

Article

Securing VLSI Layouts via Format-Preserving Encryption: A Selective Cryptographic Approach for Multi-Tiered GDSII Access

George K. Kranas ¹, Georgios Spathoulas ¹, Thanasis Loukopoulos ¹ and Antonios N. Dadaliaris ^{2,*}

¹ Department of Computer Science & Biomedical Informatics, University of Thessaly, 35131 Lamia, Greece; gekranas@uth.gr (G.K.K.); gspathoulas@uth.gr (G.S.); luke@uth.gr (T.L.)

² Department of Informatics & Telecommunications, University of Thessaly, 35100 Lamia, Greece

* Correspondence: dadaliaris@uth.gr

Abstract

The transition to a globalized, fabless semiconductor manufacturing model has integrated third-party foundries and external intellectual property (IP) vendors into the integrated circuit (IC) design cycle. While this collaborative system promotes innovation, it also exposes layouts to security threats. Protecting these designs is paramount; however, applying traditional encryption methodologies fundamentally alters the syntactic hierarchy of the industry-standard GDSII stream format, causing electronic design automation (EDA) tools to crash. Furthermore, a full encryption hinders modern system-on-chip (SoC) development, where different teams require access to specific modules of the design, without exposing the entire IP. To resolve this issue between collaborative layout sharing and zero-trust security, this paper presents a software implementing an encryption engine. By applying the NIST-standardized FF1 Format-Preserving Encryption (FPE) algorithm directly to the geometric data, the proposed software obfuscates sensitive spatial coordinates and structural nomenclature while maintaining the native GDSII format. The engine embeds multi-tiered cryptographic access control directly into the layout, utilizing native metadata properties. This framework allows proprietary logic to be securely compartmentalized, ensuring that interacting parties only view the specific structures they possess the clearance to decrypt.

Keywords: electronic design automation; hardware security; IP protection; GDSII; format-preserving encryption; tiered cryptographic access control



Academic Editors: Yang Yi and John Kalomiros

Received: 3 June 2026

Revised: 11 July 2026

Accepted: 21 July 2026

Published: 23 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

The modern semiconductor industry is driven by a highly distributed and horizontal supply chain, where the design, verification, and manufacturing of integrated circuits (ICs) are often handled by different entities. While this fabless model has accelerated technological innovation, outsourcing manufacturing has introduced severe security vulnerabilities into the chip design cycle, including IC overproduction, the sale of out-of-specification chips, and the piracy of highly sensitive intellectual property (IP) by untrusted third-party foundries [1]. At the center of this vulnerability is the exchange of physical layout data itself, which serves as the ultimate blueprint for chip fabrication.

For decades, the Graphic Data System II (GDSII) [2] format has been the industry standard for data exchange in electronic design automation (EDA). A GDSII file depicts the entire physical layout of a chip in a hierarchical structure, including geometric data, such as BOUNDARY (polygons) and PATH (wires). These pieces of information define the

exact placement of every transistor, interconnect, and standard cell on the silicon wafer. The fabrication of a modern mask is highly costly; as such, any error or misinterpretation in the GDSII file can result in the failure of the procedure, thus vastly increasing this cost. Consequently, the semiconductor industry is highly conservative, relying mostly on established GDSII flows and legacy EDA tools.

Considering the importance of the layout data, as well as the cost of the fabrication, the security of these files is a necessity that also presents a technical challenge. Traditional encryption algorithms, such as Advanced Encryption Standard (AES) [3] operating in Cipher Block Chaining (CBC) [4] mode, transform data into high-entropy binary strings that completely change the format and length of the original input. Applying this type of encryption to a GDSII file would corrupt its hierarchical structure, causing downstream EDA compilers and layout viewers to instantly crash, rendering the file format useless for the manufacturing steps. Cost-wise, modifying the already existing systems across the entire supply chain to accommodate standard encryption would require an economically unviable overhaul of the already established design flows, while also adding possible developmental delays, both in terms of modifying the procedure itself as well as its execution.

Furthermore, in academia and research environments, while simpler formats are occasionally used, they lack the industrial-scale features necessary for modern manufacturing, such as complex namespace management, array support, and datatype layers, therefore hindering researchers' effort. Based upon these reasons, finding a way to secure the industry-standard GDSII format without breaking compatibility is a critical imperative for both academic research and industrial fabricators.

Securing data without destroying its structural integrity requires the application of Format-Preserving Encryption (FPE) [5]. The advantage of FPE over classic encryption is its ability to secure data within systems that have rigid and predefined data models. FPE algorithms guarantee that the ciphertext belongs to the exact same domain and format as the plaintext. In the context of GDSII encryption, FPE allows the specific coordinate arrays of proprietary geometries to be mathematically scrambled while preserving the exact multi-line ASCII formatting and file length required by strict EDA parsers.

While applying FPE across an entire layout solves the format compatibility issue, modern collaborative environments also demand selective and multi-level data obfuscation. In system-on-chip (SoC) development, IP providers frequently license hard macros to third-party integrators. A selective encryption approach allows providers to scramble internal proprietary geometries, rendering them as "black box", while leaving external pin coordinates and outer boundaries in plaintext to enable accurate floorplanning and routing. A tiered security model is also necessary for secondary supply chain interactions, not just IP integration. For example, EDA software vendors and IT engineers require GDSII datasets in order to test and evaluate their software without risking IP exposure. Compliance and contractual obligations often dictate that external partners verify designs and non-sensitive information without acquiring the capability to access the full design; hence, an all-or-nothing encryption scheme proves restrictive and emphasizes the need for varying levels of structural visibility depending on the interacting party's role.

Similarly, in academic and research settings, this selective approach bridges the gap between theoretical EDA research and industrial application. It enables the safe sharing of benchmark layouts, allowing researchers to mask restricted proprietary foundry data while leaving the necessary structural topology fully accessible for study.

To address these challenges, this paper introduces a novel encryption engine: a security platform designed to implement tiered cryptographic access control directly within the physical layout file while preserving its original format. The proposed engine makes the following key contributions to the field of hardware security and EDA:

- **Format-Preserving Geometric Obfuscation:** We apply the NIST-standardized FF1 encryption algorithm directly to the coordinate arrays of proprietary elements. By maintaining the GDSII “Boundary Closure Rule” and preserving the multi-line formatting required by EDA parsers, the engine generates an encrypted layout that completely obfuscates visual IP while remaining compliant with the downstream tools.
- **Selective Targeting and Filtering:** The engine employs a parser that actively bypasses open-source or foundry-standard cells, leaving them in plaintext, ensuring that essential foundational routing and standard architecture remain intact and verifiable by third parties.
- **Tiered Cryptographic Access Control:** We introduce a dynamic layer-tagging system utilizing GDSII metadata fields. Using that, the engine can inject tier-based cryptographic access-control tags into the layout.
- **Look-Ahead Decryption Buffer:** We implement a decryption buffer that reads these metadata markers and checks the user’s specific key clearance. Our buffer enables partial, module-by-module decryption based on authorization level, allowing a single GDSII file to be securely distributed.

The remainder of this paper is organized as follows: Section 2 provides the background theory. Section 3 details the materials and cryptographic methods. Section 4 describes the architecture and implementation of the proposed engine. Section 5 presents the experimental results and performance evaluation, and Section 6 discusses the findings alongside future work.

2. Background Theory

To provide a foundation for our work, this section bridges the domains of hardware security, traditional cryptography, and structured data protection. Given that the direct application of FPE to VLSI layout data is highly novel, to our knowledge and research, it is necessary to examine the underlying vulnerabilities of IC design and the cryptographic primitives that enable our proposed solution.

The globalization of the semiconductor supply chain has exposed physical layouts to untrusted foundries and third-party vendors. Adversaries have developed comprehensive and sophisticated methodologies for chip-to-system reverse engineering, threatening the proprietary logic of fabless design houses [6]. Modern adversaries do not rely solely on manual inspection, while automated tools designed for gate-level netlist reverse engineering can process layout data to recover core functionality or identify insertion points for malicious logic [7]. Based on that, there is a necessity to secure geometric primitives before the GDSII file is transmitted across the supply chain.

The broader hardware security field increasingly relies on layout-aware methodologies and emerging device architectures to mitigate a wider spectrum of threats. For instance, recent advancements explore device-level protections utilizing magnetic tunnel junctions and spintronic synapses to secure deep neural networks and hardware functions against model replication and machine learning attacks [8–10]. Similarly, active physical vulnerabilities are addressed through comprehensive fault-attack countermeasure benchmarking [11] and layout-aware assessments targeting timing fault-injections [12]. These approaches highlight the critical intersection of device-level architectures, physical layout properties and hardware security. However, their primary objectives focus on active device-level defenses, leaving a gap regarding zero-trust, static data confidentiality for IP distribution. This gap also extends to traditional IP protection methods. While logic and layout obfuscation counter reverse engineering, they alter a design’s geometry, which can degrade overall performance and require re-synthesis. Additionally, standard Digital Rights Management (DRM) frameworks typically demand proprietary toolchains that operate solely at the RTL or netlist stages. These limitations emphasize the need for a solution that secures the layout at the syntactic level,

therefore protecting the layout without altering the underlying logic or relying on external toolchains. The format-preserving approach proposed in this paper addresses this constraint.

The response to data vulnerability is encryption; however, traditional block ciphers, such as the Advanced Encryption Standard (AES), maximize entropy and are designed to operate on fixed blocks, which requires padding mechanisms that change the length, character set, and structural boundaries of the plaintext [13]. As discussed in Section 1, GDSII is a rigid format; as such, standard encryption algorithms and methods are practically incompatible with this format due to the hierarchical data structures required by EDA parsers. The application of standard block ciphers upon an EDA layout file corrupts its syntactic hierarchy, causing downstream compilers to crash or produce errors. Based on that, the semiconductor industry would require a cryptographic method that operates within the confines of the GDSII data model rather than breaking it.

To address the issues produced by the use of traditional algorithms on highly structured data, the cryptographic community developed Format-Preserving Encryption (FPE). A building material of modern FPE algorithms is the tweakable block cipher, which, unlike standard ciphers, can incorporate an additional, and sometimes public, parameter called tweak. This tweak introduces variability and ensures domain separation without requiring the underlying cryptographic key to be changed [14]. Our engine prevents the generation of identical ciphertext entries inside the encrypted file, where identical coordinate integers would result in the same encrypted values, by altering the tweak and basing it upon the context of geometrical data.

The concept of securing sensitive data while leaving structural headers intact has been utilized in other fields, like multimedia security, where selective encryption allows video players to parse streaming formats while the underlying visual data remains obfuscated [15]. While this is established for image and video formats, there is currently no existing research exploring the application of FPE directly to the data of VLSI layout files. Our work is an effort to fill that gap by adapting this approach to the EDA domain.

In collaborative SoC development, monolithic file encryption is easily applicable; however, engineering teams and external partners require access to specific localized modules without exposing the entire chip. Establishing selective security and access permissions across a distributed offline system presents challenges.

Traditional organizational security relies on Role-Based Access Control (RBAC) [16] to enforce the principle of least privilege. Full RBAC systems, however, necessitate continuous identity management, active authentication, authorization, policy revocation and auditability. Applying such dynamic policy enforcement within a static offline layout file is impractical.

While inspired by the organizational goals of RBAC, specifically enforcing the principle of least privilege, our scheme realizes a strict tiered cryptographic access control model. By moving the access control logic away from the application level and embedding it directly into the physical layout data using native GDSII metadata fields, the engine injects a multi-tiered cryptographic tagging system. This framework encrypts sensitive modules based on the user's specific cryptographic key clearance while leaving non-sensitive ones in plaintext. This ensures that third parties view only the specific structures they are authorized to access, while achieving a zero-trust layout distribution model that is both self-contained and statically secure.

3. Materials and Methods

In order to achieve Format-Preserving Encryption (FPE) and multi-tiered access control over IC layout data, our methodology relies on a combination of data parsing and modern encryption standards. This section analyzes the structural materials used by our software.

3.1. GDSII Data Model

In order to process and encrypt/decrypt the GDSII data format, our system converts it into ASCII text form. Our methodology targets specific geometric and structural primitives that contain sensitive parts of a layout design. These primitives are *BOUNDARY* records that define closed polygons via sets of coordinates, *PATH* records defining the interconnections/wires, and, finally, *SREF/AREF* records that define the structural and array references of the design.

A constraint presented by the GDSII format is the “Boundary Closure”, which dictates that the first and the last sets of coordinates of a polygon must be identical. Furthermore, the format has a strict hierarchical mapping using Layer and Datatype integers to represent physical materials. As such, any encryption process must preserve these geometric rules, coordinate lengths, and layer numbers to remain compliant with EDA parsers.

3.2. Format-Preserving Encryption (FF1)

The encryption algorithm applied to the GDSII files by our engine is the FF1 algorithm, which is an NIST-standardized FPE scheme originally proposed by its creators as the “FFX Mode of Operation” in [17]. FF1 operates as a 10-round Feistel network, using AES in its core.

GDSII files contain geometric coordinates (integers) and string-based naming in their modules; consequently, FF1 is configured to operate across these two domains:

- **Geometric Coordinate Domain (Numeric):** Physical XY coordinates are encrypted using a radix of 10. To accommodate the negative coordinates that exist in layout designs, we apply an offset to map all coordinates into the positive space before FF1 encryption begins, ensuring that the cipher operates in a valid domain.
- **Module Namespace (Alphanumeric):** Cell names and structure references are encrypted using a radix of 72. This radix corresponds to the range of letters, digits, and special characters that are compliant with GDSII format.

3.3. Contextual Tweak Derivation

To ensure that plaintext coordinates will not be encrypted to identical ciphertext coordinates, our software uses a contextual tweak. For every geometric element, a unique string is constructed by combining the cell’s name, the active layer, and the use of a global counter. After its construction, this string is hashed using SHA-256 [18], providing a unique tweak for FF1. As a result, these identical geometric shapes placed in different areas of the layout will have different ciphertext values.

3.4. Selective Processing and Look-Ahead Buffering

Our method implements a “Look-Ahead” block-by-block buffering technique in order to achieve tiered access control and selective encryption. When the layout is being parsed, geometric blocks are buffered in the memory one-by-one. A filter is also applied upon the buffer that is responsible for identifying cell references and categorizing them based upon the user’s settings. As such, if a block is identified as “non-sensitive” it will not be encrypted, while if it is set as “sensitive” it will be processed by FF1. To solve the Boundary Closure issue and set the polygon coordinates correctly, the final set of encrypted coordinates will be set as equal to the first.

3.5. Threat Model and Adversary Capabilities

To evaluate the security of the proposed engine, we define the threat model based on semi-honest adversary within the IC supply chain. The adversary represents a participant with legitimate interest, such as an IP integrator, IT engineer, or research partner. In these examples, the adversary receives the GDSII file but lacks the cryptographic clearance (keys) to access specific higher-tier encrypted modules. The foundry’s role is excluded from the

protection model as it requires a fully decrypted layout for manufacturing. We also assume that the underlying cryptographic primitives used within FF1 and subsequently AES, are mathematically secure against standard cryptanalysis. Under this model, the adversary's capabilities and available knowledge are defined as follows:

- **Access to partial keys:** Tiered access control dictates that an adversary may possess the keys for lower-security tiers, but lacks the highest tier key. Because our framework utilizes a single-pass encryption architecture, described in Section 4.3, with independently generated keys per tier, possessing lower-tier keys provides no advantage in deriving higher-tier keys or decrypting higher-tier blocks.
- **Known plaintext and ciphertext pairs:** Lower-tier users receive a fully encrypted file and can generate a partially decrypted file based on their tier, as presented in Section 4.3. With these files they can derive plaintext–ciphertext pairs for their tier that they have clearance for. The security of the encrypted modules (higher tier) relies on FF1's proven resistance to known-plaintext and chosen-plaintext attacks.
- **Standard cell and layer knowledge:** The adversary possesses knowledge of standard cell libraries and PDKs. Because our engine selectively bypasses non-sensitive modules and preserves layer and datatype integers to maintain EDA compatibility, discussed in Section 4.2, the adversary can identify foundational routing and standard architecture.
- **Metadata and topological leakage:** In order to facilitate zero-trust layout sharing, the engine injects plaintext metadata tags into the layout to bind encrypted geometries to their security tiers (detailed in Section 4.2.4). As a result, the adversary can observe the number of encrypted modules, their designated security clearance levels, physical layers and structural names, as well as external pin coordinates left in plaintext for floorplanning. Furthermore, because FF1 is format-preserving, the vertex count of encrypted polygons remains identical to the plaintext.

Based upon the previous statements, the security claim of this work is the strict confidentiality of encrypted geometric coordinates and custom structural names against unauthorized decryption. Protection against active tampering (integrity attacks) or the complete concealment of macro-level design topology and module identity is not targeted by our methodology.

4. Architecture and Implementation

This section details the architecture and implementation of our software. To provide a comprehensive overview, the system is first described based on the architectural design, followed by the implementation and technical details.

4.1. Architecture

The primary challenge in processing GDSII files is managing their volume as modern SoC layouts often have sizes that range from a few megabytes to tens of gigabytes [19], which contain a lot of information, detailing geometric data. As such, loading an entire GDSII document into memory to perform encryption is computationally heavy.

To tackle this issue, our engine was designed around a Look-Ahead streaming paradigm. Instead of reading the entire file, we utilize a dynamic buffer that holds only one geometric element or structural reference at a time. When the parser encounters the End Element (*ENDEL*) termination flag, it routes the buffered block to the encryption procedure, writes the output to disk, and then empties the memory.

A secondary design choice was to ensure a self-contained output of zero-trust. Traditional access-control systems rely on external files or databases to manage permissions. However, in the actual IC supply chain, maintaining external synchronization with the physical layout file is costly and might lead to errors. We reasoned that the security metadata must exist within the geometry itself. To achieve this, our architecture uses native

GDSII property attributes, re-purposing them to act as cryptographic security tags. By injecting these tags directly into the modules that are going to be encrypted, the resulting file becomes self-contained, while also having multi-tiered security access.

Lastly, there is a Fail-Safe Cryptographic Fallback for the tiered access control. The system's key manager is structurally decoupled from the decryption parser. If a third-party user attempts to parse the layout but lacks the appropriate clearance key for a specific module, the architecture intercepts the resulting error state. Instead of terminating its execution, the system writes the block back to the output file as a ciphertext. This design allows unauthorized users to process the file for other tasks related to the fabrication pipeline (i.e., IT pipeline validation or top-level floor planning) while encrypted intellectual property remains safe.

If the sole objective was the prevention of unauthorized manufacturing, standard encryption methodologies would suffice. However, as stated previously, standard encryption transforms the whole layout into ciphertext, which breaks the EDA pipeline. A fully encrypted file cannot be opened in layout viewers, preventing third parties from performing top-level tasks such as floorplanning, area estimation, or routing. Our FPE engine tackles this by preserving the syntactic validity of the GDSII format. While the encrypted modules intersect and violate Design Rule Checks (DRCs), rendering the design un-manufacturable without the decryption key(s), the file remains EDA-compliant, and the third-party users can load the layout, view the unencrypted standard cells, and observe the boundaries and external pins of the encrypted modules. This allows the encrypted parts of the design to function as secure and viewable “black boxes”, enabling sub-system integration and routing in a zero-trust environment without exposing the encrypted geometries.

The overall architecture of our design is visualized in Figure 1

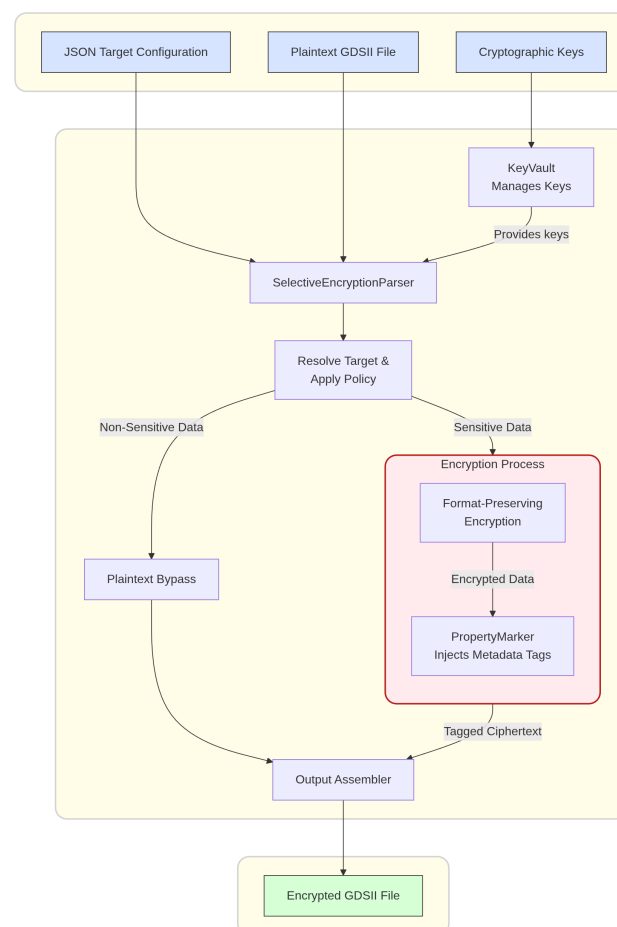


Figure 1. Architectural overview.

4.2. Implementation

Our software is implemented in Python 3.14.4 using an object-oriented design in order to support ease-of-use and extensibility.

4.2.1. Software Overview

The encryption methodology utilizes the PyCryptodome [20] library. FF1 invokes AES to serve as the underlying encryption function. Contextual tweaks (T) are generated deterministically using the standard hashlib library via SHA-256. Given that FF1 requires a specific tweak length, the resulting encrypted part is truncated to the first 16 bytes.

The execution logic is divided into different components:

- **SelectiveEncryptionParser** and **BufferedDecryptionParser**: These classes manage look-ahead buffering, track global coordinate counters to synchronize encryption tweaks, and invoke FF1.
- **KeyVault**: The secure key management center stores and retrieves encryption keys based on security tiers. It facilitates the multi-level access policy by raising a *KeyError* if a locked tier is accessed, which triggers the decryption parser to output ciphertext.
- **PropertyMarker**: Responsible for the exact string generation and parsing of the GDSII metadata tags used for tiered access control.
- **TargetConfiguration**: Maintains the selective bypass logic by utilizing string matching to map specific structural names (*SNAME*) to target security levels, or explicitly flag non-sensitive modules as plaintext. It should be noted that the target configuration is set by the user into a *json* file by explicitly declaring which cells of the design are to be encrypted and which are to be left as maintext. Also, the user must set the level of access for each of the modules they wish to encrypt. As mentioned in our threat model, this selective bypass leaves standard cell libraries and foundational routing in plaintext, acknowledging that any interacting party will possess full structural knowledge of the non-proprietary architecture.

4.2.2. Tokenization and State Context

To ensure compatibility with EDA parsers, the software uses a custom tokenizer. The tokenizer iterates through the input file, capturing the original whitespace and carriage returns, if they exist. To maintain semantic accuracy, the tokenizer forms data dictionaries that contain the parsed keyword and tokens before updating its internal state trackers. Our parser dynamically manages flags such as “inside_boundary_box” and increments a “boundary_id” when a new polygon is detected and ensures that the generated tweaks remain contextually bound to the specific physical layer declared on the file.

4.2.3. Cryptographic Execution

When a block is routed to be encrypted, alphanumeric data (such as cell names) are padded with underscores if needed to meet the minimum domain length of FF1 (four characters). For geometric data, XY arrays are flattened and the coordinate values are passed into a wrapper that pads the numeric fields to a fixed length of 12 digits, resulting in a large ciphertext domain. Furthermore, in order to prevent structural information leakage during the instantiation of hierarchical components, the engine secures Structure References (SREF) and Array References (AREF). If a referenced sub-module is flagged as sensitive, leaving its insertion coordinates in plaintext would leak the macro-level floorplan and placement topology of the encrypted parts of the design. Rather than encrypting the top-level keyword, the parser buffers the entire reference block and resolves the target clearance based on the referenced *SNAME*. If it is deemed as sensitive, the engine encrypts two fields to obfuscate both identity and spatial placement. The first is the *SNAME*

string, which is encrypted with FF1 over a radix-72 alphanumeric string, incorporating padding to satisfy FF1's minimum length requirements. The second field comprises XY insertion coordinates that are encrypted pairwise, once again, using FF1 over a radix-10 domain, mapping signed values to a shifted fixed-length decimal domain. Supporting spatial records, such as transformations *STRANS*, are preserved in plaintext to ensure that the hierarchical references remain syntactically valid for EDA tools while concealing the underlying encrypted topology.

4.2.4. Metadata Tagging

In order to implement the access control logic without violating the GDSII standard, our engine utilizes native metadata properties, specifically the *PROPATTR* (Property Attribute) and *PROPVAlUE* (Property Value) records. We decided to use *PROPATTR* 99 as the exclusive identifier for our security tags, as the number falls within the accepted range for GDSII properties (1 to 127) but remains unreserved by the original Calma specifications, allowing for safe, custom implementation without risking data collisions.

During the encryption phase, the engine's PropertyMarker segment constructs a string that maps the encrypted geometry to its security tier (level), module name (cell_name), and physical layer (layer). This string is formatted as "L:{level} | C:{cell_name} | LY:{layer}" (e.g., L:LEVEL_2 | C:alu | LY:10), and is assigned as the *PROPVAlUE* string. To ensure EDA compatibility, the metadata tag is injected as plaintext, thereby introducing the expected topological leakage defined in our threat model. Specifically, any user, regardless of their access tier, can read these tags to observe the structural names, physical layers and designated access levels of the encrypted modules.

By pairing *PROPATTR* 99 with *PROPVAlUE* string, the engine creates a standard two-line metadata tag that adheres to GDSII syntax rules. This ensures that EDA layout viewers can parse both encrypted and partially decrypted files without syntax errors. Finally, this two-line tag is injected directly into the geometric block's buffer preceding the *ENDEL* (End Element) terminator. This specific placement and structure optimizes the decryption parser. When the engine detects a *PROPATTR* 99 flag, it acts as a signal that the following *PROPVAlUE* line contains the definitive cryptographic clearance required to decrypt that specific boundary.

In order to consolidate the methodologies detailed in the preceding subsections, as well as this one, the execution pipeline is formalized in Algorithm 1.

Algorithm 1 Encryption pipeline.

Require: ASCII GDSII File (F_{in}), Target Configuration (T_{config}), KeyVault (K)

Ensure: Encrypted ASCII GDSII File (F_{out})

```

1:  $Buffer \leftarrow []$ 
2: while not End of File ( $F_{in}$ ) do
3:    $L \leftarrow \text{Read\_Next\_Line}()$ 
4:   if  $L$  begins BOUNDARY, PATH, SREF, AREF, BOX then
5:      $Buffer.append(L)$ 
6:     while  $L \neq \text{ENDEL}$  do ▷ Look-ahead buffering
7:        $L \leftarrow \text{Read\_Next\_Line}()$ 
8:        $Buffer.append(L)$ 
9:     end while
10:     $Target \leftarrow \text{Resolve\_Target}(Buffer, T_{config})$ 
11:    if  $Target$  is Encryptable then
12:       $Level \leftarrow T_{config}.get\_level(Target)$ 
13:       $Key \leftarrow K.get\_key(Level)$ 
14:       $Buffer \leftarrow \text{Encrypt\_Sensitive\_Fields}(Buffer, Key)$ 
15:      if Block is BOUNDARY then
16:         $Buffer \leftarrow \text{Enforce\_Closure}(Buffer)$ 
17:      end if
18:       $\text{Inject\_Tag}(Buffer, \text{PROPATTR } 99, Level)$ 
19:    end if
20:     $\text{Write}(Buffer)$  to  $F_{out}$ 
21:     $\text{Clear}(Buffer)$ 
22:  else
23:     $\text{Write}(L)$  to  $F_{out}$  ▷ Pass-through plaintext
24:  end if
25: end while

```

4.2.5. Boundary Closure

Identical plaintext coordinates yield different ciphertext when encrypted due to the incrementing tweak counter; to resolve that, our engine enforces polygon closure to prevent downstream errors. After a coordinate array is encrypted, the software overwrites the final ciphertext pair to match the first. The mechanism for extracting and overwriting the ciphertext vertex to maintain format compliance is showcased in Algorithm 2. During decryption, however, processing this overwritten pair through FF1 would cause a tweak mismatch, resulting in an incorrect plaintext. Once again, to tackle this, our parser identifies the final vertex of a *BOUNDARY*, bypasses the FF1 decryption module and maps the final coordinates directly back to the first decrypted pair, restoring the original structure of the polygon.

Algorithm 2 Inline boundary closure enforcement.

Require: Buffered **BOUNDARY** block, target configuration, current tweak parameters

Ensure: Format-Compliant Closed Ciphertext Polygon

```

1:  $C_{array} \leftarrow []$ 
2: for each coordinate pair  $(x, y)$  in block do
3:    $(x', y') \leftarrow \text{FF1\_Encrypt}(x, y, \text{radix} = 10, \text{tweak})$ 
4:    $C_{array}.\text{append}((x', y'))$ 
5: end for
6: if  $\text{Length}(C_{array}) \geq 4$  then                                ▷ Valid polygon requires at least 4 vertices
7:    $\text{First\_Vertex} \leftarrow C_{array}[0]$ 
8:    $C_{array}[\text{Length}(C_{array}) - 1] \leftarrow \text{First\_Vertex}$       ▷ Overwrite final vertex to close the
   ciphertext polygon
9: end if
10: return  $C_{array}$ 

```

4.3. End-to-End Execution Workflow

In order to present the architectural and implementation details described above in a more condensed manner, the complete operation workflow of our engine functions as follows.

The process starts with the user defining the security policy. The user inserts their custom settings using a structured JSON configuration file that acts as the master access control list, mapping specific cell names to their required security clearance tiers. The underlying object-oriented architecture is dynamic and supports any number of access levels (N-tiers), though our subsequent examples reference a three-tier model as a proof of concept. Non-sensitive cells are explicitly flagged to bypass the encryption and remain plaintext. Alongside this configuration, the user provides the cryptographic keys for each tier, which the engine loads into its key vault.

Once configured, the engine begins parsing the original, plaintext ASCII GDSII file block by block. The multi-tiered security is implemented via a single-pass, tagging architecture rather than nested (onion) encryption. This means each proprietary block is encrypted exactly once using the specific cryptographic key corresponding to its assigned access level. Our engine does not encrypt the data multiple times (i.e., Level 3 data are not encrypted with a Level 3 key, then a Level 2, and finally a Level 1 key). Each block is encrypted once and injected with the appropriate metadata tag, as described previously, to bind the sensitive data to its required clearance tier. This single-pass design prevents computational bloat while maintaining the tiered access control.

Assuming a three-tier configuration, a complete execution of our software produces four output files:

- **Encrypted File:** The output where all sensitive data across all tiers are scrambled.

- **Level 1 Decrypted File:** The output file containing data decrypted with the lowest access level.
- **Level 2 Decrypted File:** The output file containing data decrypted with the mid-tier access, including these from the previous level.
- **Level 3 Decrypted File:** In this file, all information is decrypted, basically restoring the layout to its original plaintext form.

By generating these files, the engine enforces the principle of least privilege, ensuring that any interacting party can parse only the data that they are authorized to access. Also, it inherently provides lower-tier users with corresponding plaintext and ciphertext blocks for their authorized modules. Our threat model accounts for this exposure. The engine utilizes independently generated keys per tier; therefore, possessing these pairs and partial lower-tier keys provides no advantage towards decrypting higher-tier encrypted parts of the design.

4.4. Auxiliary Procedures

Original GDSII files are in binary form, and in order for our engine to be able to perform its task, it is necessary that these files be converted to ASCII text files and vice versa. To achieve that, we used KLayout [21] via its Python API and package. While KLayout is open-source, we cannot embed it into our design flow, as the selection of such a tool should be up to the user's preference.

While our software incorporates a deterministic key fallback to ensure reproducible execution during benchmarking and evaluation, this mechanism is not a part of the core encryption methodology, and can only be used in the top level execution script. Consequently, this mechanism should be avoided in real-use scenarios and users should adhere to standard key management practices. To assist with secure setup, the system provides an auxiliary script that generates tier-specific AES keys using a pseudorandom number generator provided by Python's core packages. These keys should be securely stored and distributed to authorized parties via secure channels. Ultimately, our framework relies on symmetric encryption applied to static layout files, and, as such, it does not support dynamic access revocation. In the event of a key compromise, the confidentiality of previously distributed files encrypted with that specific key cannot be retroactively restored. Mitigation relies on routine key rotation, ensuring that any subsequent layout releases are encrypted with a new set of keys to re-establish secure clearances.

4.5. Quantitative Security Analysis

To evaluate the cryptographic resilience of the proposed architecture against the adversarial capabilities defined in our threat model, we present a quantitative security analysis of the engine's mitigation strategies.

The core geometric encryption utilizes the NIST-standardized FF1 algorithm, which inherits chosen-plaintext attack (CPA) resistance via its underlying AES implementation. A known vulnerability in structured layout data is geometric frequency, where, if identical coordinates produce identical ciphertext, an adversary could infer spatial relationships through differential analysis. To defend against this, the engine implements the deterministic contextual tweaking described previously. By applying an SHA-256 hash to a concatenated string of the structural name, physical layer, and a global incrementing counter, the engine enforces a unique ciphertext for every geometric block. This ensures that identical plaintext coordinates map to different ciphertexts, neutralizing geometric differential analysis.

Additionally, this type of encryption is mathematically robust against layout reconstruction and topology inference. As the true coordinates are pseudorandomly permuted, physical adjacency is destroyed and the adversary is prevented from inferring logic through

module placement. The framework counters cell recognition and pattern matching, as both the structural names and geometric boundaries are obfuscated. Ultimately, the FF1 algorithm acts as a strong pseudorandom permutation; thus, the resulting ciphertext coordinates exhibit a uniform distribution across the encrypted domain, rendering the layout mathematically resistant to coordinate-distribution attacks.

Lastly, as will be evaluated in Section 5.2, secondary attack vectors attempting to infer geometric complexity via file size analysis are inherently mitigated. The architectural design guarantees zero geometric byte expansion, and, as such, all storage overhead is proportional to the injection of plaintext metadata tags. Consequently, observing macroscopic file size differences provides the adversary with no exploitable leakage regarding the underlying vertex density, logic complexity or routing topology of the encrypted modules.

5. Results

5.1. Experimental Setup and Benchmark Datasets

To evaluate the performance, scalability, and overhead of our work, a series of benchmarks were executed. All experiments were conducted on a workstation equipped with an AMD Ryzen 7 5825U processor clocked at 4.5 GHz, 16 GB of memory, and running the Fedora 43 operating system. As mentioned previously, our software was implemented in Python 3.14.4, also using the PyCryptodome external package in its core flow, as well as KLayout for the conversion of GDSII files.

The engine was tested against 10 open-source designs provided by [22]. In the following table, Table 1, we detail the specifications of the layout designs we selected from the cited repository.

Table 1. Design specifications, sizes in MB.

#	Design Name	Original GDSII Size	Converted ASCII Size
1	jpeg	52.3	79.2
2	parameterized_freq_divider	46.3	66.1
3	bp_fe	38.5	60.5
4	tinyRocket	27.8	44.0
5	ibex	25.3	38.8
6	clock_divider	14.7	20.9
7	parameterized_fft	12.0	18.4
8	dynamic_node	12.0	19.0
9	dma_controller	11.9	17.6
10	riscv32i	9.80	15.0

Although the engine natively supports any number of access tiers, our experimental evaluation utilizes a three-tier configuration. To test our engine upon the aforementioned designs, we created two distinct scenarios with different security configurations:

- **Scenario 1:** This configuration simulates a strict distribution model that mirrors zero-trust intermediaries that require basic structural parsing without exposing any proprietary logic, for example, structural parsing for macro-level floorplanning. Level 1 (lowest tier) users are granted zero decryption rights, limiting their visibility to unencrypted plaintext data. Level 2 (middle tier) access allows the decryption of part of the modules, as seen in Table 2. Lastly, Level 3 (highest tier) represents the IP owner, possessing the clearance to fully decrypt all core logic and restore the complete original layout.
- **Scenario 2:** This configuration simulates a development pipeline where there are multiple third-party teams that require access to different sub-systems reflecting collaborative SoC integration, where distinct teams require partial decryption capabilities based upon their assigned sub-systems.. Unlike the previous scenario, security tags

are distributed across all three tiers to enable selective sharing. The levels of access maintain the same order of clearance, and the total number of modules that they can decrypt are also shown in Table 2.

The encryption settings overview of both scenarios is presented in Table 2. For every design in each scenario, we showcase the number of modules to be encrypted per level, as well as the number of blocks tagged/affected by this selection, counting the blocks of plaintext data relevant to each module that are flagged for encryption. By testing both scenarios, we establish baseline performance metrics that are shown in the following subsections of this paper.

Table 2. Security setup for both scenarios.

	Design Name	Level 1 Modules	Level 2 Modules	Level 3 Modules	Blocks Tagged
Scenario 1	jpeg	0	2	3	336,791
	parameterized_freq_divider	0	2	3	99,064
	bp_fe	0	2	3	262,281
	tinyRocket	0	2	3	193,027
	ibex	0	3	2	182,830
	clock_divider	0	1	3	30,406
	parameterized_fft	0	3	2	79,911
	dynamic_node	0	2	3	88,270
	dma_controller	0	3	2	54,443
	riscv32i	0	3	2	64,111
Scenario 2	jpeg	14	14	12	447,518
	parameterized_freq_divider	18	94	43	461,704
	bp_fe	20	70	3	360,628
	tinyRocket	22	44	28	269,290
	ibex	31	213	6	242,090
	clock_divider	17	109	7	150,309
	parameterized_fft	28	202	10	119,260
	dynamic_node	18	44	24	114,891
	dma_controller	22	72	2	116,713
	riscv32i	29	171	8	98,463

5.2. Encrypted and Decrypted File Size

A primary constraint in encryption is file size expansion, which introduces storage overhead and increased production costs. Figure 2 illustrates the sizes of the encrypted and partially decrypted files across both scenarios. Crucially, because FF1 preserves the exact character length of the coordinate arrays, the cryptographic process itself introduces zero byte expansion. The entire storage overhead is generated by the injection of the metadata tags required to implement multi-tiered access control. Consequently, a pattern emerges across all designs. The file size expansion is proportional to the volume of the injected tags.

In order to quantify the storage penalty, Table 3 details the exact percentage overhead for a representative sample of the benchmarks. In the first scenario, where fewer modules are encrypted, the overhead ranges from a 14.1% to 34.5% increase. In the second scenario, the heavier distribution of tags yields a higher expansion, reaching up to 64.7%. It should be noted that this overhead does not scale with the number of access tiers, as a Level 1 metadata tag has the same string length as a Level 3 tag. Instead, the size expansion scales with the total number of geometric blocks flagged for encryption.

However, another pattern is visible during decryption. As the user's clearances level increases, the engine removes the authorized metadata tags, causing the file sizes to progressively shrink from Level 1 to Level 3.

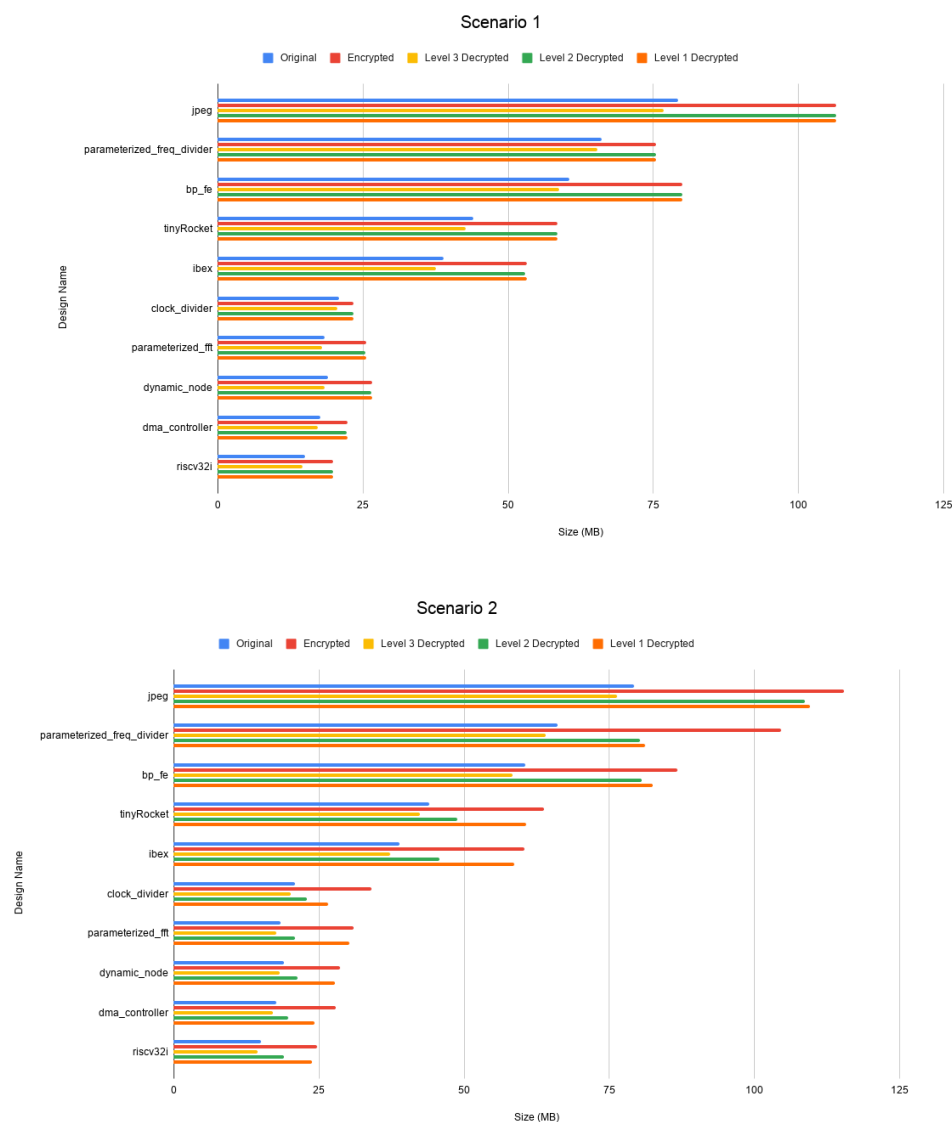


Figure 2. Encrypted and decrypted file sizes in ASCII format.

Table 3. Representative file size overhead (original vs. fully encrypted).

Design Name	Original (MB)	Scen. 1 Enc. (MB)	Scen. 1 Overhead	Scen. 2 Enc. (MB)	Scen. 2 Overhead
jpeg	79.2	106.5	34.5%	115.4	45.7%
parameterized_freq_divider	66.1	75.4	14.1%	104.6	58.2%
bp_fe	60.5	80.0	32.2%	86.7	43.3%
riscv32i	15.0	19.9	32.7%	24.7	64.7%

Across all designs and scenarios, the fully decrypted files (Level 3) consistently end up slightly smaller than the original ASCII layouts. This occurs because our engine actively strips space characters that were left in the original files during parsing, indirectly reducing the final file size footprint.

While our metadata injection introduces a storage penalty, it is an acceptable trade-off. Traditional encryption techniques would inflate the file size significantly while transforming the layout into an unusable format without full decryption. In contrast, our approach provides a selective, tiered cryptographic security framework that reinforces ease of use while also preserving EDA software compatibility.

From a security perspective, the file size expansion evaluated above introduces no exploitable vulnerabilities regarding data inference attacks. The storage overhead is derived, as stated, from the metadata tags; thus, analyzing the gross file size or observing the different size values across clearance levels provides no other leakage. An adversary possesses the capability to read these tags to identify and count the encrypted modules, and, as such, the overall file size reveals no hidden information regarding the complexity, vertex density, or topology of the encrypted geometry.

5.3. Processing Cost: Execution Runtime and Time per Tagged Block

In order to validate our encryption scheme, we measured the exact time required to encrypt the design with multiple levels of clearance and the decryption time per clearance level (Figure 3). Once again, using the largest design and workload in our disposal as benchmark (jpeg, scenario 2), a complete three-level encryption took about 97 s, while a level 3 decryption, decrypting 447,518 blocks, required almost 91 s. When the clearance level was restricted to level 2, the run time was measured at approximately 24 s. Finally, a level 1 decryption took about 22 s.

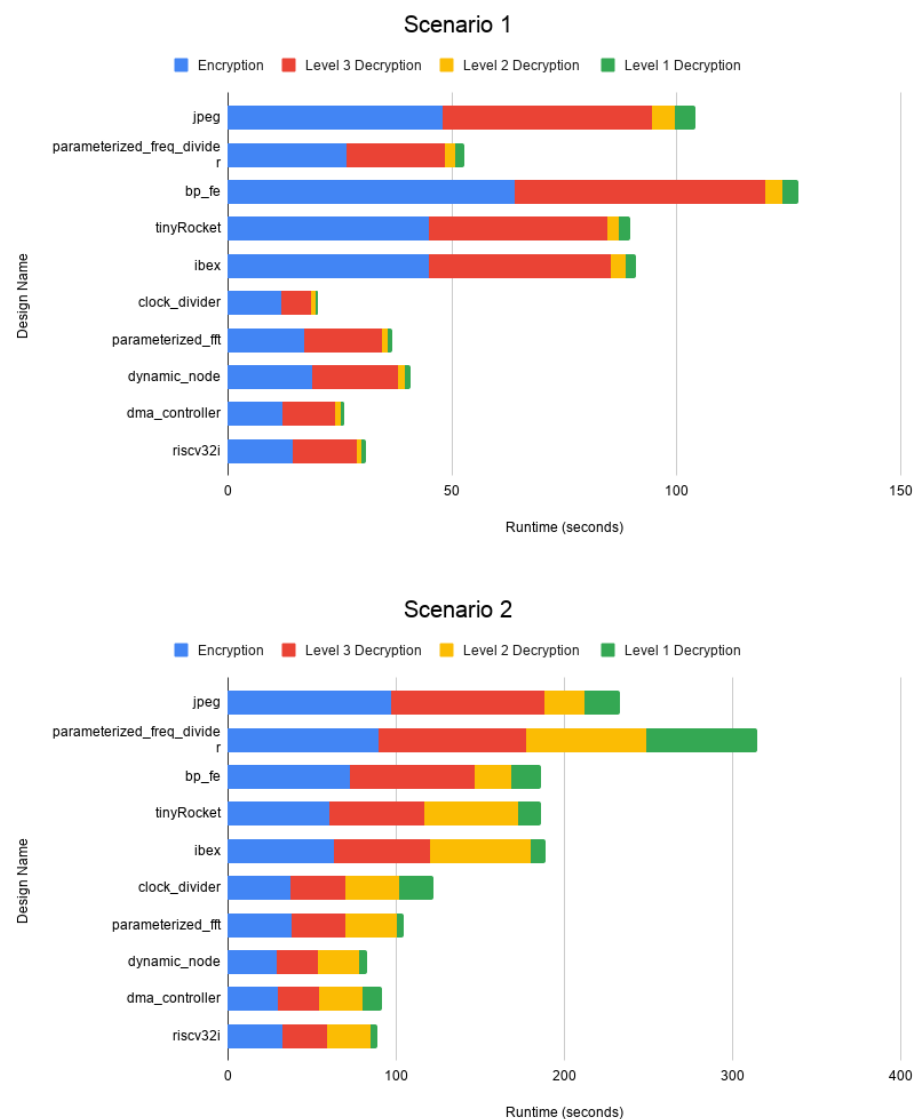


Figure 3. Stacked execution runtime.

These results provide proof that the tagging system in our implementation successfully compartmentalizes execution time. This modularity ensures that the computational burden of decryption is proportional to the number of sensitive modules the user has set per level of clearance, achieving zero-trust layout sharing. On the other hand, our encryption scheme that obfuscates each level separately with a different access key, encrypting each block once, generates similar results. An N-level tiered encryption is proportionate to the number of blocks that are going to be encrypted (user's settings) while maintaining a minimal execution runtime.

Semiconductor modules vary in size; hence, measuring cryptographic performance via "time per module" yields distorted metrics. To provide a normalized baseline, we also evaluated the cryptographic processing cost by dividing the total execution time by the exact number of tagged physical blocks processed, as depicted in Figure 4.

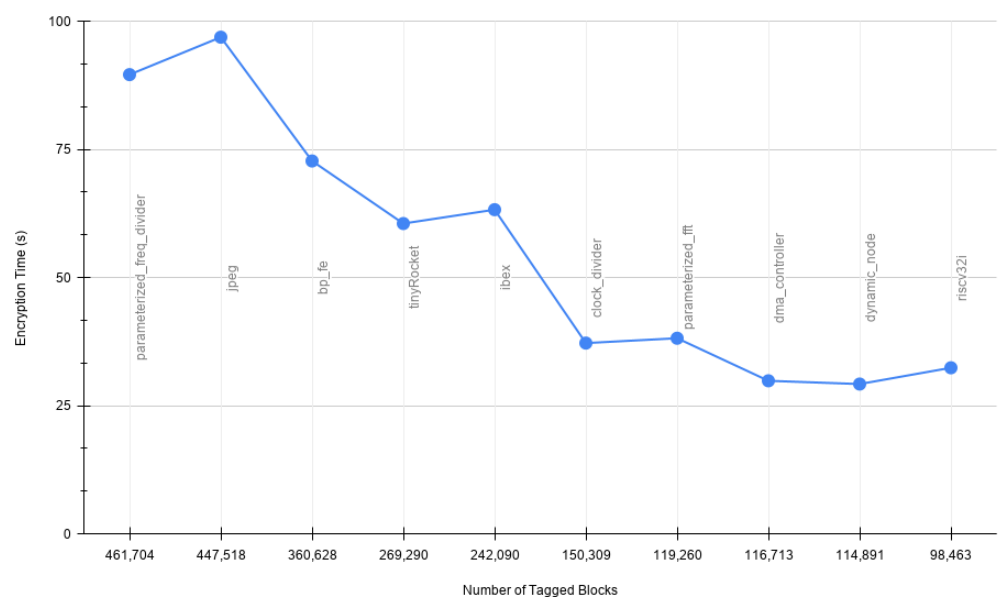


Figure 4. Encryption time (seconds) vs. number of tagged blocks in scenario 2.

The results demonstrate a strong linear correlation between the volume of encrypted geometry and the execution time. For example, processing the jpeg design required about 97 s for 447,518 blocks, equating to an approximate baseline cost of 0.216 milliseconds per block. While this scaling is predictable, minor fluctuations emerge based on each layout's composition. First, a tagged block is not computationally uniform, as complex polygons contain more vertices than standard cells. FF1 encrypts every coordinate pair; consequently, the cost depends on vertex density. This is supported by file sizes: the parameterized_fft design has fewer blocks than the clock_divider but requires more time to encrypt because its blocks are denser (about 154 against 139 bytes/block). Secondly, FF1 presents higher overhead when it converts alphanumeric string data (radix 72) compared to numeric (radix 10). This accounts for the processing spikes that are shown in the rest of the results, as they have more structure references (*SREF*) and string names (*SNAME*).

Despite these structural variations, the time cost remains bound to physical geometry and structural complexity, ensuring that the engine's overhead is light and predictable enough for EDA integration.

5.4. Encryption and Decryption Throughput

In production environments, the speed with which layout files are generated and used is paramount. We evaluated the overall data throughput of the system, measuring the speed at which our system can parse the files, encrypt, decrypt, and write the output.

During the full 3-level encryption of scenario 2, as shown in Table 4, our software achieved a consistent processing throughput of roughly 0.76 to 1.19 MB/s across all designs. Notably, during partial decryption passes, where higher-level encrypted modules are bypassed, the engine's throughput increases. When executing the lowest-level clearance decryption on parameterized_fft design, the read/write throughput reached 7.028 MB/s.

The exponential increase in throughput during lowest-level clearance decryption highlights the efficiency of the Look-Ahead Decryption Buffer and the selective bypass filter we implemented. Our engine does not waste CPU cycles attempting to decrypt modules that the user does not have access to. Instead, the bypassed ciphertext is streamed directly to the output file at high speeds. This proves that our framework can be usable, as the top-level designer can provide layout designs with variable access levels to others with minimal overhead. The same applies to the receivers of these files, as proven by the results.

Table 4. Throughput analysis (MB/SEC) across all designs in scenario 2.

Design Name	Enc.	Level 3 Dec.	Level 2 Dec.	Level 1 Dec.
jpeg	1.191	0.839	4.558	5.082
parameterized_freq_divider	1.168	0.733	1.116	1.235
bp_fe	1.192	0.789	3.670	4.738
tinyRocket	1.054	0.750	0.877	4.416
ibex	0.955	0.654	0.767	6.632
clock_divider	0.914	0.621	0.716	1.307
parameterized_fft	0.810	0.558	0.684	7.028
dynamic_node	0.983	0.741	0.870	5.916
dma_controller	0.935	0.689	0.772	2.127
riscv32i	0.763	0.544	0.733	5.743

5.5. N-Tier Scalability

To validate the structural scaling of our software, we conducted an additional experiment upon a layout design with different number of access tiers. Using the *jpeg* benchmark, the engine executed 10 independent runs across 3-tier, 5-tier and 10-tier configurations. For each run we measured encryption and decryption runtimes, as well as the resulting sizes of encrypted and decrypted files. It should be noted that for each configuration the number of modules to be encrypted is increasing, while in 3-tier access we used the configuration of the second scenario, encrypting 40 modules in total, in the 5-tier configuration the modules are 198, and in the 10-tier 200. For this reason, the number of affected blocks that are going to be encrypted is inherently increasing.

The results illustrated in Figure 5 confirm the efficiency of our single-pass architecture across varying access hierarchies. The encrypted file size increased from 115.4 (3-tier) to 120 MB (10-tier), and the encryption time ranged from 92.3 s to 104.5 s. This validates that the overall encrypted file size is governed by the total volume of geometric modules selected for encryption, rather than the number of access tiers or metadata tags. The standard deviation for the encryption phase across all 30 runs was approximately 1.3 s, proving the engine is deterministic to system resource fluctuations.

The decryption metrics, total runtime, and the accumulative output file size scale with the number of tiers. The decryption time rose from 134.3 s to 364.1, mirroring the combined storage footprint of the generated decrypted layouts. The same thing was noted for the decrypted file size, it rose from 281 (3 discrete files) to 1026.1 MB (10 files). This scaling is bound by physical I/O constraints as a 10-tier configuration requires the engine to actively generate and write over a gigabyte of layout files to disk. By isolating

this scaling to the final extraction phase, our architecture proves efficient and scalable across N-tier access configurations.

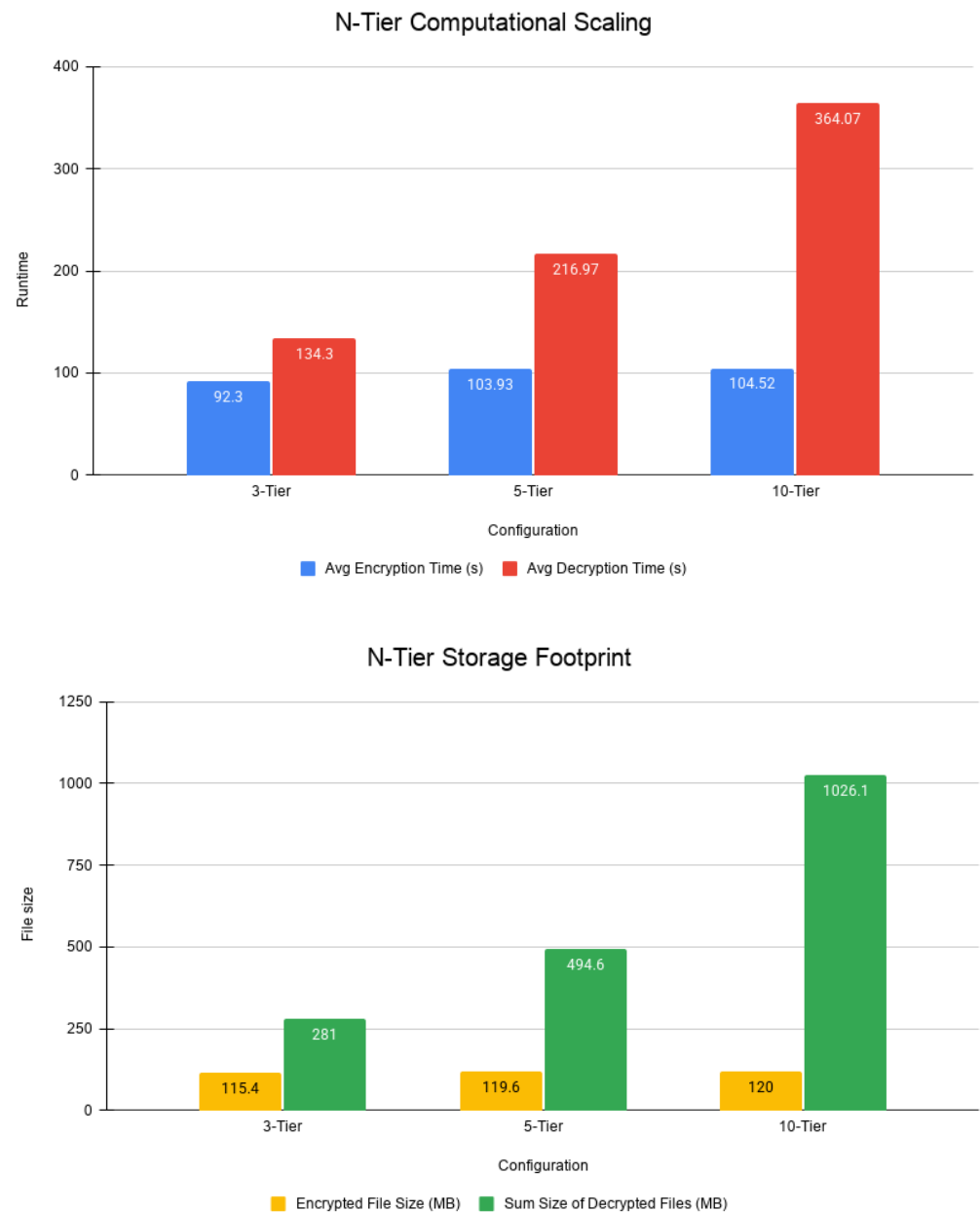


Figure 5. N-tier computational scaling and storage footprint.

5.6. Comparison with Nested Encryption Scheme

A common alternative for implementing multi-level security is a nested approach, where lower layers iteratively re-encrypt the ciphertext of higher layers. We utilized our normalized block-processing metrics to simulate the time cost of a nested architecture and compared it against our proposed single-pass, tagging method.

We calculated the runtime of encryption and decryption of the nested approach using $NT = T_1 + 2T_2 + 3T_3$, where NT is the nested time and T_i represents the baseline processing time for clearance tier i . Completing either procedure means that the lowest tier will encrypt/decrypt the highest a total of three times, the mid-tier two times, and its own tier once. Our results are depicted in Figure 6.

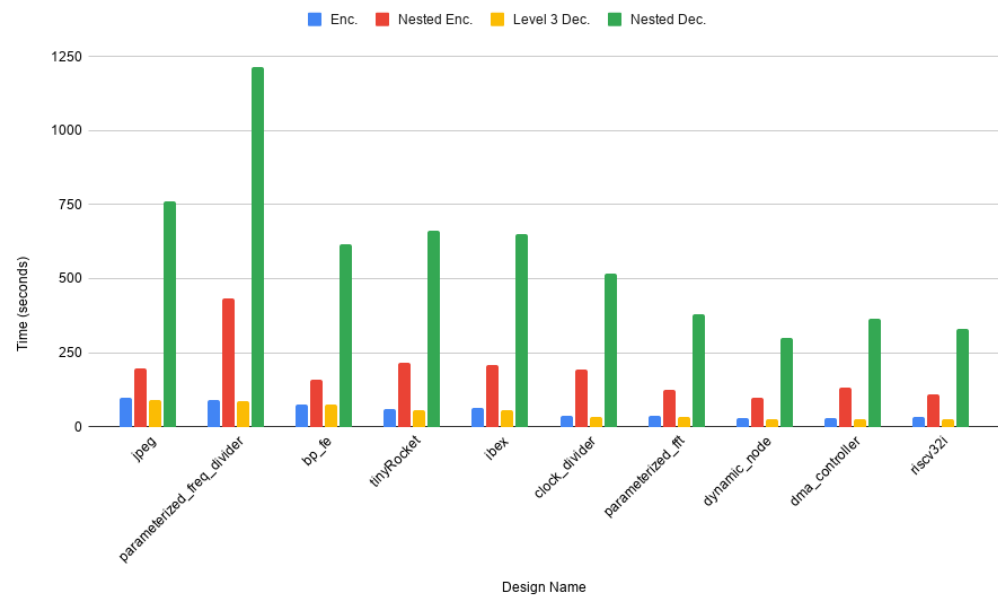


Figure 6. Flat vs. nested encryption and decryption time (seconds).

The results show a vast difference between the two methods. For jpeg design, our tagging architecture completed full encryption in 96.86 s and decryption in 91.09. The simulated nested method for the same layout would require 196.39 s to encrypt and 203.43 s to decrypt. The resulting data indicate that our tagging methodology over the nested is superior. Nested encryption forces the system to process identical geometric arrays multiple times per tier, therefore introducing an exponential runtime overhead, especially in decryption.

It should be noted that this comparison remains a theoretical complexity analysis based on algorithmic scaling. Due to the fact that our proposed software is made for a single-pass execution and native GDSII metadata tagging, developing an entirely separate nested encryption framework falls outside the scope of this work. Instead, the mathematical model presented effectively demonstrates the inherent computational bloat of the nested approach compared to our single-pass architecture.

6. Discussion and Future Work

In this paper, we introduce a practical implementation of FPE and tiered cryptographic access control directly within the IC layout domain. Our results confirm that the single-pass, tagging architecture scales linearly and avoids the computational overhead associated with nested encryption models. However, our evaluation also highlighted the security versus performance trade-off of this approach, which is the introduction of a moderate file size overhead. As FF1 guarantees zero cryptographic expansion, this storage penalty is generated by the metadata strings injected to enforce multi-tiered access.

While the results demonstrate the framework's capabilities, we must acknowledge certain limitations regarding its practical evaluation. Due to the closed-source nature of the semiconductor industry, evaluating the engine against active proprietary layouts or conducting industrial user studies was not feasible. The performance metrics were derived using open-source benchmark designs. However, the proposed scheme operates at a syntactic level and processes arrays of coordinates and structure references; thus, its computational overhead and file size expansion scale with polygon density and hierarchy depth, rather than functional complexity. As a result, we consider that these open-source layouts provide a structurally representative baseline for performance evaluation.

To address the data expansion introduced by the injected metadata tags, future work will focus on optimizing the metadata architecture. Exploring hierarchical tagging, where security properties are assigned to top-level structural references (SREF) rather than individual geometric boundaries, or implementing a dictionary-based tokenization scheme could compress the metadata footprint. Furthermore, extending this framework to the Open Artwork System Interchange Standard (OASIS) could be a next step. OASIS is the successor to GDSII, made to resolve file size bottlenecks of modern technological nodes through advanced data compression and modal variables. Adapting our software to navigate and secure OASIS format without breaking its strict syntax presents a relevant, yet complex, avenue for future research. The current version of our framework relies on a user-defined JSON file to map cell names to security levels. To automate this procedure, future work will explore the integration of graph neural networks (GNNs) or heuristic layout-analysis algorithms. By analyzing structural topology, component density and hierarchical naming conventions, such models could be trained to automatically differentiate sensitive modules (IP) from non-sensitive, thus generating the security configuration file dynamically while minimizing the user's workload.

Author Contributions: Conceptualization, A.N.D. and G.K.K.; methodology, A.N.D. and G.K.K.; software, G.K.K.; validation, G.S. and G.K.K.; formal analysis, G.S.; investigation, G.K.K.; resources, G.K.K. and T.L.; data curation, A.N.D. and T.L.; writing—original draft preparation, A.N.D., G.K.K. and G.S.; writing—review and editing, A.N.D. and T.L.; visualization, G.K.K. and G.S.; supervision, A.N.D.; project administration, A.N.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Our software, and benchmark testing scripts are openly available in the accompanying GitHub repository (<https://github.com/PlebeianDev/Securing-VLSI-Layouts-via-Format-Preserving-Encryption> accessed on 20 July 2026).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Zhang, Y.; Jain, A.; Cui, P.; Zhou, Z.; Guin, U. A novel topology-guided attack and its countermeasure towards secure logic locking. *J. Cryptogr. Eng.* **2021**, *11*, 213–226.
2. Calma Company. *GDSII Stream Format Manual, Release 6.0*; Calma Company: Milpitas, CA, USA, 1987.
3. National Institute of Standards and Technology. *Advanced Encryption Standard (AES)*; Federal Information Processing Standards Publication (FIPS) 197; U.S. Department of Commerce: Washington, DC, USA, 2001. [CrossRef]
4. Dworkin, M. *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*; NIST Special Publication (SP) 800-38A; National Institute of Standards and Technology: Gaithersburg, MA, USA, 2001. [CrossRef]
5. Dworkin, M.; Mouha, N. *Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption*; NIST Special Publication (SP) 800-38G Rev. 1 (2nd Public Draft); National Institute of Standards and Technology: Gaithersburg, MA, USA, 2025. [CrossRef]
6. Quadir, S.E.; Chen, J.; Forte, D.; Asadizanjani, N.; Shahbazmohamadi, S.; Wang, L.; Chandy, J.; Tehranipoor, M. A Survey on Chip to System Reverse Engineering. *J. Emerg. Technol. Comput. Syst.* **2016**, *13*, 1–34. [CrossRef]
7. Meade, T.; Zhang, S.; Jin, Y.; Zhao, Z.; Pan, D. Gate-level netlist reverse engineering tool set for functionality recovery and malicious logic detection. In *Proceedings of the International Symposium for Testing and Failure Analysis*; ASM International: Materials Park, OH, USA, 2016; Volume 81368, pp. 342–346.
8. Rezayati, M.H.; Amirany, A.; Moaiyeri, M.H.; Jafari, K. A new method for securing binary deep neural networks against model replication attacks using magnetic tunnel junctions. *Int. J. Inf. Secur.* **2025**, *24*, 52. [CrossRef]
9. Amirany, A.; Moaiyeri, M.H.; Jafari, K. Nonvolatile associative memory design based on spintronic synapses and CNTFET neurons. *IEEE Trans. Emerg. Top. Comput.* **2020**, *10*, 428–437. [CrossRef]
10. Adel, M.J.; Rezayati, M.H.; Moaiyeri, M.H.; Amirany, A.; Jafari, K. A robust deep learning attack immune MRAM-based physical unclonable function. *Sci. Rep.* **2024**, *14*, 20649. [CrossRef] [PubMed]

11. Potestad-Ordóñez, F.E.; Tena-Sánchez, E.; Acosta-Jiménez, A.J.; Jiménez-Fernández, C.J.; Chaves, R. Hardware countermeasures benchmarking against fault attacks. *Appl. Sci.* **2022**, *12*, 2443. [[CrossRef](#)]
12. Mazumder Shuvo, A.; Zhang, T.; Farahmandi, F.; Tehranipoor, M. FLAT: Layout-Aware and Security Property-Assisted Timing Fault-Injection Attack Assessment. *IEEE Trans. Very Large Scale Integr. (Vlsi) Syst.* **2024**, *32*, 1150–1163. [[CrossRef](#)]
13. Daemen, J.; Rijmen, V. *The Design of Rijndael*; Springer: Berlin/Heidelberg, Germany, 2002; Volume 2.
14. Bellare, M.; Kohno, T. A theoretical treatment of related-key attacks: RKA-PRPs, RKA-PRFs, and applications. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques*; Springer: Berlin/Heidelberg, Germany, 2003; pp. 491–506.
15. Massoudi, A.; Lefebvre, F.; De Vleeschouwer, C.; Macq, B.; Quisquater, J.J. Overview on selective encryption of image and video: Challenges and perspectives. *Eurasip J. Inf. Secur.* **2008**, *2008*, 179290. [[CrossRef](#)]
16. Sandhu, R.S.; Coyne, E.J.; Feinstein, H.L.; Youman, C.E. Role-based access control models. *Computer* **1996**, *29*, 38–47. [[CrossRef](#)]
17. Bellare, M.; Rogaway, P.; Spies, T. *The FFX Mode of Operation for Format-Preserving Encryption*; Submission to NIST; National Institute of Standards and Technology: Gaithersburg, MA, USA, 2010.
18. National Institute of Standards and Technology. *Secure Hash Standard (SHS)*; Federal Information Processing Standards Publication (FIPS) 180-4; U.S. Department of Commerce: Washington, DC, USA, 2015. [[CrossRef](#)]
19. Gu, A.; Zakhor, A. Lossless Compression Algorithms for Hierarchical IC Layout. *IEEE Trans. Semicond. Manuf.* **2008**, *21*, 285–296. [[CrossRef](#)]
20. Eijs, H. PyCryptodome: A Self-Contained PYTHON Package of Low-Level Cryptographic Primitives. Available online: <https://www.pycryptodome.org> (accessed on 12 March 2026).
21. Köfferlein, M. KLayout: Your Mask Layout Friend. Available online: <https://www.klayout.de> (accessed on 3 March 2026).
22. SJTU YONGFU RESEARCH GRP. PDB-Physical-Design-Database: A Comprehensive Database and Toolset for Physical Design Exploration. 2024. Available online: <https://github.com/SJTU-YONGFU-RESEARCH-GRP/PDB-Physical-Design-Database> (accessed on 21 April 2026).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.