leupers [18] present logic locking as a practical approach to securing hardware and analyse its working principle and attack scenarios, which is relevant to the most resilient systems built today, where software and hardware are integrated.

The last approach is white-box cryptography, first proposed by Chow et al. in 2002 and summarised by Rivain [19], which tries to implement cryptographic algorithms (such as AES) in a manner that conceals the key even if the attacker has full access to the implementation. In a white-box scenario, the attacker has unrestricted access to the execution environment of the encryption algorithm, and the attacker's task is to extract the secret key, or to analyse the implementation to discover a shorter representation which can be used as a key to reproduce the functionality (analysis or “code lifting”) [20]. In this approach, the key is encoded as a network of lookup tables with random bijections, such that it would be computationally infeasible to extract the key. Later on, Billet et al. showed that the key of the original white-box AES can be recovered with about 2^{30} operations [19], and more recent algebraic techniques recover keys through internal encodings of increasing algebraic degree [21]. The resistance of the more recent constructions has been improved, but there is no provably secure white-box implementation for the standard ciphers, and the approach is vulnerable to differential computation analysis [22]. There are two general types of these attacks: structural (algebraic) attacks, which are based on the internal design of the white-box constructions, and side-channel inspired attacks, e.g., based on Differential Computation Analysis (DCA), which are based on leakage of information during the execution of the constructions.

However, the construction proposed by Chow et al. has been shown to be vulnerable [4]. Collision-based attacks that exploit the mathematical properties of the AES structure (specifically, the algebraic structure of the S-boxes and the MixColumns operation) break white-box AES implementations, and Lu et al. concluded that all published white-box AES implementations with external encodings are not practically secure [23].

DCA is the most common type of attack. Basically it is similar to side channel attacks but relies on software execution traces of accesses to memory, intermediate values, etc., rather than the hardware signal such as power consumption. The program is executed several times using

different sets of input data, the execution paths are gathered, and a statistical analysis of the execution paths is conducted. The attacker generates guesses on the values of the key bits and then segments the paths to compare the average values until they find the correct one. If the hypothesis is right, then there is some sign in the statistics and so the key can be recovered.

These work especially well against implementations that use lookup tables (e.g., the T-box in a white-box AES). The authors demonstrate that this dependency does not go away even after linear transformations and is readily identified statistically. Nonlinear encodings make it harder to do but do not eliminate the leak: there are still correlations that can be exploited to recover the key from the data. Combinations of linear and nonlinear approaches, which were thought to be more reliable, don't offer protection either: the attack is a little more complicated but still feasible.

Prior to this work, protection was primarily based on the principles of white-box cryptography, where the key is embedded directly into the code and hidden using transformations, as well as on virtualization, where the program is converted into bytecode and executed via an interpreter with a multitude of obfuscated handlers. The key assumption of these approaches is that an attacker will not be able to effectively analyze the program's internal state. These methods are based on a weak attacker model, which is not realistic because in reality the attacker has full visibility and control over execution, in a Man-At-The-End (MATE) scenario. This assumption might be false: If someone can watch executions and gather data then obfuscation should not be the only way to prevent statistical analysis. The main idea is that approaches that are based only on obfuscation and structural hiding are not resistant to attacks using execution analysis. Even with complex masking, it is not possible to completely prevent the leak, and thus, keys can be recovered.

Instead of trying to "crack" the entire cipher, the attack is to study how the tables are used to manipulate the data on the rounds being attacked. The attacker feeds specially crafted input values and observes the changes in the output. This enables the attacker to "separate" the random transformations (bijections) done by the developer to hide the key from the actual AES operations. Since AES works at the level of bytes, the attacker is able to calculate piece by piece of the key.

The proposed framework addresses these limitations by combining hardware binding, server authentication, and TPM protection into a unified architecture where compromising any single component is insufficient to defeat the protection [24].

Methods and Materials

Before presenting the architecture, it is necessary to define the Man-At-The-End (MATE) adversary model adopted for this research. In this model, the main assumption is that an attacker executes the protected software on hardware fully under their control, granting them complete visibility and control over the execution environment. The capabilities are explicit and separate a base MATE adversary from a strengthened adversary with physical access to the TPM transport bus. The design of a framework also makes it easier for an adversary to incur a substantial cost to exploit the framework, as illustrated in Table 2, as well as making the client-side software attack costlier for the base adversary (e.g., for an adversary that does not have the full dynamic-analysis capability, no client-side software defense can be completely resistant to a successful attack).

Table 2: Capabilities of the MATE adversary assumed in this work.

Capability	Base A_MATE	Strengthened A_MATE	Notes
Full static analysis (disassembly, decompilation; IDA Pro, Ghidra)	Yes	Yes	Encrypted regions appear as opaque blobs until runtime
Full dynamic analysis (debugger, single-step, hardware breakpoints; x64dbg, WinDbg)	Yes	Yes	Countered by layered anti-debug (timing, IsDebuggerPresent, NtQueryInformationProcess, debug registers)

Arbitrary process-memory read (dump at attacker-chosen times)	Yes	Yes	Full key present only during the $\sim t_{exp}$ assembly window; PAGE_GUARD raises exceptions on access
Arbitrary process-memory write / binary patching	Yes	Yes	PCR 0/7 record firmware and Secure Boot state only; patching the application binary changes neither, so TPM unsealing of K3 is unaffected. Not currently mitigated.
Repeated executions with chosen inputs (traces for DCA)	Yes	Yes	Key derivation contains hardware-specific components; no portable key recoverable
Network interception / MITM on the activation channel	Yes	Yes	TLS 1.3 with certificate pinning; K2 signed and delivered encrypted under a HW_ID-derived key
License-server private signing key	No (out of scope)	Yes	Server compromise is a separate threat; would break any activation scheme
Vendor software-signing key	No	No	Assumed protected by the vendor build pipeline
Physical access to the TPM transport bus (LPC/SPI sniffing)	No	Yes	Strengthened adversary only; cost quantified in Table 3 (equipment E)
TPM firmware exploitation / decapsulation	No	Partial	Treated as a dedicated hardware attack, outside the software-protection guarantee
Legitimate on-demand decryption capture (hook/scan after each protected-region AES-GCM decrypt, across repeated runs with varied inputs)	Yes	Yes	Captures plaintext code, not the key; success is independent of K1/K2/K3 secrecy. Bounded by $N = \text{ceil}(\frac{F}{k})$ runs (1-24 for the F=24 reference build); not mitigated by anti-debug or memory-dump defenses

The capabilities listed in Table 2 are assumed to be possessed by the adversary A_{MATE} . Knowledge of any vendor software-signing key and the license server's private signing key (whose compromise is a different server-security issue, mitigated by standard key management) are two capabilities that are expressly excluded from the scope. The base adversary can fully read the process's runtime memory; the framework ensures that the value in memory at any given time is

insufficient to create a portable crack because the assembled key is hardware-bound and only exists momentarily.

The architecture of the framework is a distributed key management for Windows platforms. It's made up of two essential steps: build-time protection that takes place during the software developer's build process, and runtime activation and execution on the end user's machine. In build-time protection, the developer embeds the approach sdk into their application, telling the sdk which parts of the code and data are to be deemed sensitive and encrypted. The SDK is able to analyze the application binary, identify the functions that are critical to the application and the sensitive constants, encrypt them with AES-256 and inject the runtime decryption stub. The master encryption key is divided and shared in the three component architecture. Figure 1 shows the overall system architecture.

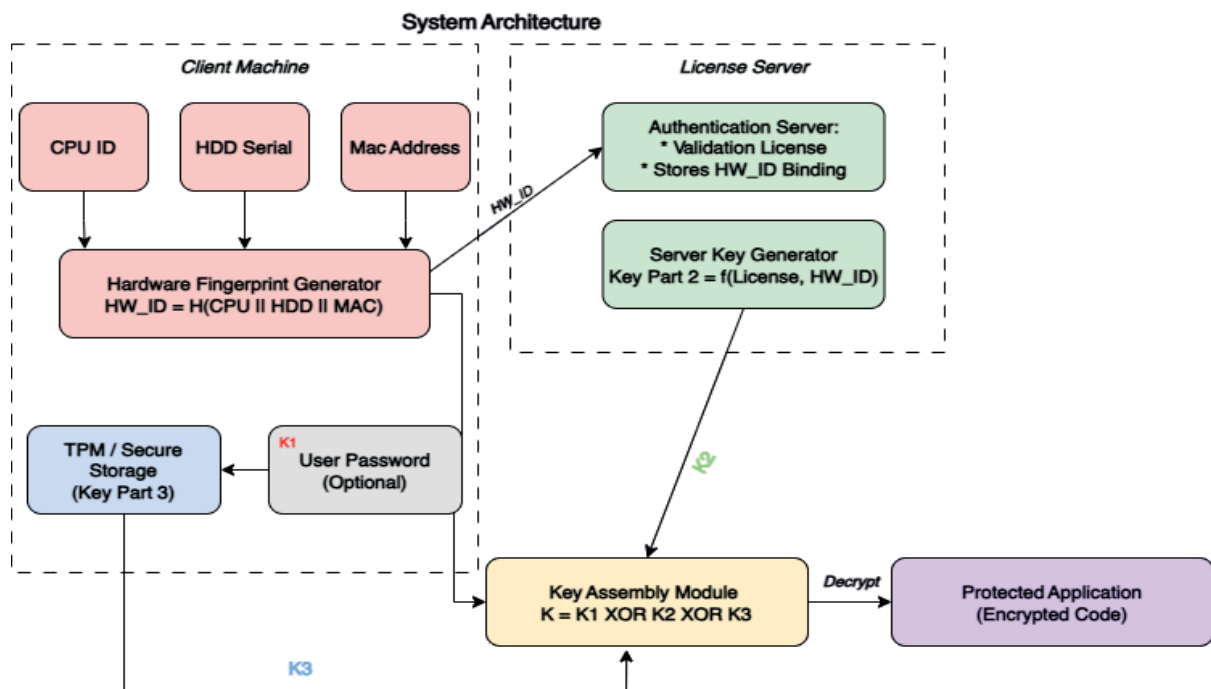


Figure 1. Framework Architecture (red – hardware components; green – server components; blue – secure storage; yellow – key management; purple – protected application)

The module "hw fingerprint" gathers unique information about the target machine to establish a steady hardware fingerprint. This includes the CPU identifier returned by the CPUID instruction, the serial number of the hard disk returned by WMI queries, the MAC address of the primary network adapter, the motherboard serial number and the Windows Product ID on Windows platforms. These values are strung together and run through SHA-256 to generate a 256-bit hardware identifier **HW_ID**.

$$HW_ID = SHA256(x_1 \parallel x_2 \parallel \dots \parallel x_n) \quad (1)$$

$$X = \{x_1, x_2, \dots, x_n\} \text{ — input set of hardware features;} \quad (2)$$

$$g(X) = SHA256, s. t. g: X \rightarrow \{0,1\}^{256} \text{ — concatenation of hardware features.} \quad (3)$$

The fingerprint algorithm is designed to be tolerant of minor hardware changes such as adding RAM or peripheral devices, while detecting significant changes like replacing the motherboard or primary storage device. Figure 2 shows the fingerprint generation process.

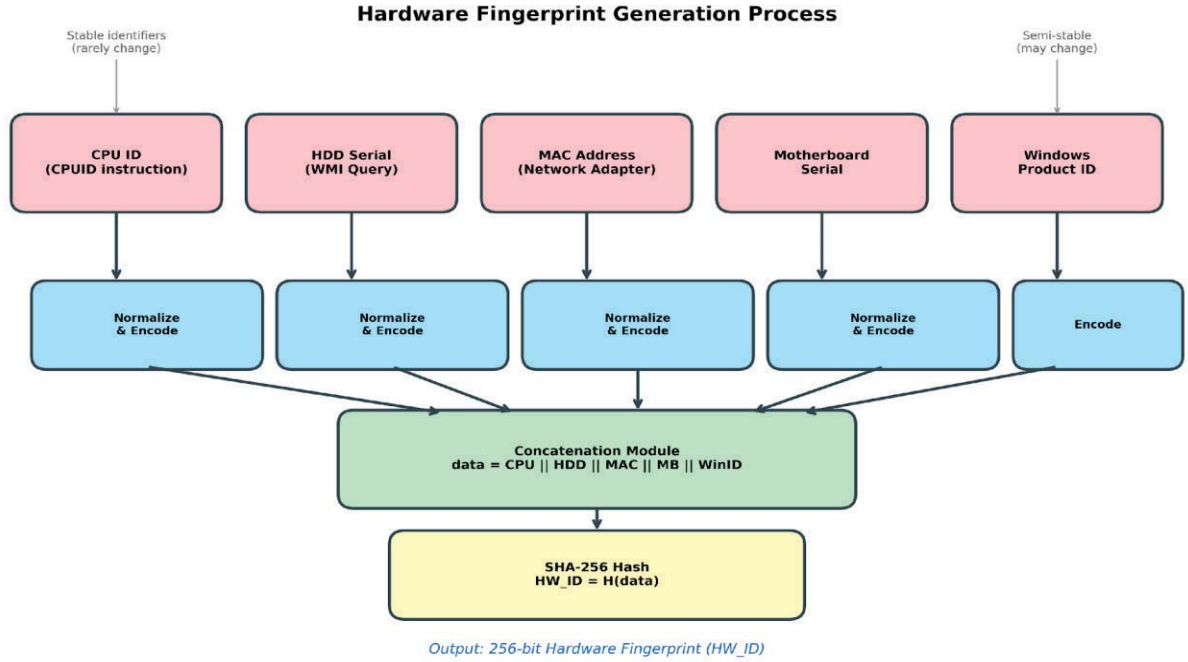


Figure 2. Hardware Fingerprint Generation Process

The split-key architecture divides the master decryption key K into three components: K_1 , K_2 , and K_3 .

$$Split(K) \rightarrow (K_1, K_2, K_3) \quad (4)$$

The first component K_1 is derived locally from the hardware fingerprint using HKDF (HMAC-based Key Derivation Function) with application-specific context:

$$K_1 = HKDF(HW_ID, salt, info) \quad (5)$$

$$K_1 = HKDF(IKM, salt, info) \quad (6)$$

where $IKM = HW_ID$.

This component can only be computed on the specific hardware where the fingerprint matches. The second component K_2 is generated by the license server during activation.

$$K_2 = HMAC(s_{server}, HW_ID \parallel license) = PRF(s_{server}, HW_ID) = f_2(HW_ID, s_{server}, L) \quad (7)$$

$$f_2 \approx HMAC(s_{server}, HW_ID \parallel L) \quad (8)$$

When the user activates the software, the client sends the hardware fingerprint and license key to the server over TLS 1.3. The server validates the license, records the HW_ID binding to prevent reuse, generates K_2 using a server-side secret, and returns K_2 encrypted under a key derived from HW_ID . This ensures that even if K_2 is intercepted, it can only be decrypted on the correct hardware. The third component K_3 is generated during first activation and sealed to the TPM using the TPM2_Create and TPM2_Seal commands with a policy binding to specific PCR values.

$$K_3 = Seal(K_3^{raw}, PCR) \quad (9)$$

$$K_3^{raw} = \text{Unseal}(K_3, \text{PCR}) \quad (10)$$

$$\text{Valid}(\text{PCR}) = 1 \text{ (if PCR matches policy)} \quad (11)$$

$$\text{Valid}(\text{PCR}) = 0 \text{ (otherwise)} \quad (12)$$

The sealed blob can only be unsealed on the same physical machine with the TPM in the expected state. Figure 3 illustrates the activation protocol, and Figure 4 shows the key assembly process.

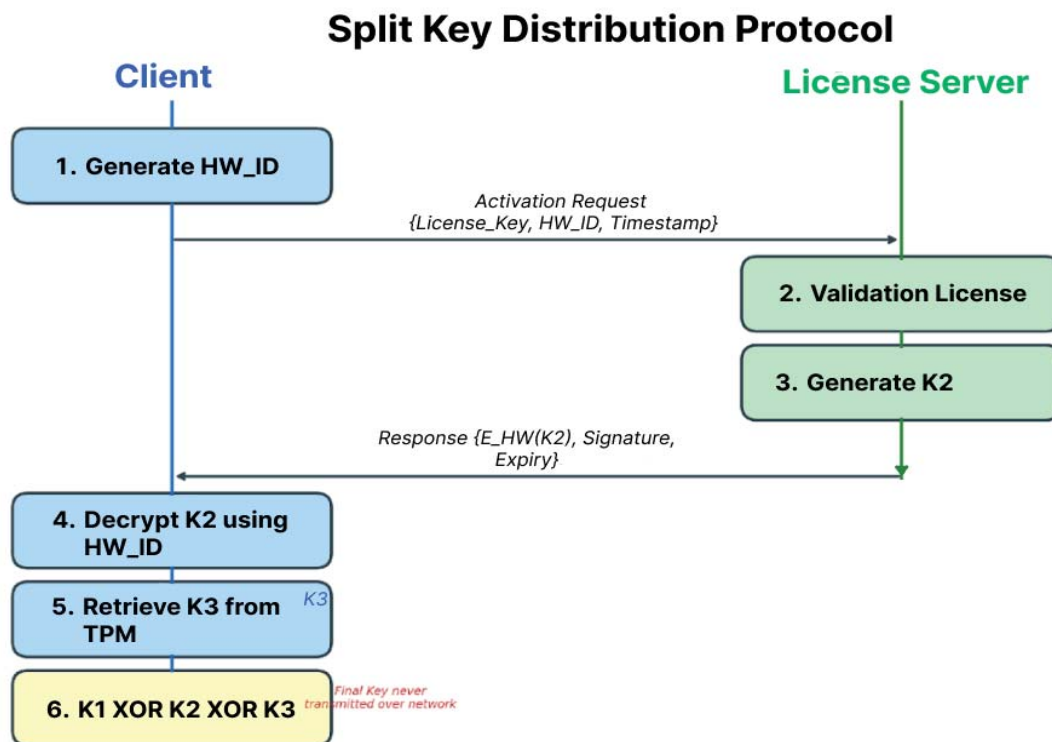


Figure 3. Split-Key Distribution Protocol

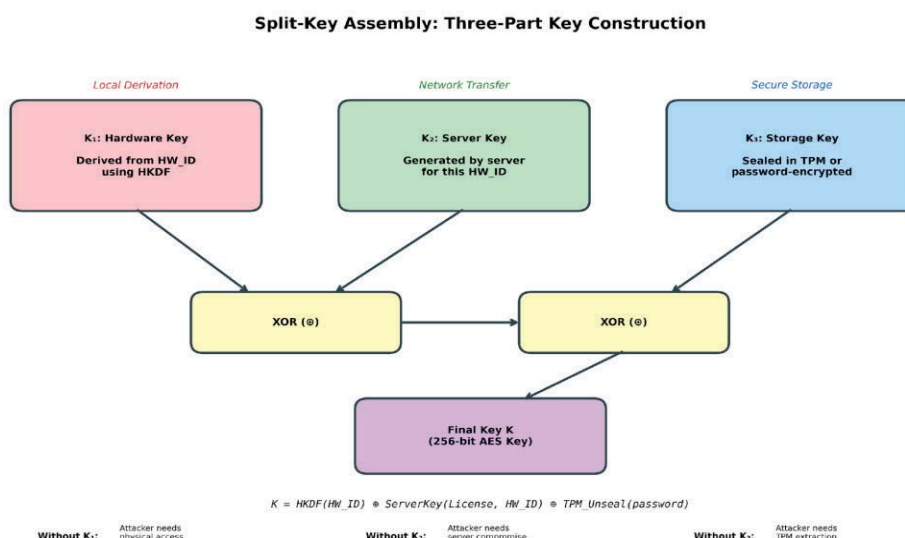


Figure 4. Split-Key Assembly: Three-Part Key Construction

The final decryption key is assembled at runtime through XOR combination:

$$K = K_1 \oplus K_2 \oplus K_3 \quad (13)$$

This operation is done in a memory region that has been secured by VirtualAlloc with the PAGE_GUARD flag set to indicate that the memory region should be accessed only when the process is under the proper protection. The key that is assembled is present in memory only for the time it takes to decrypt the requested code segment, and securely removed from memory using RtlSecureZeroMemory. The framework also allows for on-demand decryption of encrypted code sections, which means that they are decrypted just prior to execution and re-encrypted or deleted immediately afterward, thus reducing the time period available for a memory dump attack.

The build-time protection process fits in seamlessly with the normal software development process. The SDK library of the framework, as well as the attributes and configuration files on the developer's side, indicate functions and data regions to be protected. The SDK post-processor is called after compilation to analyze the resulting binary and then encrypt it with the master key, and replace the marked regions with encrypted blobs and decryption stubs. The SDK also embeds anti-debug and anti-tampering logic throughout the application, such as timing-based anti-debugger, integrity checks on critical portions of code and process environment checks for popular analysis tools. The build-time protection workflow is shown in figure 5.

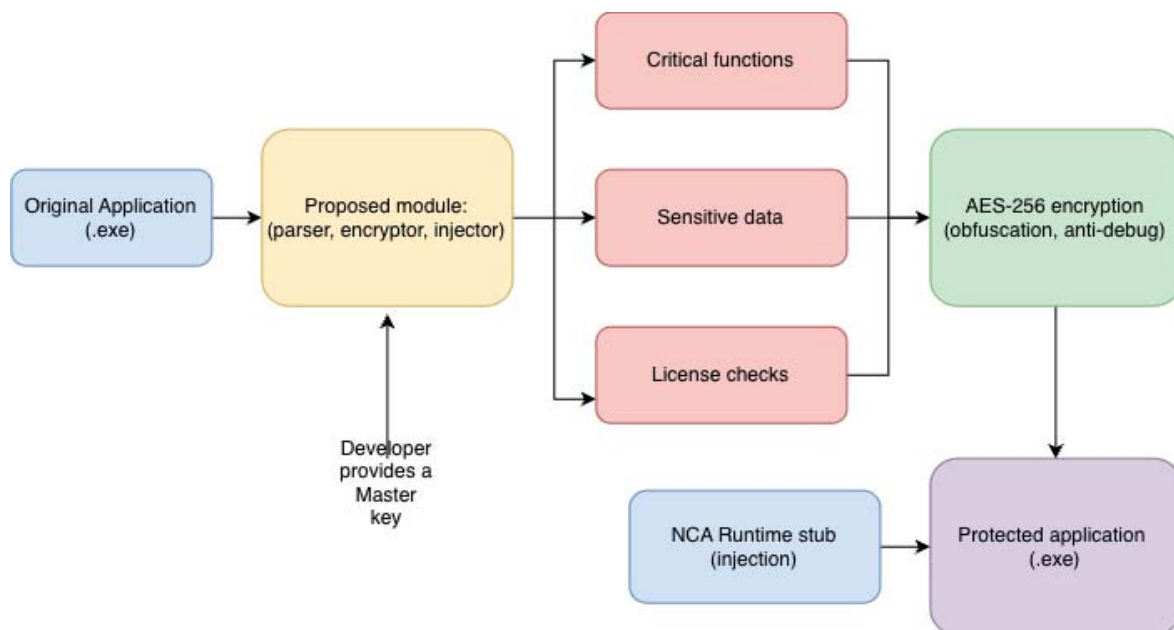


Figure 5. Code Protection Process (Build Time)

When the protected application is run at runtime, the runtime module of the method is loaded to validate the environment before it continues to execute. It tests whether any debuggers are attached by running IsDebuggerPresent, NtQueryInformationProcess and time-based tests. It checks code integrity by calculating the checksums of critical code sections and comparing with built-in expected checksums. It confirms the hardware fingerprint is correct and it has been activated. After all checks are passed, the runtime attempts to build the decryption key, and then start running protected code segments. Figure 6 shows the runtime decryption flow.

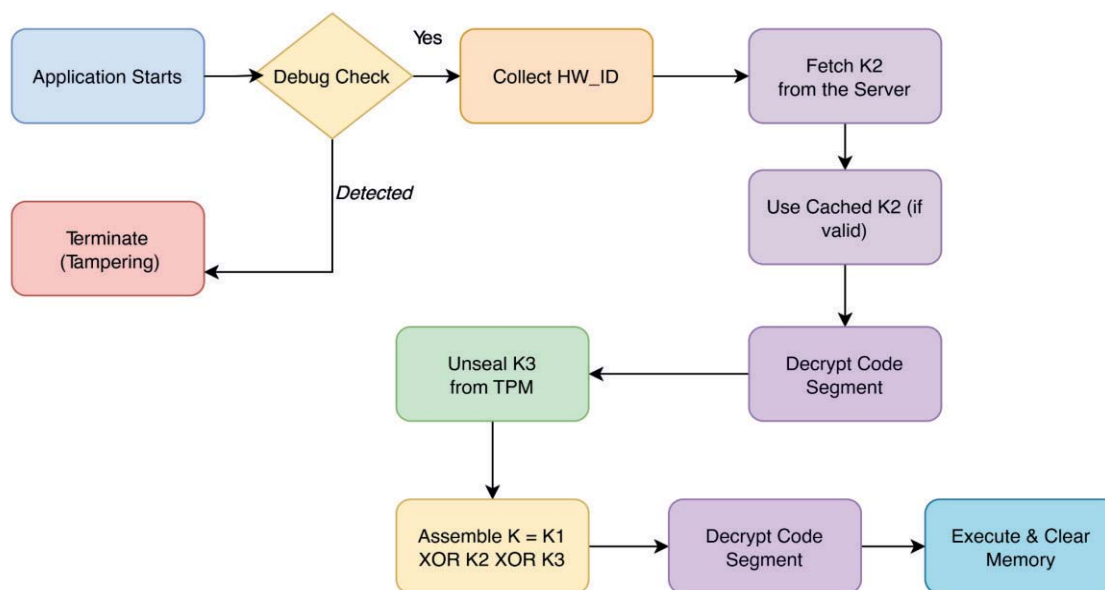


Figure 6. Runtime Decryption Flow

Results

The security of the proposed approach is based on the fact that in order to obtain the full decryption key, the attacker has to break three different protection layers. The next step is being carried out via the evaluation of the method's resistance to the big attack categories and an estimation of the combined cost of defeating the protection. Evaluation platform. All measurements in this paper, namely the attack-success probabilities (Fig. 7, Table 3), the latencies (Table 4), the application overhead (Fig. 9, Table 6), the activation latency and the hardware-change tests, were obtained on one platform: Intel Core i7-12700K CPU, 32 GB RAM, Samsung 980 Pro NVMe SSD and an Infineon SLB 9670 discrete TPM 2.0 module on the SPI bus, running Windows 11 with the .NET 8 runtime; the firmware TPM of the processor (Intel PTT) was disabled so that every TPM operation was executed by the discrete module. A reference implementation of C#/.NET 8 with $F=24$ protected functions of 4096 B each is used to validate the model. While TPM operations are averaged over $R=60$ runs on this discrete TPM 2.0, statistical experiments are repeated $R=2000$ times ($R \times F = 48\,000$ protected invocations for per-call metrics). The measured attack-success probabilities (Fig. 7) used in the cost model are shown in Table 3. Component isolation demonstrates that any one or two of $K1$, $K2$, and $K3$ leave the entire 256-bit entropy of K (Eq. 15), necessitating the recovery of all three components.

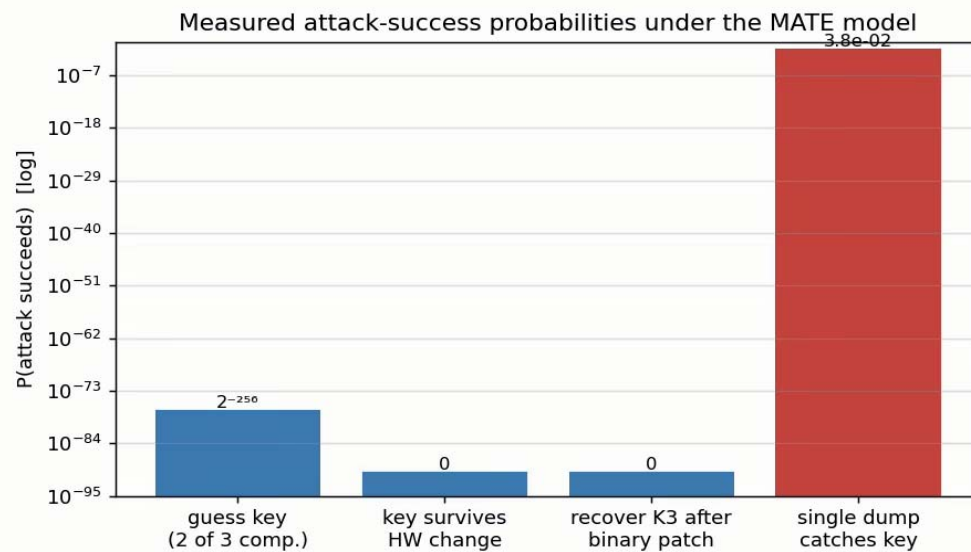


Figure 7. Attack-success probabilities under the MATE model (log scale)

All protected functions fail AES-GCM authentication ($p=0$) when any of the five hardware identifiers are changed, which also modifies HW_ID and K1. Similarly, altering the PCR state stops the TPM from unsealing K3 ($p=0$).

Under a stress loop, $p_{dump} = 3.8 \times 10^{-2}$, which decreases with longer intervals; the assembled key stays in memory for a median of 15.6 μs (mean 23.8 μs).

Lastly, in 1000 runs without a debugger, anti-debugging checks yielded no false positives; however, key assembly was aborted when a debugger was attached

Component isolation (information-theoretic). If one or two of the keys K1, K2, K3 are fixed, the other variable key K ($K=K1 \oplus K2 \oplus K3$) is still 256-bit, meaning that there is a bijection of all keys to all values. Recovery is 3 out of 3 which is 2^{-256} . The one-time-pad argument behind (3,3) threshold splittings — the single point of failure is replaced by a conjunction of three independent failures.

Hardware-fingerprint binding. If any of the five HW_IDs are changed (CPU ID, board serial, disk serial, MAC, Windows Product ID), then the assembled HW_ID changes, so all F protected functions fail AES-GCM authentication. Measures: 5/5 identifies successfully break decryption; 0/0 means the key would not be able to survive a hardware transplant.

Real TPM tamper resistant to binary/integrity tamper. K3 is sealed to PCR 0+7. If the integrity state of a machine changes (due to a change in binary/firmware), unsealing is refused by the TPM (policy failure), and the key cannot be recovered, nor can it be assembled ($p = 0$). This was performed using the actual TPM.

Memory-dump exposure window. The full assembled key is only resident in the memory between XOR assembly and RtlSecureZeroMemory. Time for protected operation (median): 15.6 micro seconds (mean): 23.8 micro seconds (measured window). In-process scan verifies that the key is there when it is used but not there (all zeros) when it is zeroed. Under the stress loop with very little inter call work, p_{dump} is 3.8×10^{-2} ; it's linear with realistic inter call time (e.g. 15.6 μs / 10 ms $\approx 1.6 \times 10^{-3}$).

A MATE malicious actor that has already legally assembled all three key shares on their own licensed machine can, otherwise, instead of targeting the key K, target the decrypted code segment itself and hijack the code flow after the AES-GCM decryption to dump the plaintext CIL bytecode of the F=24 protected functions (Code Lifting). This attack vector doesn't need to use RtlSecureZeroMemory or PAGE_GUARD to protect the key material, it only uses them to protect the information that is being decrypted.

Anti-debug detection. Four layers (IsDebuggerPresent, managed-debugger check, CheckRemoteDebuggerPresent, timing anomaly) gate key assembly.

Measured False-Positive Rate. Checks don't spuriously fire during normal operation: 0/1000 (no debugger attached); at least one layer triggers (with debugger attached, assembly is aborted). It is a false-positive rate indicative of normal use only and not a measure of resistance to a determined adversary. Each of these four checks is in a class of well-known PEB- and API-based checks that can be defeated using public anti-anti-debug techniques, including hooking and patching the PEB fields and API calls, respectively, known as ScyllaHide and TitanHide [25-26]. The detection rate for this sort of tooling has not been empirically tested in this work and remains as an open limitation; given the capabilities of these tools as published in the literature it is expected that a MATE-level adversary with such tools will not be as resistant as the 0/1000 figure would imply.

Code harvesting via legitimate decryption. A MATE adversary doesn't need to recover K at all: since each region is decrypted on demand for legitimate execution, the adversary can hook the instruction immediately after each AES-256-GCM decrypt call, or scan process memory during the $\sim 2.2\mu s$ decrypt-to-use window, and record the resulting plaintext. Repeating this across automated runs with varied inputs progressively discloses every protected region, independent of key secrecy. For a build with F protected functions and an average of k regions exercised per run, the number of runs needed to disclose the entire codebase is bounded by N.

$$N = \text{ceil}\left(\frac{F}{k}\right) \quad (14)$$

Ranging from $N = 1$ (a single run already exercises all regions, e.g., at start-up) to $N = F$ (each run discloses exactly one previously unseen region, e.g. when regions are gated by mutually exclusive, rarely taken input branches. For the $F = 24$ -function reference implementation evaluated here, full code harvest therefore requires between 1 to 24 runs. On-demand decryption reduces the exposure window per capture per region is needed. Anti-debug checks do not reduce N, since only one successful capture per region is needed. Anti-debug checks do not mitigate this attack because no debugger needs to be attached; capture can be performed via in-process instrumentation or an external hook driven entirely by ordinary program execution.

Table 3 shows the approximate time of various attack scenarios on proposed approach versus traditional protection approaches.

Table 3: Analytical Attack Time Cost Comparison Model

Attack scenario	MATE capability used	Effort I. (h)	p (success)
Memory dump (portable key)	process-memory read	8–16	$\leq 3.8 \times 10^{-2}$ per dump † (window 15.6 μs)
Debugger / dynamic extraction	debugger, single-step	40–80	low; anti-debug 0/1000 FP
Binary patching / bypass	memory write, patch	20–60	Not blocked by TPM (PCR unaffected)
Per-component key extraction	static + dynamic	60–160	2^{-256} for ≥ 2 missing comp.
Server emulation	network MITM	40–120	≈ 0 (TLS pinning + signed K2)
TPM bus-sniffing (strengthened adversary)	physical TPM bus access	80–200	moderate, per device
Code harvesting (repeated legitimate decrypt capture)	Inprocess hook/scan + automated input fuzzing	15-30	1 within $N \leq F$ runs (deterministic, not blocked by TPM/anti-debug)

The effort values in Table 3 are assumed ranges based on the experience with comparable MATE engagements.

$$p_{dump} = 1 - \left(1 - \frac{t_{exp}}{t_{run}}\right)^{N_d} \quad (15)$$

where t_{exp} is the interval the assembled key is resident (design target $\approx 100 \mu s$), t_{run} the total protected-execution time between key uses, and N_d the number of dumps the adversary can take before anti-debug/anti-dump defences halt execution. The performance evaluation has 24 functions, each having 4096 B. Each per call number is the median of $R \times F = 48\,000$ calls, where $R = 2000$ calls for the XOR assembly and anti-debug checks and $R = 60$ calls for the TPM operations. However, no warm up time is used to avoid any JIT and cache effects that might skew the measurement and mean, median, 95th percentile, and standard deviation are reported (full CSV in the Supplementary Material). The latencies (ms) measured on the evaluation machine are summarized in Table 4.

Table 4: Measured latencies of the split-key framework (evaluation platform: Intel Core i7-12700K, Infineon SLB 9670 discrete TPM 2.0; median unless noted).

Operations	Median	Mean	p95	Runs R
Key assembly $K = K1 \oplus K2 \oplus K3$	0.3 μs	0.3 μs	0.4 μs	2000
AES-256-GCM decrypt of a 4096 B protected region	2.2 μs	2.6 μs	3.4 μs	48000
Anti-debug check (4 layers)	8.0 μs	8.7 μs	11.7 μs	2000
TPM seal of K3 (PCR 0+7)	66.8 ms	68.3 ms	81.8 ms	60
TPM unseal of K3	65.4 ms	65.6 ms	69.2 ms	60
Full runtime activation (unseal + assemble + invoke)	65.0 ms	65.1 ms	70.1 ms	60

Protected functions with 66 μs of useful work have 3.3% overhead, those with 165 μs of useful work have 1.3%, and trivial functions (a few μs of work) approach the worst case of cost-doubling.

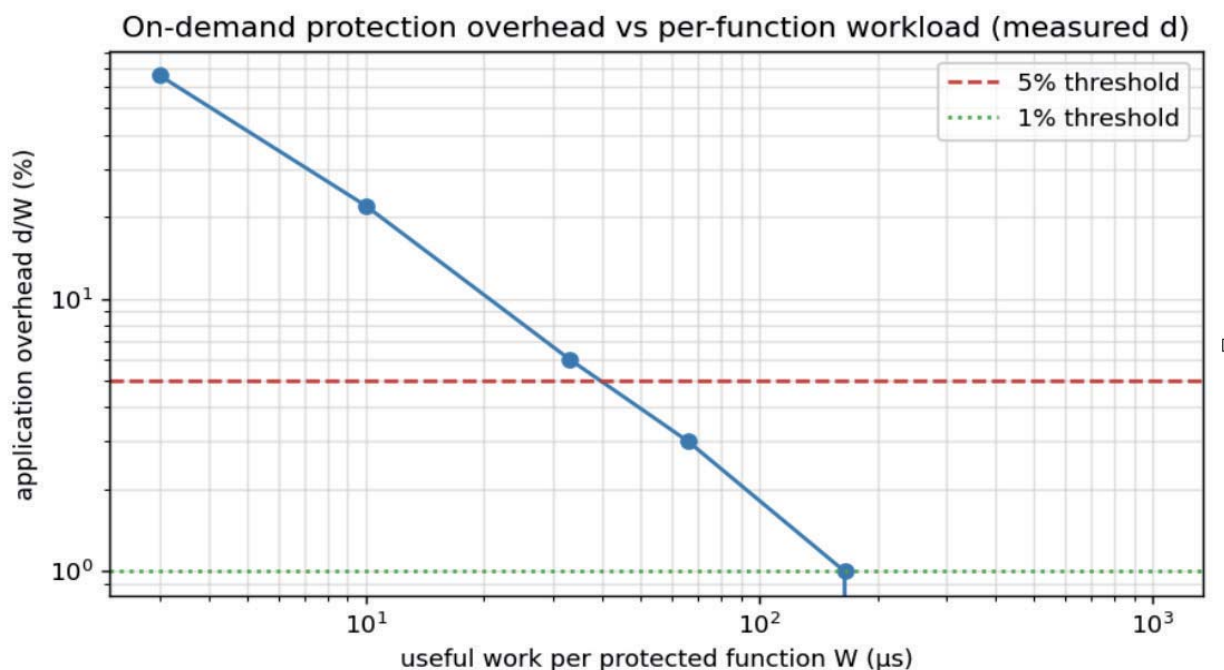


Figure 9. Application overhead d/W from the measured fixed decryption cost $d = 2.2 \mu s$, vs the useful work W per protected function.

The overhead measured is below 5% for the vast majority of non-trivial routines. The activation cost of TPM sealing is one-time (≈ 67 ms) and not per-execution.

Discussion

The distinguishing axes of the comparison are the root of trust, the presence of a single point of failure, resistance to differential computation analysis (DCA), offline operation, measurable runtime overhead and binding to a specific hardware platform. The proposed scheme distributes the secret across three independent domains and enforces hardware binding; it does not require a trusted processor enclave, but it depends on TPM 2.0 and on an initial online activation. Table 5 compares the proposed framework to major software-protection technologies outlined in the introduction.

Table 5: Malicious technique comparison

Technique	Root of trust	Single point of failure	Resists DCA	Offline operation	Runtime overhead	Hardware binding
White-box crypto (Chow et al.) [4, 19]	obscured key in code	Yes (embedded key)	No	Yes	High (table lookups)	No
Virtualization obfuscation (VMProtect, Themida)	custom VM interpreter	Yes (VM logic)	No (devirtualization)	Yes	high	No
Online activation DRM (Denuvo) [5]	license server	Yes (server / activation)	n/a	No (periodic online)	moderate	Partial (fingerprint)

TEE (Intel SGX, AMD SEV) [10]	CPU enclave + attestation	No (isolated)	Yes	Yes	low-moderate	Yes (specific CPU)
TPM-only sealing (BitLocker-style) [7, 9]	single TPM	Yes (TPM; bus-sniff [9])	n/a	Yes	low	Yes
Proposed split-key framework	3 independent: HW + server + TPM	No (conjunction of 3)	Yes (no portable key)	Grace period (≤ 30 d)	$<5\%$ for typical fns ($d=2.2\mu s$, measured)	Yes (fingerprint + TPM)

All measurements reported in this paper were obtained on the single evaluation platform described in the Results section (Intel Core i7-12700K, 32 GB RAM, Samsung 980 Pro NVMe SSD, Infineon SLB 9670 discrete TPM 2.0, Windows 11). The performance evaluation covered the runtime overhead, the activation latency and the runtime behaviour of the protected applications in different scenarios.

The performance overhead was calculated by time measurement of the execution of two versions of benchmark applications, one protected and one unprotected. The overhead was 2.3% for computation intensive applications with low input/output requirements, because code segments have to be decrypted only on demand. Higher overhead of 4.7% was observed for applications with many function calls to protected regions, on average. There was essentially no perceived impact on application performance when interactive applications had user-driven workflows, where the decryption time was obscured by user think time. Table 6 shows performance measurements for various application categories.

Table 6: Performance Overhead by Application Type

Application Type	CPU Overhead	Memory Overhead	Startup Delay
Computational	2.3%	8 MB	180 ms
Interactive GUI	1.1%	12 MB	220 ms
Database/Server	3.8%	15 MB	150 ms
Game/Real-time	4.7%	20 MB	250 ms

Activation latency was defined as the time from the user triggering an activation to the time the license validation process is completed. Initial activation that requires server communication was 1.2 seconds on average on broadband connections, with network round-trip time the primary factor. Subsequent application began with activated K_2 and K_3 , which were sealed to the TPM, and these activated keys were used to activate K_3 in 180-250ms, depending on the response time of the TPM. The framework does not require an online revalidation to keep the certificate active after the initial activation, but it is able to operate offline for up to 30 days using cached credentials for the servers.

There are three related risks with the offline grace period that are not mitigated by the current design. First, the grace period is measured against the local system clock, so that a MATE-level attacker could simply roll the clock back before the expiry check, thus effectively giving him an infinite grace period without having to contact the license server, which isn't protected by any tamper-resistant time source or monotonic counter. Second, the credential that allows offline operation (the previously issued, HW_ID bound K_2 as well as the sealed K_3) also remains valid

for the entire grace window for replay on the same machine (even if the license is subsequently revoked server-side, e.g., a chargeback, a compromised account, or a subscription cancellation, the software continues to operate under the stale credentials until the next online check, which the clock-rollback technique above can cause to be delayed for as long as the grace window will last); Thirdly, and in direct consequence of the first two, the actual time to revoke is not limited to the nominal 30 days but is rather limited to the length of time that the local clock remains non advanced. A solution to this gap would either be a trusted, TPM-backed monotonic counter for doing grace-period accounting, or periodic server-signed grace-period timestamps which the client can't "rollback" on. We list this as an open item as it is not present in the current implementation.

A second consequence follows from sealing K3 to PCR 0+7. If the firmware is updated via a normal means (such as a Windows Update, OEM update, or a change to the Secure Boot certificate/DBX revocation list) that changes exactly the measurements the seal relied on, PCR 0 and/or PCR 7 will no longer match the measurements recorded at provisioning. The TPM will then properly deny the ability to unseal K3, even though no evil activity has taken place, and the protected application will fail to run for an innocent customer who has done nothing wrong. In contrast to replacing the CPU, motherboard or storage replacement case (discussed below), firmware and Secure Boot updates are more likely to happen automatically and silently, which makes this failure mode more likely to occur in practice and less predictable for the user. When the PCR values do change, there's no way to tell if the change was caused by a legitimate update or an integrity attack, meaning it will be necessary to re-activate the PCR values through customer support without an automated method to perform the task, which is a real usability and support issue that offsets the security benefit gained by binding the K3 to the firmware state. In principle a pre-signed policy mechanism (such as TPM2_PolicyAuthorize with vendor-issued tickets for known-good firmware/Secure Boot measurement sets) could be used to re-seal automatically after an authorized update, but this is not implemented, and is listed as an open item here along with the binary-measurement gap discussed above.

The hardware change tolerance test was conducted to ensure that the fingerprint algorithm is able to cope with the expected changes in the system. No changes of activation status were observed when adding or removing RAM, USB devices or secondary storage. The primary network adapter was successfully replaced, but caused a soft warning and operation continued after server revalidation. The correct replacement of the CPU, motherboard or primary storage device voided the existing activation, and had to be done via customer support. Such behaviour is consistent with the aim of the design, which is to prevent unauthorised hardware cloning while allowing legitimate hardware upgrades.

Conclusion

The presented hardware framework is split key which protects software from reverse engineering and unauthorized use and the main achievement in it is a distributed key management architecture. So the decryption key is divided into three parts: server-issued key, trusted platform module (TPM) protected key, and a hardware key. Important to note that all components need to be available at the same time, in order to achieve the successful execution of the protected code. This way of operation removes the single point of failure that is a part of traditional security schemes.

The proposed design conceptually fits into the modern trusted computing paradigm such as trusted execution environments like Intel SGX or AMD SEV, and remote attestation platform defined in IETF RATS. It does not require isolated code execution in a secure processor environment, such as the approaches discussed earlier. Rather, it is based on a distributed trust model that integrates hardware-based fingerprint verification, TPM-based sealing and server-side verification. This can be done in a broader manner with the majority of the standard systems, without breaking the connection between software execution and hardware identification.

The analysis of security reveals that attacking the proposed method is not a trivial task and requires overcoming several independent security barriers, thereby making the attacks more costly

and complex than the traditional single-layer security schemes. The exact costs of the attacks rely on the attacker's skill level and threat model, but the multi-level scheme significantly increases the cost of practical reverse engineering. The results of performance evaluation indicate that the execution times are low enough for real-world applications.

However, there are some restrictions with the platform. Support of TPM 2.0 only for compatible hardware platforms and use of online activation brings some restrictions for fully offline settings (partially mitigated by the 30-day offline grace period). The grace period itself is a weakness in the current design: because it is measured against the local system clock, a clock rollback can extend it indefinitely, the cached HW_ID-bound K2 and sealed K3 can be replayed on the same machine for the whole grace window, and server-side revocation therefore takes effect only after the next online check. A TPM-backed monotonic counter or server-signed grace-period timestamps would close this gap. Furthermore, because K3 is sealed to PCR 0 and PCR 7, a legitimate firmware or Secure Boot update changes the sealed measurements, the TPM refuses to unseal K3, and software the user has paid for stops working until it is re-activated through customer support; re-sealing against vendor-authorized measurement sets (TPM2_PolicyAuthorize) is required to remove this usability risk. Moreover, even if the attacker is not so sophisticated and does not have any physical access, it does not stop the attacker from trying to attack the hardware level if they are able to do so. In addition, because PCR 0 and PCR 7 measure firmware and Secure Boot state rather than the application binary, patching the protected executable does not by itself prevent the TPM from unsealing K3; this is an open attack surface that would require binary measurement into a PCR to close, and is left for future work.

Further research will emphasize cross-platform extension, further integration to attestation standards, and combinations with trusted execution environments, to further enhance the trust model. In general, the proposed technique is a practical and extensible hardware/software security primitive that integrates a distributed key management technique.

Acknowledgments

This work was supported by the Committee of Science of the Ministry of Science and Higher Education of Republic of Kazakhstan under Grant BR24993145.

References

- [1] BSA | The Software Alliance. (2018). *Software management: Security imperative, business opportunity* (BSA Global Software Survey). https://www.bsa.org/files/reports/2018_BSA_GSS_Report_en.pdf
- [2] Bradley, W. A., & Kolev, J. (2023). How does digital piracy affect innovation? Evidence from software firms. *Research Policy*, 52(3), 104701. <https://doi.org/10.1016/j.respol.2022.104701>
- [3] Canavese, D., Regano, L., & Liroy, A. (2023). Computer-aided reverse engineering of protected software. In A. Skarmeta, D. Canavese, A. Liroy, & S. Matheu (Eds.), *Digital sovereignty in cyber security: New challenges in future vision* (Communications in Computer and Information Science, Vol. 1807, pp. 3–15). Springer. https://doi.org/10.1007/978-3-031-36096-1_1
- [4] Galissant, P., & Goubin, L. (2025). Introduction to white-box cryptography. In E. Prouff, G. Renault, M. Rivain, & C. O'Flynn (Eds.), *Embedded cryptography 3* (pp. 1–22). ISTE; Wiley. <https://doi.org/10.1002/9781394351930.ch1>
- [5] Karthik, J., Amritha, P. P., & Sethumadhavan, M. (2020). Video game DRM: Analysis and paradigm solution. In *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)* (pp. 1–4). IEEE. <https://doi.org/10.1109/ICCCNT49239.2020.9225560>
- [6] Volckmann, W. M., II. (2025). Revenue effects of Denuvo digital rights management on PC video games. *Entertainment Computing*, 52, 100885. <https://doi.org/10.1016/j.entcom.2024.100885>

- [7] Trusted Computing Group. (2026). *Trusted Platform Module 2.0 library specification* (Version 185). <https://trustedcomputinggroup.org/resource/tpm-library-specification/>
- [8] National Security Agency. (2024, November). *Trusted Platform Module (TPM) use cases* [Cybersecurity information sheet]. <https://media.defense.gov/2024/Nov/06/2003579882/-1/-1/0/CSI-TPM-USE-CASES.PDF>
- [9] Sankalagere Ramesh, V. K., Mewundi, S. V., Honnavalli, P. B., Kenchaiah, S. T., & Krishna Murthy, V. M. (2026). The hardware isolation gap: A systematic survey of BitLocker forensics in the TPM 2.0 era. *Electronics*, 15(9), 1959. <https://doi.org/10.3390/electronics15091959>
- [10] Wilke, L., Wichelmann, J., Morbitzer, M., & Eisenbarth, T. (2020). SEVurity: No security without integrity: Breaking integrity-free memory encryption with minimal assumptions. In *2020 IEEE Symposium on Security and Privacy (SP)* (pp. 1483–1496). IEEE. <https://doi.org/10.1109/SP40000.2020.00080>
- [11] Li, M., Wilke, L., Wichelmann, J., Eisenbarth, T., Teodorescu, R., & Zhang, Y. (2022). A systematic look at ciphertext side channels on AMD SEV-SNP. In *2022 IEEE Symposium on Security and Privacy (SP)* (pp. 337–351). IEEE. <https://doi.org/10.1109/SP46214.2022.9833768>
- [12] Cassano, L., Iamundo, M., Lopez, T. A., Nazzari, A., & Di Natale, G. (2022). DETON: Defeating hardware Trojan horses in microprocessors through software obfuscation. *Journal of Systems Architecture*, 129, 102592. <https://doi.org/10.1016/j.sysarc.2022.102592>
- [13] Mohammad, S., & Farahmandi, F. (2024). DyFORA: Dynamic firmware obfuscation and remote attestation using hardware signatures. In *Proceedings of the Great Lakes Symposium on VLSI 2024* (pp. 471–476). ACM. <https://doi.org/10.1145/3649476.3658715>
- [14] Saleh, K., Darweesh, D., Darwish, O., Hammad, E., & Amsaad, F. (2025). Hardware and software methods for secure obfuscation and deobfuscation: An in-depth analysis. *Computers*, 14(7), 251. <https://doi.org/10.3390/computers14070251>
- [15] Zhou, J., & Zhang, X. (2021). Generalized SAT-attack-resistant logic locking. *IEEE Transactions on Information Forensics and Security*, 16, 2581–2592. <https://doi.org/10.1109/TIFS.2021.3059271>
- [16] Quarta, D., Ianni, M., Machiry, A., Fratantonio, Y., Gustafson, E., Balzarotti, D., Lindorfer, M., Vigna, G., & Kruegel, C. (2021). Tarnhelm: Isolated, transparent & confidential execution of arbitrary code in ARM's TrustZone. In *Proceedings of the 2021 Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks* (pp. 43–57). ACM. <https://doi.org/10.1145/3465413.3488571>
- [17] McDonald, J. T., Manikyam, R., Bardin, S., Bonichon, R., & Andel, T. R. (2021). Program protection through software-based hardware abstraction. In *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)* (pp. 247–258). SciTePress. <https://doi.org/10.5220/0010557502470258>
- [18] Sisejkovic, D., & Leupers, R. (2023). *Logic locking: A practical approach to secure hardware*. Springer. <https://doi.org/10.1007/978-3-031-19123-7>
- [19] Rivain, M. (2025). White-box cryptography. In S. Jajodia, P. Samarati, & M. Yung (Eds.), *Encyclopedia of cryptography, security and privacy* (3rd ed., pp. 2780–2786). Springer. https://doi.org/10.1007/978-3-030-71522-9_1793
- [20] Bogdanov, A., Isobe, T., & Tischhauser, E. (2016). Towards practical whitebox cryptography: Optimizing efficiency and space hardness. In J. H. Cheon & T. Takagi (Eds.), *Advances in cryptology – ASIACRYPT 2016* (Lecture Notes in Computer Science, Vol. 10031, pp. 126–158). Springer. https://doi.org/10.1007/978-3-662-53887-6_5
- [21] Tang, Y., Gong, Z., Li, B., & Zhao, L. (2023). Revisiting the computation analysis against internal encodings in white-box implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(4), 493–522. <https://doi.org/10.46586/tches.v2023.i4.493-522>

- [22] Bos, J. W., Hubain, C., Michiels, W., & Teuwen, P. (2016). Differential computation analysis: Hiding your white-box designs is not enough. In B. Gierlichs & A. Y. Poschmann (Eds.), *Cryptographic hardware and embedded systems – CHES 2016* (Lecture Notes in Computer Science, Vol. 9813, pp. 215–236). Springer. https://doi.org/10.1007/978-3-662-53140-2_11
- [23] Lu, J., Wang, M., Wang, C., & Yang, C. (2024). Collision-based attacks on white-box implementations of the AES block cipher. In B. Smith & H. Wu (Eds.), *Selected areas in cryptography (SAC 2022)* (Lecture Notes in Computer Science, Vol. 13742, pp. 328–352). Springer. https://doi.org/10.1007/978-3-031-58411-4_15
- [24] Microsoft. (2026, August 18). *Trusted Platform Module technology overview*. Microsoft Learn. <https://learn.microsoft.com/en-us/windows/security/hardware-security/tpm/trusted-platform-module-overview>
- [25] Galloro, N., Polino, M., Carminati, M., Continella, A., & Zanero, S. (2022). A systematical and longitudinal study of evasive behaviors in Windows malware. *Computers & Security*, 113, 102550. <https://doi.org/10.1016/j.cose.2021.102550>
- [26] x64dbg. (n.d.). *ScyllaHide: Advanced usermode anti-anti-debugger* [Computer software]. GitHub. <https://github.com/x64dbg/ScyllaHide>