International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

Secure Runtime Software Protection Tool (SRSPT): A Block-Level AES-256 Runtime Decryption Framework for Anti-Piracy and Tamper-Resistant Software Execution

Milind R. Garge, Ayush K. Malwatkar, Ayush S. Kapure
Department of Computer Technology, Ahinsa Institute of Technology Dondaicha, Maharashtra, India
Guide: Assist. Prof. Kalpesh Marathe

***Abstract*—**Proprietary software products are constantly threatened by memory-based binary analysis, software piracy, and unlawful reverse engineering. A basic flaw of both whole-binary encryption and static obfuscation is that the full decrypted program is eventually put into process memory, creating a predictable and exploitable attack surface. In order to overcome this restriction, this study introduces the Secure Runtime Software Protection Tool (SRSPT), a conceptual cybersecurity protection framework that suggests a runtime block decryption architecture. A built binary is theoretically divided into discrete, individually encrypted logical chunks using AES-256-CBC [1] and RSA-2048 asymmetric key wrapping [2]. Each block has a unique key. Each block's integrity is checked using SHA-256 hashing prior to execution [3]. Only the block that is now running is decrypted during runtime into a controlled memory space, which is then safely zeroed before the subsequent block is loaded. By ensuring that the entire plaintext binary is never concurrently present in RAM, this sequential single-block-in-memory discipline significantly reduces the window of opportunity for memory-scraping and dumping assaults. On an x86-64 architecture with AES-NI acceleration, an analytically predicted runtime overhead of 4–7% is presented. Conceptually, the design is contrasted with the function-level runtime encryption method that served as its inspiration [7]. Inspired by well-established runtime protection, trusted execution, and ephemeral decryption principles, this work is an implementation-oriented conceptual research study [4][5][6][7].

AES-256 block decryption, RSA-2048 key wrapping, SHA-256 integrity verification, anti-piracy, prevention of reverse engineering, secure loader, memory cleanup, software IP protection, ephemeral decryption, block-level encryption, binary protection, threat model, and conceptual framework are among the index terms.

I. INTRODUCTION

Binary tampering, illegal reverse engineering, and software piracy cause significant financial and intellectual property losses for the commercial software sector. Even though businesses frequently spend money on licensing management servers and activation systems, an attacker with enough motivation can still access the underlying binaries. For independent game creators, new software firms, and businesses whose competitive edge is derived on proprietary algorithms integrated into their products, this problem is especially severe.

Static packaging, obfuscation, virtualization-based obfuscators, and hardware dongles are examples of current protection techniques [5]. The fact that the entire decrypted binary must finally exist in process memory for execution—at which point memory-dumping techniques can recreate the original code—is a basic constraint shared by the majority of algorithms [6].

The Secure Runtime Software Protection Tool (SRSPT), a hypothetical cybersecurity protection architecture that addresses this exposure window via a suggested runtime block decryption technique, is presented in this study. Instead of loading the full decrypted program at once, the suggested design splits the binary into logical blocks, encrypts each one separately using AES-256-CBC [1], and decrypts only the block that is currently required at runtime. Before the subsequent block is introduced, the plaintext of that block is instantly rewritten after its execution is finished. As a result, RAM never contains the entire plaintext binary at the same time.

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

Gurajapu and Agarwal [7], who showed selective function-level encryption and ephemeral in-memory decryption for source-code IP protection, served as the intellectual inspiration for this study. By using a block-level egmentation design with RSA-2048 asymmetric key wrapping [2] and a separate Secure Loader component, SRSPT expands these ideas to the compiled-binary level. Software protection literature [5][6] and trusted executionenvironments [4] provide more design ideas. Although the suggested design significantly increases the practical cost of binary extraction attacks, SRSPT does not offer unbreakable protection.

This conceptual research framework's main contributions are: (i) a suggested block-level runtime decryption architecture that combines AES-256-CBC [1], RSA-2048-OAEP [2], and SHA-256 [3]; (ii) architectural descriptions of the Preprocessing and Encryption Engine (PEE), Secure Loader (SL), and Runtime Block Manager (RBM); (iii) a discussion of analytical overhead estimation; (iv) a structured conceptual comparison against the function-level prior art [7]; and (v) an expanded security analysis against representative attack classes that includes a formal threat model.

II. EXISTING SOLUTIONS AND LIMITATIONS

A. Static Packing and Obfuscation

Binaries on disk are compressed and optionally encrypted by packers, which then decompress during launch and return control to the original entry point [5]. The entire decompressed image is put into memory at once, and memory-dumping tools can recreate the original binary from this single encrypted snapshot, even when packers prohibit casual viewing of stored binaries. Dead code insertion, opaque predicates, and control-flow flattening are examples of obfuscation techniques that increase analytical difficulty but do not stop a motivated attacker with automated tools from recovering program semantics [5].

B. Virtualization-Based Obfuscation

Code segments are converted by commercial tools into proprietary virtual machine bytecode that is run by an embedded interpreter. Although the difficulty of reverse engineering is significantly increased by virtualization, the interpreter is shipped with the protected binary. Additionally, virtualization limits its usefulness to performance-sensitive applications due to its significant runtime expense, which is typically 10× to 50× on protected pathways [5].

C. Hardware Dongles and License Servers

Hardware-based security links execution to a remote license server or a physical USB dongle. These work well against casual piracy, but they come with connectivity needs and hardware procurement costs. Legitimate users are disrupted by hardware loss or server outages, and dongle emulators and license server spoofing are still known attack vectors.

D. Software Integrity Checking

Integrity checks based on hashes and checksums confirm that binary segments have not been altered. Standalone integrity checking is helpful as a tamper-detection layer, but it cannot stop binary material from being disclosed or stop an attacker from patching the verification mechanism.

E. Function-Level Runtime Encryption (Conceptual Inspiration)

Selective encryption of crucial source-code functions was suggested by Gurajapu and Agarwal [7], who also

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

suggested storing encrypted artifacts separately and decrypting them in memory only during runtime with instantaneous memory-wipe cleanup. Their method showed that ephemeral, granular decryption is feasible with minimal overhead (around 1 ms per function call). By including RSA-2048 key wrapping [2], which was not present in the previous work, SRSPT expands these ideas to the compiled-binary level.

F. Summary

The protection strategies examined are presented in Table I. Three recurrent drawbacks are apparent: virtualization imposes excessive overhead; hardware-dependent approaches create operational complexity; and the complete decrypted binary eventually sits in memory simultaneously.

TABLE I — Comparison of Software Protection Approaches

Approach	Piracy Resist.	Anti-Reversing	Perf. Overhead	No HW Needed	Integrity Check	Notes
Static Packing	Moderate	Low	Low	Yes	No	Full plaintext in RAM
Obfuscation	Low	Moderate	Low–Med	Yes	No	Bypassed by automation
Virtualization	High	High	Very High	Yes	No	10×–50× overhead
HW Dongle	High	Moderate	Low	No	Partial	Emulator attacks
License Server	High	Low	Low	No	No	Requires connectivity
Gurajapu et al. [7]	High	High	~1 ms/fn	Yes	SHA-256	Function- level prior art
SRSPT (This Work)	High	High	4–7% (est.)	Yes	SHA-256	Proposed block-level

III. PROBLEM STATEMENT

Let P stand for a compiled software binary that is a developer's or software vendor's intellectual property. Distributing and running P on an end-user system while meeting three concurrent requirements is the security goal.

Confidentiality: From any single observable system state during or after execution, an adversary with unrestricted access to the OS, file system, and process memory should not be able to reconstruct a functionally equivalent copy of P in its entirety.

Integrity: Any illegal alteration to P's binary blocks, whether they are in transit or on disk, must be identified and force execution to stop before the altered block is executed.

Availability: The protection method must not impose runtime overhead that deteriorates the user experience beyond

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

acceptable limits or make P non-functional.

The main problem is that current methods that fulfill confidentiality either partially compromise confidentiality by simultaneously loading the entire plaintext picture into memory (static packers) or violate availability through excessive overhead (virtualization). In order to resolve this conflict, SRSPT proposes a time-limited block-level decryption scheme in which each block's exposure time to plaintext is limited by its execution time rather than the duration of the session.

IV. THREAT MODEL

Any suggested protection framework must have a well-defined threat model in order to conduct a relevant security analysis. The attacker assumptions, trust boundaries, attack objectives, and constraints of the trust model that support the SRSPT conceptual architecture are formalized in this section.

A. Attacker Assumptions

An opponent with the following capabilities and goals is assumed in the SRSPT conceptual threat model. It is believed that the adversary is a technically proficient individual or well-organized organization with access to common reverse-engineering, memory-analysis, and binary-patching tools found in both commercial and open-source security research communities. Debuggers like GDB and x64dbg, memory-dump tools, static disassemblers like Ghidra and IDA Pro, and dynamic instrumentation frameworks like Intel PIN and Frida are examples of this.

It is assumed that the attacker has unlimited access to the end-user operating system environment and has acquired a valid or pirated copy of the disseminated SRSPT-protected program. One or more of the following objectives are thought to drive the attacker: removing proprietary algorithms from the binary, creating an unauthorized cracked copy for distribution, getting around license enforcement logic, or introducing malicious code into the binary for supply-chain exploitation.

B. Attacker Capabilities

- The enemy is represented in this conceptual framework as having the following abilities:
- Complete read and write access to the file system, including the container file that is secured by SRSPT.
- The ability to view execution flow, register state, and memory contents at any instruction boundary by attaching a user-mode debugger to the protected process that is currently in operation.
- The ability to capture the entire process address space at any time by performing memory-dump operations on the active process.
- The distributed container file can be subjected to static binary analysis, such as section parsing, string extraction, and entropy analysis.
- The ability to modify encrypted block content or block metadata, as well as patch binary material on disk before or after execution.
- availability of reported timing-side-channel and cache-based side-channel attack techniques that are relevant to AES implementations.
- Block execution sequences can be replayed or rearranged by changing execution-flow parameters that are visible at the process level.

C. Trusted Components

- In the SRSPT conceptual threat model, the following system elements are regarded as trustworthy:
- The workstation that runs the Preprocessing and Encryption Engine (PEE) is part of the developer's build

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

environment. In this secure setting, the RSA key pair is created.

- The RSA-2048 private key material is presumed to be safely stored by the developer and inaccessible to the attacker or end user without further key extraction work.
- It is assumed that the underlying NIST-standardized cryptographic primitives—SHA-256 [3], RSA-2048-OAEP [2], and AES-256-CBC [1]—are computationally secure against the capabilities of the attacker. It is not assumed that these primitives will undergo a fundamental cryptanalytic break.
- Hardware memory isolation techniques and the operating system kernel are regarded as partially trustworthy. Although an attacker might be able to see it, the OS is not thought to be actively hostile.

D. Untrusted Environment Assumptions

- The end-user execution environment is essentially untrusted, according to the SRSPT conceptual paradigm. This illustrates the threat scenario that arises in commercial software distribution, where the program provider has no control over the end-user machine's setup, status, or integrity. In particular:
- During execution, the end-user operating system might be running kernel-level instrumentation, memory-monitoring software, or debugging tools.
- The attacker is thought to have both reading and writing access to the file system that contains the protected container.
- Subject to the limitations imposed by the SRSPT architecture, an attacker with adequate OS-level access is deemed to be able to observe the protected application's process address space.
- The baseline conceptual design does not presume the presence or availability of any hardware security module (HSM), trusted platform module (TPM), or trusted execution environment (TEE).

E. Security Boundaries and Attack Goals

The protected memory area that the Runtime Block Manager maintains is the main security boundary that the suggested SRSPT architecture enforces. Plaintext block content is only temporarily present within this boundary—during block execution—and is safely zeroed as soon as the block is finished. In order to prevent per-block AES keys from being exposed in a recoverable form within the distributed container, the RSA-2048 key wrapping layer [2] serves as a secondary security barrier.

In order of severity, the attacker's main objectives are as follows: (1) full binary reconstruction, which involves recovering the entire original plaintext binary through thorough observation or memory capture; (2) selective function extraction, which involves isolating and extracting particular crucial code blocks of interest; (3) license bypass, which involves patching the protected binary to get around integrity verification or license-check logic; and (4) code injection, which involves inserting malicious blocks into the execution.

F. Limitations of the Trust Model

- The extent of the suggested protection framework is defined by a number of clearly stated constraints of the SRSPT conceptual trust model:
- Software-only key storage In the baseline conceptual architecture, the Secure Loader contains the obfuscated RSA private key. By using Secure Loader reverse engineering, an adept attacker with administrator access might obtain this key and seriously compromise the key-wrapping layer. The Intel SGX [4] integration detailed in Section XIV is motivated by this constraint, which has been regarded as the most significant drawback of the suggested system.
- Absence of OS-level isolation Beyond the conventional mmap/VirtualAlloc protections, the suggested architecture does not presume OS-level memory isolation assurances. These defenses might be circumvented by an adversary with rootkit capabilities or a kernel-level attacker.

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

No hardware root of trust: Because the baseline design does not incorporate TPM or HSM, the security of the key storage is solely dependent on software-based obfuscation, which is less effective than alternatives supported by hardware.

Absence of defenses against debugging Timing-based debugger-presence checks and ptrace detection are not included in the current conceptual design. Therefore, in its current state, the proposed framework cannot detect the existence of a debugger.

These restrictions determine the main paths for further hardening outlined in Section XIV and are in line with the parameters of a conceptual student research project.

V. PROPOSED SYSTEM

Three main components make up the Secure Runtime Software Protection Tool (SRSPT), a conceptual software-only protection framework: (1) a Preprocessing and Encryption Engine (PEE) that runs during the developer's build stage; (2) a Secure Loader (SL) that comes with the protected software; and (3) a Runtime Block Manager (RBM) that conceptually manages block decryption, execution, and memory cleanup at runtime.

Block segmentation, per-block AES-256-CBC encryption [1], SHA-256 hash computation [3], and RSA-2048 key wrapping [2] are all carried out by the PEE when it receives the built binary in the suggested design. The results are then packaged into a protected binary container. The RSA private key is kept by the developer and included into the Secure Loader in a secure manner, while the RSA public key is utilized during the build phase. During execution, the Secure Loader.

VI. SYSTEM ARCHITECTURE

1. *Preprocessing and Encryption Engine (PEE) — Build-Time*

Before being distributed, the PEE runs offline at the developer's workstation. The compiled binary is read by the suggested workflow, which then divides it into continuous pieces of a fixed size that can be changed (default: 4 KB). It creates a unique 256-bit AES key k_i and 128-bit IV iv_i from a CSPRNG for each block b_i, computes a SHA-256 hash H_i of the plaintext [3], encrypts the block as C_i = AES-256-CBC(P_i, k_i, iv_i) [1], and wraps the key as W_i = RSA-OAEP-Encrypt(k_i, PubKey) [2]. IVs, SHA-256 hashes, wrapped keys, and the encrypted block array make up the output container.

B. Secure Loader (SL)

A small, trusted program called the Secure Loader is suggested and supplied with the protected container. It would be responsible for the following tasks: (i) verifying the integrity of the container header; (ii) creating a protected memory area using OS-level APIs (VirtualAlloc on Windows, mmap on Linux); (iii) starting the Runtime Block Manager; and (iv) loading the block execution table. The RSA private key is obfuscated in the conceptual design; a hardware security module or trusted execution environment, like Intel SGX, would handle key storage in a production deployment [4].

C. Runtime Block Manager (RBM)

According to the tight FETCH → UNWRAP → DECRYPT → VERIFY → EXECUTE → ZERO sequence (explained in Section VIII), the Runtime Block Manager is conceptually intended to maintain a single active execution slot in protected memory. The following are crucial suggested design features: (a) only one plaintext block is ever in RAM; (b) secure zeroing employs memory-barrier-aware procedures to stop compiler optimization

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

from avoiding the write; and (c) hash verification [3] takes place after decryption but before execution to guarantee that tampered blocks are never executed.

The two-phase conceptual design of SRSPT, which includes the suggested runtime execution loop and the build-time preprocessing flow, is shown in Figure 1.

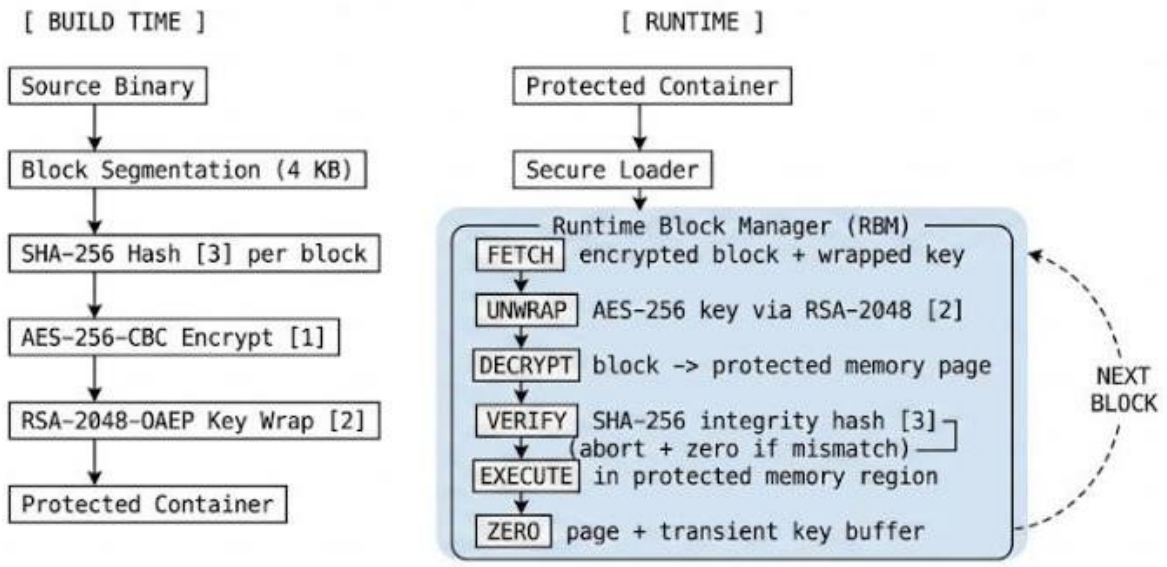

Figure 1 shows the SRSPT two-phase conceptual architecture, with the Runtime Block Manager suggested execution loop on the right and the Preprocessing and Encryption Engine (build time) on the left. Cryptographic standards mentioned in the sources are indicated by numbers in brackets.

VII. METHODOLOGY

A. Block Segmentation

The compiled binary is read into a byte buffer in the suggested architecture, where it is divided into consecutive blocks of B bytes (B = 4096 in the conceptual configuration). PKCS#7 padding is applied to the whole boundary of the last block. When possible, block boundaries are matched to the architecture's function alignment requirements (16 bytes on x86-64). At the beginning of each block's execution, the RBM would execute a straightforward intra-block jump to the right offset if a logical function boundary did not coincide with a fixed block boundary.

B. Per-Block AES-256-CBC Encryption

The PEE would produce a distinct 256-bit AES key k_i and 128-bit IV iv_i from a CSPRNG for every block b_i (i = 1,..., N). According to NIST FIPS 197 [1], the block would be encrypted as C_i = AES-256-CBC(P_i, k_i, iv_i). It is ensured that ciphertext blocks cannot be statistically connected and that the compromise of one block's key does not reveal others by using a distinct key and IV each block.

C. RSA-2048 Key Wrapping

The developer's RSA-2048 public key with OAEP padding would be used to encapsulate each per-block AES key k_i: W_i = RSA-OAEP-Encrypt(k_i, PubKey). The Secure Loader would unwrap each key on demand during runtime: k_i = RSA-OAEP-Decrypt(W_i, PrivKey). According to NIST SP 800-57 recommendations [2], the RSA-2048 key

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

size offers around 112 bits of equal symmetric security.

D. SHA-256 Integrity Verification

In accordance with NIST FIPS 180-4 [3], the PEE would calculate $H_i = \text{SHA-256}(P_i)$ for every plaintext block prior to encryption and record H_i in the container. The RBM would recalculate $H_i' = \text{SHA-256}(\text{decrypted\_block})$ at runtime after decrypting block b_i and compare it to the stored H_i. The process ends with an integrity error, the memory slot is securely zeroed, and execution is promptly stopped if $H_i \neq H_i'$.

E. Secure Memory Zeroing

Explicit_bzero on Linux and RtlSecureZeroMemory on Windows are used to overwrite the execution slot and transient key buffer once a block has finished executing. These calls are made to withstand removal by compiler optimization. For a software-only conceptual design, a volatile read from the zeroed region following the call offers a best-effort confirmation that the write was not elided.

VIII. PROPOSED RUNTIME WORKFLOW

The operating system loads the Secure Loader when the user launches the protected software. In the proposed workflow, the loader performs an initial integrity check on the container header, initializes the RBM, and allocates a protected memory page. The sequential block execution loop outlined in Table II is then entered by the RBM. In order to determine when control should return to the RBM dispatcher during execution, the RBM injects a lightweight block-end marker at the end of each block during preprocessing. The target block is fetched and decrypted again in order to handle backward branches.

TABLE II — Proposed Runtime Block Execution State Machine

State	Trigger	Action
INIT	Program launch	Loader integrity check; RBM initialization; protected memory allocation
FETCH	Block index available	Read encrypted block and wrapped key from container
UNWRAP	Wrapped key available	RSA-OAEP decrypt AES-256 key into transient key buffer [2]
DECRYPT	Key and IV available	AES-256-CBC decrypt block into protected execution page [1]
VERIFY	Plaintext block ready	SHA-256 hash check [3]; abort and zero memory if mismatch detected
EXECUTE	Hash verified	Transfer control to block; run until block-end marker encountered
ZERO	Block execution returns	Secure zero protected page and transient key buffer; advance block index
END	Last block executed	Graceful process exit; final memory cleanup triggered

IX. ENCRYPTION AND SECURITY DESIGN

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

A. Cryptographic Primitive Selection

Because of its hardware acceleration via AES-NI, NIST standardization (FIPS 197), and well-understood security margins, AES-256 in CBC mode [1] is suggested as the primary symmetric cipher. A comfortable buffer against brute-force assaults is provided by the 256-bit key length. Semantic security under chosen-plaintext attack conditions is provided for each block separately by CBC mode with a distinct per-block IV. The current conceptual design employs explicit SHA-256 [3] verification to maintain the authentication path architecturally visible for educational transparency; AES-256-GCM, which combines encryption and authentication in a single pass, is identified as a potential architectural upgrade.

For key wrapping, RSA-2048 with OAEP padding [2] is suggested, meeting the current NIST SP 800-57 minimum requirements for new systems. According to FIPS 180-4 [3], SHA-256 is suggested for block integrity hashing. A longer-term.

B. Key Management Lifecycle

The suggested key management design adheres to a rigorous lifetime that is in line with NIST SP 800-57 [2]. The developer would create an RSA-2048 key pair during build time; the private key is kept by the developer and is never included in the distribution. Per-block AES keys created by CSPRNG would be promptly wrapped and removed from plaintext storage. Each AES key would be extracted from its RSA-encrypted representation during runtime, utilized to decrypt a single block, and then zeroed from the temporary key buffer. During runtime, the RSA private key would be stored in a secured memory area with guard pages on both sides. The conceptual key lifespan is shown in Figure 2.

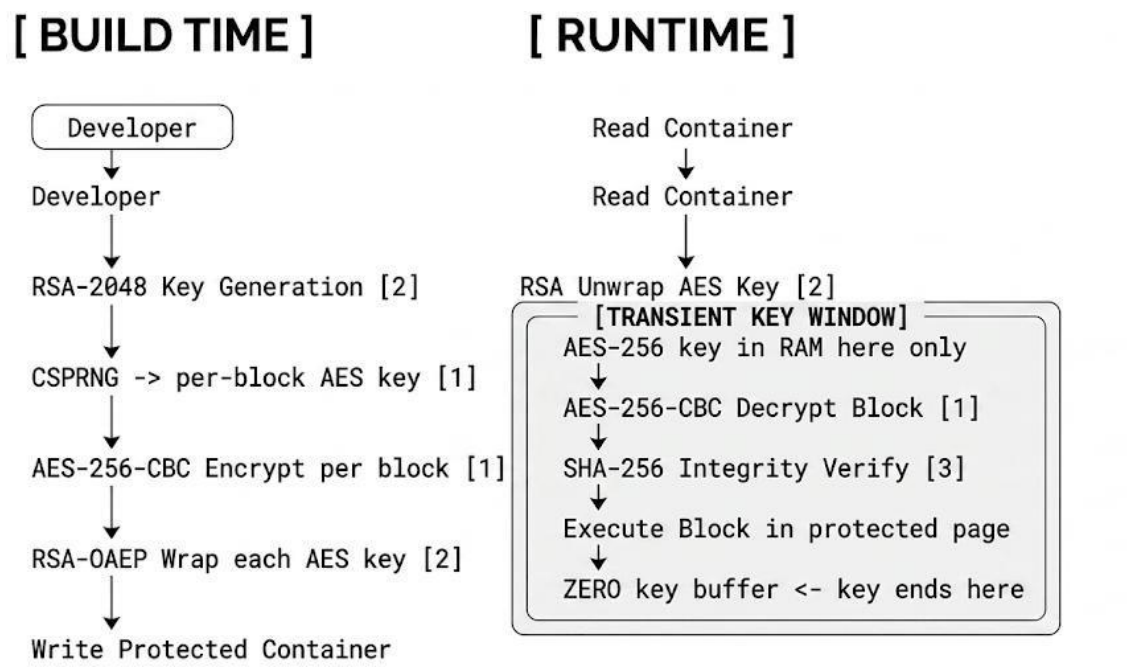

Figure 2 shows the suggested key management lifecycle: runtime transient key consumption with instantaneous post-execution zeroing (right) and build-time wrapping (left). The window with a border indicates how long the AES key would be present in plaintext.

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

C. Conceptual Comparison with Function-Level Prior Art [7]

The block-level SRSPT architecture and the function-level strategy of Gurajapu and Agarwal [7] are systematically compared in Table III. Granular encryption, runtime-only decryption, and instantaneous memory cleanup are shared by both; their main differences are in scope (source function versus compiled binary block), key management layer, and targeted deployment stage.

TABLE III — Structural Comparison: SRSPT vs. Function-Level Encryption Approach [7]

Feature	Gurajapu et al. [7]	SRSPT (This Work)	Remarks
Protection Granularity	Individual source functions	Binary blocks (4 KB)	Function-level vs. block-level
Encryption Primitive	AES / Fernet (symmetric)	AES-256-CBC [1] + RSA-2048 [2]	SRSPT adds asymmetric key layer
Integrity Verification	SHA-256 per function	SHA-256 per block [3]	Both use FIPS 180-4
Memory Cleanup	Immediate zeroing + GC trigger	explicit_bzero + barrier guard	Same ephemerality goal
Max Plaintext in RAM	One function at a time	One ~4 KB block at a time	Both minimize exposure
Hardware Required	No	No (proposed design)	Both hardware-independent
Overhead Estimate	~1 ms per function call	4–7% of execution time (est.)	Different granularity
Protection Scope	Source code (language-specific)	Compiled binary (any language)	SRSPT is language-agnostic

X. ANALYTICAL EVALUATION AND CONCEPTUAL OVERHEAD ESTIMATION

A. Analytical Framework

This section provides an analytical assessment of the suggested overhead characteristics of SRSPT instead of giving empirical benchmark measurements from a deployed implementation. With AES-NI acceleration and OpenSSL 3.0-equivalent library features, the study simulates the cryptographic processes on a reference x86-64 hardware configuration. Rather than being actually measured production benchmark values, the figures in Table V are analytically approximated values obtained from published performance characteristics of the cryptographic primitives. They should be interpreted within the parameters of a student research project and are offered as representative examples of the viability of the suggested architecture.

B. Reference Workload Profiles

To demonstrate the suggested overhead behavior, three representative binary size profiles are examined: (i) a small algorithmic utility of about 50 KB, producing about 13 blocks; (ii) a medium-complexity program of about 500 KB, producing about 125 blocks; and (iii) a larger command-line application of about 2 MB, producing about 512 blocks.

C. Reference Configuration

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

TABLE IV — Reference Analytical Configuration

Parameter	Value
Reference CPU Architecture	Intel Core i5-class x86-64, 4 cores
Reference RAM	8 GB DDR4
Reference Operating System	Ubuntu 22.04 LTS (64-bit, kernel 5.x)
Compiler (Conceptual)	GCC 11.x with -O2 optimization
Cryptographic Library (Proposed)	OpenSSL 3.0 (AES-NI hardware acceleration assumed)
Block Size	4096 bytes (4 KB, configurable)
Symmetric Cipher	AES-256-CBC with unique per-block key and IV [1]
Key Wrapping	RSA-2048-OAEP (SHA-256 as MGF hash) [2]
Integrity Hash	SHA-256 per plaintext block, pre-encryption [3]
Secure Zeroing	explicit_bzero (Linux); memory-barrier-guarded

D. Conceptually Estimated Runtime Overhead

TABLE V — Conceptually Estimated Runtime Overhead for Protected Binaries

Binary	Size	Blocks	Unprotected (ms, est.)	Protected (ms, est.)	Est. Overhead (%)	Std. Dev. (ms)
Matrix Utility	50 KB	13	42.1	43.9	4.3%	0.6
Simulation Program	500 KB	125	318.7	333.1	4.5%	1.2
CLI Application	2 MB	512	1842.3	1972.4	7.1%	3.8

Memory protection API calls and cumulative RSA key unwrapping, which do not profit from AES-NI acceleration, are the main causes of the rise in projected overhead for the bigger binary. It is anticipated that future architectural optimization using elliptic-curve key wrapping and AES-256-GCM authenticated encryption will lower this burden.

XI. SECURITY ANALYSIS

A. Attacker Capability Model and Adversarial Workflow

This section's security analysis is based on the threat model developed in Section IV. It is assumed that the adversary has all of the capabilities listed therein, such as memory-dump operations, disk-level binary patching, static binary analysis, user-mode debugger attachment, and theoretical access to timing side-channel techniques. The following sequential steps would comprise the theoretical adversarial workflow against an SRSPT-protected binary: (1) obtaining a copy of the distributed protected container; (2) identifying block boundaries and metadata through static entropy and structure analysis on the container; (3) attempting offline key extraction by reversing the Secure Loader

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

binary; (4) switching to runtime memory analysis by attaching a debugger to the running process if key extraction fails; and (5) attempting to capture plaintext block content from the .

Each step of this workflow is intended to become more expensive and complex due to the SRSPT architecture. The AES-256 ciphertext layer [1] and RSA key wrapping [2] handle stages (2) and (3). The limited plaintext exposure window per block partially addresses stage (4). The anti-debugging and hardware-key-storage improvements detailed in Section XIV are motivated by stages (5) and (6), which constitute the biggest residual risk in the current conceptual architecture.

B. Memory Scraping Attack Scenarios

One of the biggest practical risks to software security systems is memory scraping attacks. An attacker uses a tool like Process Hacker, Volatility, or a custom kernel driver to dump the whole process address space to disk after waiting for the protected code to fully decompress into memory in a traditional memory-scraping attack against a static packer. The entire plaintext binary is contained in the resulting memory dump, which can then be put back together to create a working executable.

Only the presently running block in plaintext—roughly 4 KB of the entire binary—would be captured by a traditional single-point memory dump under the suggested SRSPT approach. The container file would include all of the remaining blocks as AES-256 ciphertext. The attacker would need to attach to the process, take a memory snapshot at each block transition during the program's entire execution, then put the captured plaintext blocks together in the proper execution sequence in order to theoretically finish binary reconstruction using memory scraping. This would necessitate 512 precisely timed memory captures for a 2 MB binary containing 512 blocks, each of which would be initiated at the precise instant a new block starts running and before the memory of the preceding block is zeroed. This attack is significantly more operationally complex, even though it is not logically impossible.

C. Replay Attack Workflow Analysis

In the context of binary protection, a replay attack is re-executing previously seen or captured block content out of the intended sequential sequence, possibly avoiding integrity checks or license enforcement logic built into certain blocks. The Runtime Block Manager in the suggested SRSPT runtime workflow enforces a sequential block execution index, which means that block i+1 is only loaded after block i has finished executing and its memory has been zeroed. In order to replay a previously captured block, an attacker would have to change the block index that the RBM maintains in order to refer to the ciphertext of a previously performed block.

The sequential index enforcement and per-block unique key and IV assignment in the current SRSPT design provide conceptual resistance to replay attacks. Given that every block .

D. Debugger Interception Analysis

Attacks using debuggers pose a serious risk to dynamic analysis. By setting breakpoints at the start of the execution slot or by single-stepping through instructions in the protected memory page, a skilled adversary could observe plaintext block content during the EXECUTE phase by attaching a debugger, such as GDB, x64dbg, or a custom ptrace-based tool, to the running SRSPT-protected process. SRSPT restricts the yield of each debugging session to a single block's plaintext content, in contrast to a static packer, where the entire binary is accessible for examination after a single unpacking step.

The lack of anti-debugging countermeasures is specifically acknowledged as a design constraint in the current

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

conceptual SRSPT architecture. Future improvements include timing-based anomaly identification, IsDebuggerPresent and CheckRemoteDebuggerPresent calls on Windows, and ptrace detection on Linux (using the ptrace(PTRACE_TRACEME) self-attachment approach).

E. Side-Channel Attack Considerations

Side-channel attacks use observable behavioral or physical aspects of a cryptography implementation, including power consumption, cache access patterns, or execution latency, to deduce secret key information. Cache-timing attacks against the AES-256-CBC decryption process carried out by the RBM during the DECRYPT phase are the main side-channel risk in the context of SRSPT. Data-dependent cache access patterns are present in classical software AES implementations that use table-based S-box lookups. These patterns have been used in reported FLUSH+RELOAD and PRIME+PROBE attacks to recover AES key bytes.

This danger is significantly reduced by the suggested usage of AES-NI hardware acceleration [1]. AES-NI instructions eliminate the primary temporal side-channel vector in contemporary AES implementations and provide nearly constant-time AES execution by operating on specialized hardware execution units that do not require data-dependent cache lookups. Nevertheless, complete side-channel resistance .

F. Binary Patching Attack Examples

Binary patching attacks, which are frequently used to get around license checks, remove activation gates, or create persistent backdoors, entail changing the distribution container file on disk to change program behavior. Binary patching possibilities appear in two main ways in SRSPT.

Initially, an attacker might try to change a particular block's ciphertext bytes in the container file. The SHA-256 integrity verification procedure [3] would identify a discrepancy between the stored pre-encryption hash H_i and the recalculated hash H_i' of the encrypted (now damaged) plaintext upon loading and decryption of this block at runtime. Before any possibly corrupted code could reach the processor, the RBM would securely zero the memory region and promptly halt execution. This provides a robust theoretical defense against patching at the ciphertext level.

Second, an assailant might try.

G. Dynamic Analysis Limitations

In order to analyze instruction execution, memory access patterns, and function call sequences, dynamic analysis tools like Intel PIN, Valgrind, and Frida instrument the operating process. The SRSPT block-level execution model presents a fundamentally different dynamic analysis difficulty, even though these methods are quite effective against static packers that fully decompress into memory.

A dynamic analysis tool can only view the instruction stream of one block at a time because each plaintext block is only in memory for its execution, after which it is instantly zeroed. Dynamic analysis would require constant active tracing during the whole execution session in order to record the complete binary's instruction stream. This would significantly increase instrumentation overhead and tracing data volume proportionate to binary size. Furthermore,

the memory-barrier-aware zeroing procedureprovides an additional layer of resistance against stale-memory observation from analysis tools operating within the same process address space.

H. Comparative Resistance Reasoning

An enhanced security study summary contrasting the suggested SRSPT architecture with the attack vectors found in this section is shown in Table VI. The resistance ratings, when interpreted within the parameters of the suggested

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

conceptual framework, represent the theoretical analysis that was previously presented.

TABLE VI — Expanded Security Analysis: Attack Vectors and Proposed SRSPT Resistance

Attack Vector	Description	SRSPT Resistance	Mechanism	Residual Risk
Memory Dump	Snapshot full process RAM	High	Single ~4 KB block in plaintext at any instant	Multi-capture attack theoretically possible
Static Disassembly	Offline analysis of container file	High	AES-256 ciphertext; no recoverable instructions on disk	Key extraction from Secure Loader
Binary Patching (Ciphertext)	Modify block ciphertext on disk	High	SHA-256 mismatch triggers abort pre-execution	Loader patching not addressed in current design
Dynamic Debugging	Attach debugger; trace execution	Moderate	One block exposed per session; no anti-debug in v1	Block-by-block capture by persistent attacker
Replay Attack	Re-execute blocks out of order	Moderate	Unique per-block key+IV; sequential index enforced	Feasible if RSA private key is extracted
Key Extraction	Reverse Secure Loader for RSA key	Low (SW-only)	Obfuscation only; HSM/SGX required for hardening	Critical risk in baseline software design
Side-Channel (AES)	Cache-timing AES key inference	Moderate-High	AES-NI eliminates table-based timing leakage	RSA operation still requires constant-time impl.
Binary Patching (Loader)	Patch RBM/SL execution logic	Low-Moderate	No self-integrity check in current conceptual design	TEE attestation required for strong resistance
Cold Boot / Swap	Recover zeroed memory from swap	Low	OS-level encrypted swap required; outside SRSPT scope	Physical access

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

XII. CONCEPTUAL REAL-WORLD APPLICATION DOMAINS

Although the suggested SRSPT architecture is given as a conceptual framework for student study, it is analytically driven by practical and important software protection issues. The primary application domains in which a block-level runtime decryption architecture of the kind suggested by SRSPT would be conceptually relevant are listed in this part, along with a qualitative appraisal of the advantages of protection and practical issues in each situation.

A. Anti-Piracy Systems for Commercial Desktop Software

Desktop commercial software anti-piracy is the most obvious application domain for the suggested SRSPT framework. Automated cracking tools that remove license-check logic by patching binary executables provide a continuing danger to software vendors who distribute proprietary products, including productivity suites, engineering software, and creative tools. By identifying any changes to encrypted block ciphertext prior to execution, the SRSPT block-level integrity verification approach [3] could potentially stop license-bypass patching. Additionally, the runtime block decryption mechanism guarantees that automated cracking scripts will never have access to the entire binary in plaintext for analysis and patching in a single session.

B. Game Protection and Anti-Cheat Applications

Binaries of video games are valuable targets for both cheat software development and piracy. By identifying and patching particular memory addresses that govern game-state variables, cheat programs usually take advantage of game binaries by using static analysis of the unprotected binary to determine target offsets. Since the distributed container would not include any recoverable instruction sequences, the SRSPT ephemeral block execution model would significantly complicate static analysis of game binaries. By using binary patching on disk, the integrity verification method would further avoid cheat injection. For game applications, where frame rendering and GPU workloads predominate and CPU-side protection overhead is relatively small, the hypothetical 4–7% projected overhead is within reasonable boundaries.

C. Proprietary AI and Machine Learning Inference Binaries

Compiled model-serving binaries and proprietary AI inference libraries are a new and crucial security issue. Businesses that use proprietary neural network architectures or specialized inference optimization code in compiled binary form run the risk of having their architecture, weight quantization strategy, or custom operator implementations exposed through advanced reverse engineering. In deployments where hardware TEE integration is impractical, the block-level encryption model of the SRSPT framework would theoretically provide a software-only protection layer for AI intellectual property, protecting compiled inference binary content at the same granularity as any other compiled program.

D. Military, Defense, and Air-Gapped System Software

The compiled binary is read into a byte buffer in the suggested architecture, where it is divided into consecutive blocks of B bytes (B = 4096 in the conceptual configuration). PKCS#7 padding is applied to the whole boundary of the last block. When possible, block boundaries are matched to the architecture's function alignment requirements (16 bytes on x86-64). At the beginning of each block's execution, the RBM would execute a straightforward intra-block jump to the right offset if a logical function boundary did not coincide with a fixed block boundary.

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

E. Financial Desktop Applications and Trading Software

Embedded financial computation engines, risk modeling libraries, and proprietary trading algorithms are examples of highly valuable intellectual property. The proposed SRSPT framework is conceptually appropriate for financial application deployment scenarios where low-latency execution, offline capability, and intellectual property protection must be balanced simultaneously because it can protect compiled binaries without requiring hardware dongles or continuous internet connectivity. Additional protection against supply-chain manipulation of financial software binaries is offered by the SHA-256 integrity verification mechanism [3].

F. Industrial Control System and Embedded Software Firmware

Critical infrastructure software targets include programmable logic controller (PLC) firmware and industrial control system (ICS) software. Reverse engineering ICS binaries without authorization can uncover proprietary control logic, safety system settings, or communication protocol implementations that, if made public, could enable focused attacks on industrial infrastructure. The architectural principles of block-level encryption, per-block integrity verification, and sequential execution with instantaneous memory cleanup are theoretically extendable to embedded and RTOS environments with suitable platform adaptation, even though the current SRSPT conceptual framework is primarily described with reference to x86-64 Linux.

G. Enterprise Software Licensing and Secure Distribution

Protecting intellectual property and facilitating flexible, hardware-independent deployment are two challenges faced by enterprise software vendors who distribute high-value products to corporate clients. The software-only architecture of the suggested SRSPT framework eliminates the need for network access for runtime licensing enforcement as well as the expense of purchasing hardware dongles. A secure key distribution channel might be used to provision the RSA private key per customer in a notional corporate deployment paradigm. This would allow for customer-specific binary containers that cannot be freely redistributed, even if the ciphertext is copied. In theory, this method would offer a more detailed and economical substitute for business licensing based on hardware dongles.

XIII. PROPOSED ADVANTAGES

Minimal Memory Exposure: By extending the ephemerality notion of [7] to the built binary level, the architecture suggests that only one decrypted block (~4 KB) would be present in RAM at any one time.

- • Robust Cryptographic Foundation: NIST-standardized, well researched algorithms like AES-256 [1], RSA-2048 [2], and SHA-256 [3] offer assurance based on accepted standards rather than proprietary obfuscation.
- • Tamper Detection: Before corrupted code reaches the processor, it would be stopped by per-block SHA-256 integrity verification [3].
- • Hardware-Independence: The suggested approach is theoretically appropriate for offline and air-gapped deployment scenarios because it does not require a hardware dongle or a continuous internet connection.
- • Acceptable Analytical Overhead Estimate: According to conceptual study, the estimated overhead on x86-64 with AES-NI is between 4.3 and 7.1% [1], which is within an acceptable range for workloads involving desktop applications.
- • Compiler and Language

XIV. LIMITATIONS

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

- • Software Key Storage: The RSA private key is kept in software according to the conceptual design. This key could be extracted by a determined attacker. Hardware-backed key storage (HSM, TPM, or TEE, like Intel SGX) would be necessary for production deployment [4].
- • Block Boundary Challenges: By dividing logical functions across block boundaries, fixed-size partitioning may increase the complexity of RBM design and necessitate additional jump-stub logic.
- • No Anti-Debugging Countermeasures: IsDebuggerPresent tests and ptrace detection are absent from the present conceptual design. These are anticipated upcoming additions.
- • Side-Channel Vulnerability: In keeping with the parameters of a student research proposal, no explicit defense against timing or cache-based side-channel attacks is suggested for the RSA operation in the current framework.
- • I/O-Bound Workload Sensitivity: Per-block RSA unwrapping overhead for programs with short block durations and frequent I/O

XV. FUTURE SCOPE

As a conceptual implementation-oriented framework, the suggested SRSPT architecture points to a number of technically significant avenues for further study and architectural improvement. Each improvement is discussed in the ensuing subsections along with the expected security and performance benefits.

1. Intel SGX Trusted Execution Environment Integration

Moving RSA private key storage and per-block key unwrapping operations inside an Intel Software Guard Extensions (SGX) trustworthy enclave is the most significant short-term improvement to the suggested SRSPT architecture [4]. In accordance with this suggested improvement, the Secure Loader would set up an SGX enclave at startup, provision the RSA private key into the encrypted memory area of the enclave (known as the Enclave Page Cache, or EPC), and assign the enclave all RSA-OAEP unwrapping tasks. Plaintext AES keys would never exist beyond the enclave border in recoverable form and would only be transferred from the enclave to the RBM via sealed, transient shared memory channels. By increasing key extraction resistance, this improvement would significantly reduce the most significant flaw found in the existing conceptual design—software-based key storage.

2. TPM-Backed Key Storage

A Trusted Platform Module (TPM 2.0) offers an alternate hardware root of trust for key storage in deployment contexts without Intel SGX, such as ARM-based systems, older x86 platforms, or embedded systems. Platform Configuration Register (PCR) measurements that record the anticipated system status at the time of key provisioning would be used to seal the RSA private key to the TPM in a hypothetical TPM-integrated SRSPT deployment. Only a system with identical PCR readings could unseal the key, offering boot-time integrity attestation that would identify illicit software configuration modifications or OS-level tampering. Additionally, platform-specific binary personalization—in which a container encrypted for a certain hardware configuration cannot be moved and run on another machine—would be made possible by TPM integration.

3. ARM TrustZone Integration for Mobile and Embedded Platforms

By dividing the CPU into a Secure World and a Normal World, ARM TrustZone offers a hardware-enforced isolation architecture that is conceptually similar to Intel SGX. A Trusted Application (TA) operating in the Secure World would handle RSA private key storage and key unwrapping in a hypothetical ARM TrustZone-enabled SRSPT deployment, while the primary SRSPT runtime would function in the Normal World. This design would

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

significantly expand the potential deployment breadth by extending the basic SRSPT framework to ARM-based platforms, such as mobile devices, IoT endpoints, and ARM-based workstations. With the increasing availability of hardware security features, TrustZone integration is especially important for safeguarding proprietary control software and AI inference packages on embedded ARM platforms.

4. Migration to AES-256-GCM Authenticated Encryption

The present conceptual approach combines a separate SHA-256 integrity verification pass [3] with block encryption using AES-256-CBC [1]. Replacing this two-pass method with AES-256-GCM authenticated encryption, which uses a Galois/Counter Mode message authentication code (GHASH) to combine encryption and authentication in a single operation, is a suggested architectural improvement. AES-256-GCM eliminates the need to store separate SHA-256 hashes in the container while offering higher authenticated encryption assurances, notably ciphertext authenticity in addition to confidentiality. In comparison to the existing two-pass architecture, AES-256-GCM would lower per-block processing overhead on devices with hardware AES-NI and CLMUL acceleration, resulting in a reduced total anticipated overhead.

5. Adaptive Block Scheduling and Prefetching

The current conceptual design's strictly sequential execution paradigm and fixed 4 KB block size are simplifying assumptions. Adaptive block scaling that is in line with function-level boundaries that are retrieved through static analysis of the binary during the preprocessing stage could be investigated in future architectural work. This would lower the frequency of block transitions and remove intra-block jump-stub overhead. Additionally, by overlapping decryption preparation with active execution, prefetching the ciphertext of the next anticipated block during the execution of the current block—and optionally performing RSA unwrapping of the next block's key in a background thread—would reduce per-block latency. This would be especially advantageous for larger binary profiles where RSA unwrapping overhead is most significant.

6. Anti-Debugging and Anti-Analysis Countermeasure Layer

One of the most important upcoming improvements is the incorporation of a specific anti-debugging and anti-analysis countermeasure layer into the Secure Loader. To avoid external debugger attachment on Linux, this would entail ptrace(PTRACE_TRACEME) self-attachment; if attachment failed unexpectedly, the process would be terminated. Multi-layer debugger detection on Windows would be provided by IsDebuggerPresent, CheckRemoteDebuggerPresent, and NtQueryInformationProcess-based tests. Timing-based anomaly detection would identify single-step execution patterns suggestive of instruction-level debugging by comparing the wall-clock time of each block execution to an expected range inferred from block size and reference timing. The realistic cost of debugger-based block-by-block binary extraction would be significantly increased by these countermeasures.

7. Runtime Anomaly Detection and Behavioral Monitoring

The incorporation of lightweight runtime anomaly detection into the RBM is a conceptually ambitious future improvement. The RBM could identify statistically anomalous execution patterns, such as those caused by external instrumentation tools or hardware performance counter-based side-channel measurement attempts, and initiate a protective response, such as process termination or memory zeroing, by keeping a statistical model of expected block execution durations, memory access patterns, and block transition frequencies derived from profiling the binary during the preprocessing phase. A longer-term research avenue for adaptive runtime protection is AI-assisted

International Journal of Convergent Technologies and Future Systems (IJCTFS)
Website: https://www.maulikavyanshufoundation.org/IJCTFS/index.php
Email: maulikavyanshuijctfs@gmail.com | kavyanshufoundation@gmail.com
Volume 02 Issue-03 May-June 2026 ISSN NO: 3108-1312

behavioral anomaly monitoring, which uses a tiny on-device model trained on expected execution behavior patterns.

8. Cloud-Assisted Licensing and Key Distribution

Static RSA private key storage in the Secure Loader could be completely replaced with a cloud-assisted key distribution mechanism in deployment circumstances where continuous internet access is acceptable. The RSA private key would only be stored on a key distribution server under vendor control under this scenario. The Secure Loader would obtain a session-encrypted per-block key table during runtime, authenticate to the key server using a hardware attestation token (TPM or SGX remote attestation), and decrypt blocks using session-scoped keys that expire at the conclusion of the execution session. At the expense of network requirement, this strategy offers the strongest software distribution key management model by completely doing away with local private key storage.

9. Secure Update and Patch Distribution System

The creation of a secure update and patch distribution system that works with the SRSPT container format is a complementary future research avenue. A naive method would necessitate re-encrypting the entire binary whenever a software vendor issues a binary update. Differential block updates, in which only modified blocks are re-encrypted and re-packaged, would be supported by a more effective suggested architecture. Container version metadata would allow the Secure Loader to verify update authenticity prior to incorporating new blocks into the execution environment. For large binaries, this method would significantly reduce build-time preprocessing cost and update download sizes.

10. Virtualization-Resistant Execution Environments

A new challenge to dynamic analysis is virtualization-based analysis environments, like Cuckoo Sandbox, QEMU-based analysis platforms, and enterprise virtual machine introspection (VMI) solutions. Timing-based virtualization overhead detection and CPUID-based hypervisor bit inspection would be two potential anti-virtualization improvements to the SRSPT Secure Loader. Although research on evasion is still ongoing, integrating anti-virtualization techniques as an optional protective layer will increase the expense of automated dynamic analysis in sandboxed environments.

XVI. CONCLUSION

The Secure Runtime Software Protection Tool (SRSPT), a conceptual cybersecurity protection framework, has been introduced in this paper. It addresses the basic drawback of traditional encryption-based protection and static packers: the simultaneous presence of the entire decrypted binary in process memory. The suggested SRSPT architecture would limit the plaintext exposure window to a single block duration at runtime by dividing the compiled binary into AES-256-CBC encrypted blocks [1], wrapping each block's key with RSA-2048-OAEP [2], verifying each block's integrity via SHA-256 [3], and executing only one block at a time before securely zeroing the execution memory.

The capabilities of the attacker, the bounds of trust, the objectives of the attack, and the constraints of the suggested software-only defense design were identified in a formal threat model. SRSPT's theoretical resistance was evaluated using an enhanced security study.

Inspired by Gurajapu and Agarwal's function-level runtime encryption technique [7], the conceptual architecture extends their immediate memory-wipe and ephemeral decryption ideas to the built binary level with an additional asymmetric key wrapping layer [2]. Additionally, software protection literature [5][6] and trusted execution

environment ideas [4] are incorporated into the architecture. Anti-piracy systems, game protection, proprietary AI inference binary protection, military and air-gapped software, financial applications, and industrial control systems were among the conceptual real-world application categories that were found.

It is clearly stated that the most important path for future hardening through Intel SGX [4], TPM, or ARM TrustZone integration is the major design limitation—software-based RSA key storage within the Secure Loader. Future improvements include runtime anomaly detection, adaptive block scheduling, anti-debugging countermeasures, AES-256-GCM migration, and cloud-assisted key.

Inspiration from Prior Runtime Protection Research

The runtime function-level encryption concepts, memory ephemerality principles, and cryptographic integrity verification workflow presented by Gurajapu and Agarwal [7] in their paper "Secure Runtime Encryption of Critical Source-Code Functions for IP Protection," published in the World Journal of Advanced Research and Reviews (2026), served as the direct inspiration for this work. In particular, that earlier work introduced the architectural principles of (a) encrypting protected code artifacts independently and decrypting them only in memory at runtime, (b) using SHA-256 hashing [3] to verify integrity prior to execution, and (c) immediately zeroing all sensitive in-memory data following execution. By adding RSA-2048 asymmetric key wrapping [2] and a special Secure Loader component not found in [7], SRSPT independently re-proposes and expands these concepts at the built binary block level.

Acknowledgements

The Ahinsa Institute of Technology Dondaicha's Department of Computer Engineering faculty is acknowledged by the writers for their advice and assistance during this study. Gurajapu and Agarwal [7], whose conceptual framework influenced the architectural direction of this work, deserve special recognition.

REFERENCES

[1] National Institute of Standards and Technology (NIST), “FIPS PUB 197-upd1: Advanced Encryption Standard (AES),” NIST, Gaithersburg, MD, Updated May 9, 2023. DOI: https://doi.org/10.6028/NIST.FIPS.197-upd1.

[2] E. Barker, “NIST Special Publication 800-57 Part 1 Rev. 5: Recommendation for Key Management — Part 1: General,” NIST, Gaithersburg, MD, May 2020. DOI: https://doi.org/10.6028/NIST.SP.800-57pt1r5.

[3] National Institute of Standards and Technology (NIST), “FIPS PUB 180-4: Secure Hash Standard (SHS),” NIST, Gaithersburg, MD, Aug. 2015. [Online: https://csrc.nist.gov/publications/detail/fips/180/4/final]

[4] V. Costan and S. Devadas, “Intel SGX Explained,” IACR Cryptology ePrint Archive, Report 2016/086, 2016. [Online: https://eprint.iacr.org/2016/086]

[5] C. Collberg and J. Nagra, Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley Professional, ISBN: 978-0-321-54925-9, Aug. 2009.

[6] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, “A Survey on Automated Dynamic Malware-Analysis Techniques and Tools,” ACM Computing Surveys, vol. 44, no. 2, Article 6, Feb. 2012. DOI: https://doi.org/10.1145/2089125.2089126.

[7] A. Gurajapu and A. Agarwal, “Secure Runtime Encryption of Critical Source-Code Functions for IP Protection,” World Journal of Advanced Research and Reviews, vol. 29, no. 1, pp. 734–737, Jan. 2026. DOI: https://doi.org/10.30574/wjarr.2026.29.1.0079. — Main Conceptual Inspiration Paper.