A Unified Topological Framework for System Abstraction via Reverse Engineering

Introduction: From Low-Level Artifacts to High-Level Abstractions

1.1 The Fundamental Problem of System Comprehension

In the domains of software and hardware engineering, a persistent and critical challenge is the comprehension of complex systems for which original design documentation is unavailable, incomplete, or has become untrustworthy over time.[1] As systems evolve, modifications are often made without corresponding updates to the specifications, leading to a growing gap between the documented design and the actual implementation.[3] This lack of adequate and up-to-date documentation, often a result of resource constraints or the departure of original developers, creates a significant barrier to maintenance, modernization, security analysis, and innovation.[1]

Reverse engineering (RE) emerges as the primary methodology to bridge this comprehension gap. It is formally defined as the process of deconstructing an existing man-made product, device, or system to understand its design, components, and functionality through deductive reasoning.[4] This process involves taking an engineered artifact apart, back to its elemental components, to extract design information and recover the underlying principles of its operation.[6] The motivations for undertaking this process are diverse, ranging from the recovery of legacy designs and software maintenance to competitive analysis, patent infringement detection, and cybersecurity.[1] For instance, a security analyst may reverse engineer malware to understand its attack vectors and develop countermeasures, while a company might analyze a competitor's product to inform its own development cycle.[1] Similarly,

reverse engineering is indispensable for modernizing legacy systems where the original documentation or developers are no longer available, allowing for safe migration and evolution.[8] All these varied applications converge on a singular, fundamental need: to reconstruct an abstract, comprehensible model of a system's architecture and behavior from its opaque, low-level implementation.

1.2 The Role of Abstraction in Reverse Engineering

The ultimate goal of reverse engineering is not merely to deconstruct but to reconstruct understanding at a higher level of abstraction. Abstraction is the cognitive and analytical process of simplifying complexity by systematically ignoring irrelevant details to focus on the essential characteristics of a system.[10] In the context of reverse engineering, this involves creating a series of simplified, high-level models from complex, low-level data, making it easier to analyze and comprehend the system in a step-wise fashion.[10]

This process is typically conceptualized as a hierarchy of abstraction levels. At the lowest level is the implementation, such as machine code in software or a gate-level netlist in hardware—a "sea of gates" that is structurally complete but functionally incomprehensible.[11] The reverse engineering process aims to ascend this hierarchy:

- **Low-Level Abstraction:** Focuses on the most granular details, such as assembly language, memory allocation, and individual logic gates.[11]
- **Intermediate-Level Abstraction:** Represents the system in terms of its constituent components and modules, detailing interfaces, data structures, and control flow between them.[11] This level corresponds to design recovery, where domain knowledge is used to identify meaningful higher-level abstractions beyond what is directly observable in the code or netlist.[11]
- **High-Level Abstraction:** Provides an overview of the system's overall architecture and functionality, often visualized through diagrams, flowcharts, and other representations that capture the system's conceptual design and requirements.[11]

The successful navigation of these levels, from the concrete implementation to the abstract specification, is the central objective of any reverse engineering effort. It is this process of abstraction that transforms a raw collection of technicals into actionable intelligence.

1.3 Thesis: A Unified Topological Approach

While software and hardware reverse engineering are often treated as distinct disciplines, their fundamental workflows exhibit a profound structural parallel. Both begin with a low-level, often linear or unstructured, artifact (a binary file, a physical chip) and proceed by extracting a graph-based representation of its internal structure (a control flow graph, a gate-level netlist). This graph, or topology, then becomes the primary object of analysis for deriving higher-level abstractions. This methodological convergence suggests the possibility of a unified framework for analysis.

This report posits that a single, mathematically rigorous methodology—Topological Data Analysis (TDA)—can be applied to the structural representations derived from both software and hardware to achieve system abstraction in a principled and repeatable manner. TDA is a field of applied mathematics that provides tools to analyze the "shape" of data, identifying robust structural features that are insensitive to noise and the specific choice of metrics.[13] By treating the graphs extracted from reverse engineering as complex datasets, TDA, and particularly its core tool of persistent homology, can be used to identify significant structural patterns such as modules, clusters, and feedback loops.[13] This "exact same topological approach" provides a domain-agnostic framework for quantifying and simplifying system topology, thereby offering a powerful new method for recovering design intent from the bottom up.

Extracting Topological Structures from Software Binaries

2.1 The Graph as a Foundational Abstraction

The initial and most critical step in the reverse engineering of software is the transformation of the linear, sequential representation of machine code into a structured, graph-based format. Raw source code or disassembled binary is often challenging to reason about in terms of execution paths.[15] A graph-based representation, by contrast, makes the control flow explicit, providing the visual and logical structure necessary to understand the program's behavior at a higher level.[15] In this paradigm, every relevant entity within the software artifact, such as a block of code or a function, is modeled as a vertex (node), and the relationships between these entities are modeled as edges.[16] This transformation from a one-dimensional stream of instructions to a two-dimensional graph is the foundational act of abstraction upon

which all subsequent analysis is built.

2.2 Intra-procedural Topology: The Control Flow Graph (CFG)

The most fundamental topological representation of a program's logic within a single function or procedure is the Control Flow Graph (CFG).

Definition and Components

A CFG is a directed graph that represents all possible paths that might be traversed through a program during its execution.[15] The two main components of a CFG are:

- **Nodes:** These represent *basic blocks*. A basic block is a straight-line sequence of code with no branches in except to the entry point and no branches out except at the exit.[17] The flow of control enters at the beginning of a basic block and leaves at the end without any possibility of halting or branching in the middle.[17]
- **Edges:** These represent control flow transfers between basic blocks. An edge from Block A to Block B indicates that execution can pass directly from the last instruction of Block A to the first instruction of Block B. This can occur via a conditional jump (e.g., je), an unconditional jump (e.g., jmp), a function call, or by sequential execution ("fall-through").[15]

Generation from Compiled Code

When source code is unavailable, which is the typical scenario in reverse engineering, CFGs must be reconstructed directly from the compiled binary. This process is automated by tools like IDA Pro and Ghidra and involves the following steps [15]:

1. **Disassembly:** The machine code is first converted into a human-readable assembly language representation.
2. **Leader Identification:** The instruction stream is partitioned into basic blocks by identifying "leaders," which are the first instructions of a basic block. According to standard algorithms, a leader is defined as: the very first instruction of a function; any instruction that is the target of a conditional or unconditional jump; or any instruction

that immediately follows a jump or call instruction.[17]
3. **Block Construction:** A basic block consists of a leader and all subsequent instructions up to, but not including, the next leader.[17]
4. **Edge Creation:** Edges are added between blocks based on the semantics of the last instruction in each block. For example, a conditional jump instruction je target at the end of Block A would create two edges: one to the basic block at target and one to the sequentially next basic block (the fall-through case).[15]

This reconstructed CFG serves as a precise map of the low-level control logic within each function of the program.

2.3 Inter-procedural Topology: The Call Graph

While the CFG details the logic *within* a function, the Call Graph provides a higher-level view of the relationships *between* functions.

Definition and Significance

A call graph is a control-flow graph where each node represents a procedure (a function or subroutine) and a directed edge from node f to node g indicates that procedure f calls procedure g.[19] This graph is essential for understanding the overall architecture of a program, identifying dependencies between different parts of the code, and tracing the flow of execution at a macro level. A cycle in the call graph indicates a recursive relationship.[19]

Static vs. Dynamic Extraction

Call graphs can be generated using two distinct approaches, each with a fundamental trade-off between completeness and precision.[2]

- **Static Analysis:** A static call graph is generated by analyzing the binary code without executing it.[19] The goal is to represent *every possible* run of the program. This method is comprehensive and aims for soundness, meaning it should not miss any potential call relationship.[20] However, because determining the exact static call graph is an undecidable problem, static analysis algorithms must be conservative and typically

produce an *over-approximation*.[19] This means the graph may include edges for calls that would never occur in any actual execution, introducing imprecision or "false positives".[19] This problem is particularly acute in languages with features like function pointers, dynamic dispatch (virtual function calls in C++), and polymorphism, where the exact target of a call cannot be determined without sophisticated alias analysis.[19]

- **Dynamic Analysis:** A dynamic call graph is a record of an actual execution of the program, often generated by a profiler or a debugger.[19] This approach is exact and highly precise for the specific execution path that was observed; it contains no false positives.[19] However, it is fundamentally *incomplete* or unsound, because it only captures the behavior elicited by the specific inputs used for that run.[2] Any code paths not exercised during the analysis will be absent from the resulting graph.[2]

The Hybrid Approach

The inherent tension between the soundness of static analysis and the precision of dynamic analysis necessitates a hybrid approach for the most accurate results. Modern reverse engineering workflows often start with a comprehensive but imprecise static analysis and then use the results of targeted dynamic analysis to confirm or refute the existence of certain call edges.[2] For example, if static analysis identifies a call through a function pointer with ten potential targets, dynamic analysis might reveal that in practice, only two of those targets are ever called. This allows the analyst to prune the static graph, creating a more accurate and manageable model that combines the strengths of both techniques.[21]

2.4 System-Level Topology: Component Dependency Graphs

The highest level of software topology is represented by component dependency graphs, which visualize the complex web of interactions between a system's modules, libraries, and frameworks.[26]

Definition and Importance

In a software dependency graph, nodes represent high-level components (e.g., a Python package, a Java JAR file, a dynamic-link library), and directed edges represent the

dependencies between them.[28] These graphs are crucial for understanding the overall software architecture, especially in modern systems that are rarely monolithic and often consist of thousands of interconnected components.[27] Analyzing these dependencies is vital for several reasons [26]:

- **Impact Analysis:** Understanding how a change in one component will affect downstream dependencies.
- **Vulnerability Management:** Identifying security risks introduced by third-party libraries, including both direct and transitive dependencies.
- **System Comprehension:** Gaining a clear view of all components involved in a complex application, including open-source software.

Types of Dependencies

The edges in a dependency graph can represent several types of relationships, each with different implications [27]:

- **Direct Dependencies:** Components that a module explicitly references in its code, such as libraries included via an import statement. These are typically listed in a project's manifest file (e.g., package.json, requirements.txt).
- **Transitive Dependencies:** Indirect dependencies that are required by a project's direct dependencies. These are often managed automatically by package managers and can introduce hidden complexities and vulnerabilities.
- **Compile-time Dependencies:** Components required to build or compile the project, such as compilers or code generation tools, which are not needed at runtime.

Analyzing this system-level topology provides the broadest view of the software's structure and its place within a larger ecosystem of dependencies.

Extracting Topological Structures from Integrated Circuits

3.1 The Physical-to-Logical Challenge

Hardware reverse engineering (HRE) is the process of translating a physical, manufactured Integrated Circuit (IC) back into a logical, circuit-diagram-like representation known as a netlist.[12] Unlike software RE, which is a virtual process, HRE is a costly, time-consuming, and often destructive endeavor that requires specialized laboratory equipment to probe the physical artifact.[30] The objective is to extract the ground-truth topology of the hardware, revealing the precise interconnection of all logic gates on the chip.[30] This process is essential for detecting hardware Trojans, analyzing for patent infringement, or recovering designs for legacy components.[12]

3.2 Stage 1: Physical Deconstruction and Imaging

The first phase of HRE involves systematically deconstructing the IC to visually access its internal structure. This is a delicate, multi-step process.

Decapsulation and Delayering

The process begins with **decapsulation**, where the protective packaging (plastic or ceramic) is removed to expose the bare silicon die.[29] This is typically achieved using corrosive acids that are carefully selected to etch away the packaging material without damaging the underlying circuitry.[29] Once the die is exposed, the **delayering** process commences. This involves a layer-by-layer etching of the die, removing the metal interconnect and dielectric layers one at a time to reveal the transistor-level structures beneath.[29] This requires immense precision, as modern ICs can have over a dozen metal layers with feature sizes in the nanometer range.[29]

Advanced Microscopy

High-resolution imaging is the cornerstone of HRE. After each layer is removed, advanced imaging technologies are used to capture detailed photographs of the exposed circuitry. The primary tools for this task are [29]:

- **Scanning Electron Microscopy (SEM):** An SEM scans the surface of the die with a

focused beam of electrons, providing high-resolution grayscale images of the topography. It is invaluable for visualizing the layout of transistors and the interconnects (wires) that connect them.[31] In a technique known as Passive Voltage Contrast (PVC), an SEM can even be used to distinguish the state of memory cells (e.g., '0' vs. '1') in non-volatile memory like Flash EEPROM by detecting differences in surface charge.[35]

- **Transmission Electron Microscopy (TEM):** For even higher resolution, a TEM can be used. This requires preparing an extremely thin cross-section of the chip and passing an electron beam through it, allowing for the study of the vertical structure of the device, such as the thickness of the gate oxide layer or the profile of different material layers.[34]

This imaging process is repeated for every layer of the IC, generating a massive dataset of high-resolution images that collectively document the complete physical layout of the chip.

3.3 Stage 2: Netlist Extraction and Representation

The second phase of HRE involves translating the collected image data from a geometric representation into a logical one.

Image Processing and Circuit Extraction

The series of images from each layer must first be digitally aligned with extreme precision. Specialized software tools are then used to perform **circuit extraction**. This software analyzes the composite images to automatically identify transistors, logic gates, and their interconnections based on their known visual patterns and material properties.[29] The process involves recognizing designed devices (like transistors) and parasitic devices (unintended but inherent capacitances and resistances), and tracing the conductive paths (interconnects) that wire them together.[36] This automated process converts the visual data into a formal description of the circuit's connectivity.[36]

The Gate-Level Netlist

The output of the circuit extraction process is the **gate-level netlist**. This is the definitive topological representation of the IC—a directed graph where the nodes are fundamental logic

gates (e.g., NAND, NOR, flip-flops) and other circuit elements, and the edges are the wires that connect their inputs and outputs.[12] This flat netlist is the hardware equivalent of raw, disassembled code. It contains the complete structural information of the circuit but lacks any higher-level functional or hierarchical context, presenting the reverse engineer with an undifferentiated "sea of gates" that can consist of millions of elements.[12]

3.4 Higher-Level Hardware Abstraction

The final and most challenging phase of HRE is the analysis of the flat netlist to recover the design's functional architecture. This is an act of abstraction, aiming to group the millions of individual gates into a hierarchy of meaningful, high-level structures.[12] Techniques used in this phase include:

- **Dataflow Analysis:** Analyzing the flow of data between storage elements (like flip-flops) to identify and group them into larger structures such as multi-bit registers.[12]
- **Topological and Structural Pattern Matching:** Searching the netlist graph for subgraphs that correspond to known digital logic building blocks, such as adders, multiplexers, or finite state machines (FSMs).[37]
- **Register Categorization:** Differentiating between registers that are part of the datapath (which process data) and those that are part of the control logic (which dictate the circuit's state and operation). This can be achieved by analyzing the topological similarity of the logic surrounding different registers, as registers within the same datapath often share similar fan-in logic structures.[37]

Successfully performing this abstraction is the key to moving from a structural description of *what the circuit is* to a functional understanding of *what the circuit does*.

Table 1: Comparison of Topological Representations in Software and Hardware

Representation	Domain	Nodes Represent	Edges Represent	Granularity	Primary Extraction Method	Key Analytical Challenge
Control Flow Graph	Software	Basic Blocks (sequenti	Control Flow Transfer	Intra-procedural	Static Disassembly and	Resolving targets of indirect

(CFG)		al instructions)	(jumps, calls, fall-through)		Analysis of Compiled Code [15]	jumps and function pointers.[3] [8]
Call Graph	Software	Functions / Subroutines	A call from one function to another	Inter-procedural	Static Analysis (over-approximated) or Dynamic Analysis (incomplete) [2]	Achieving both soundness and precision, especially with polymorphism and dynamic dispatch.[1] [9]
Component Dependency Graph	Software	Modules, Libraries, Frameworks	A dependency of one component on another (e.g., import)	System-level	Analysis of project manifests, build files, and import statements [26]	Tracking and managing complex webs of transitive (indirect) dependencies.[27]
Gate-Level Netlist	Hardware	Logic Gates (NAND, NOR, etc.) and Flip-Flops	Physical Wires connecting gate inputs and outputs	Gate-level	Physical Imaging (SEM/TEM) followed by automated Circuit Extraction [29]	Grouping millions of gates into meaningful, high-level functional blocks (e.g., registers,

						FSMs).[12]

A Methodological Framework: Topological Data Analysis for System Abstraction

4.1 Introduction to Topological Data Analysis (TDA)

Motivation

Having established that both software and hardware reverse engineering produce complex graph structures, the central challenge becomes how to analyze these graphs to extract meaningful, high-level abstractions. Traditional graph algorithms can be brittle, while manual inspection is intractable for large systems. Topological Data Analysis (TDA) offers a novel and powerful alternative. TDA is a field of applied mathematics that uses techniques from algebraic topology to study the "shape" of data.[13] Its primary motivation is to provide a framework for analyzing datasets that are high-dimensional, noisy, and incomplete—characteristics that are highly representative of the data derived from reverse engineering.[13]

The key advantages of TDA are its coordinate-free nature and its robustness. It provides insights that are insensitive to the particular metric chosen to compare data points and is inherently robust to noise, making it well-suited for identifying the essential structure within imperfectly recovered system graphs.[13] TDA is premised on the idea that the underlying shape of a dataset contains relevant information, and it provides mathematically rigorous tools to quantify this shape.[13]

Core Concepts

At the heart of TDA are concepts from algebraic topology, which studies properties of spaces

that are preserved under continuous deformations. The main objects of study are:

- **Simplicial Complexes:** These are generalizations of graphs. A 0-simplex is a point (vertex), a 1-simplex is an edge, a 2-simplex is a filled triangle, a 3-simplex is a solid tetrahedron, and so on. A simplicial complex is a collection of simplices glued together along their faces, providing a way to build a high-dimensional "shape" on top of a set of data points.[14]
- **Homology:** This is an algebraic tool for counting the "holes" in a topological space. The 0-dimensional homology group (H_0) counts the number of connected components. The 1-dimensional homology group (H_1) counts the number of loops or tunnels. The 2-dimensional homology group (H_2) counts the number of voids or cavities, and so on.[13] These counts, known as Betti numbers, provide a quantitative signature of a shape's topology.

4.2 Persistent Homology: Capturing Features Across Scales

A single, static analysis of a dataset's topology is often insufficient, as the choice of a connectivity parameter can drastically change the perceived structure. The main insight of TDA is to analyze the topology across *all* possible scales simultaneously using a technique called persistent homology.[13]

The Filtration Process

Persistent homology works by building a **filtration**, which is a nested sequence of simplicial complexes built on top of the data. Imagine a point cloud where each point is the center of a growing ball. As the radius of these balls increases, they begin to intersect. A simplicial complex (specifically, a Vietoris-Rips complex) can be constructed at each radius value t: an edge (1-simplex) is added between two points if their balls of radius $t/2$ intersect, a triangle (2-simplex) is added if three points are pairwise connected, and so on.[14] As t increases, new components merge, loops form, and loops get filled in. This process creates a sequence of shapes, one for every possible scale.[13]

Persistence Barcodes and Diagrams

Persistent homology tracks the "birth" and "death" of topological features (components, loops, voids) throughout this filtration process. The lifetime of each feature can be visualized in two equivalent ways [13]:

- **Persistence Barcode:** Each topological feature is represented by a horizontal bar. The bar's start point is the scale at which the feature first appears (its "birth"), and its end point is the scale at which it disappears (its "death," e.g., when a loop is filled in or two components merge).
- **Persistence Diagram:** Each feature is represented by a point (b, d) in a 2D plot, where b is the birth time and d is the death time.

The crucial insight of persistent homology is that features with long lifetimes (long bars in the barcode, or points far from the diagonal $y=x$ in the diagram) are considered robust, significant features of the data's underlying shape. Features with short lifetimes are treated as topological noise.[13] This provides a principled way to distinguish signal from noise in a complex structural dataset.

4.3 A Proposed TDA Pipeline for Reverse Engineering

To apply TDA to the graphs extracted from reverse engineering, the following methodological pipeline can be employed:

Step 1: Graph to Point Cloud Conversion

The first step is to represent the nodes of a graph (e.g., basic blocks in a CFG, functions in a call graph, or gates in a netlist) as points in a metric space. This requires defining a meaningful **distance function** $d(x, y)$ between any two nodes x and y in the graph.[14] The choice of this metric is critical as it determines the geometric embedding of the graph. A simple and common choice is the shortest path distance within the graph. More sophisticated metrics could incorporate semantic information, such as the similarity of instructions in basic blocks or the data dependencies between functions.

Step 2: Building the Filtration

With the graph nodes now represented as a point cloud with a defined distance metric, a filtration can be constructed. The standard approach is to build a **Vietoris-Rips complex**. For a continuously increasing scale parameter t, a k-simplex is included in the complex if all k+1 of its vertices are pairwise within distance t of each other.[14] This process generates the nested family of simplicial complexes required for persistent homology.

Step 3: Computing Persistent Homology

Using standard TDA software libraries, the persistence barcode or diagram is computed from the filtration. This calculation, while conceptually complex, is handled by efficient algorithms that output the birth and death times of all topological features.[13]

Step 4: Interpreting Topological Signatures for Abstraction

This final step involves mapping the mathematical results from the persistence diagram back to the original engineering context to drive the abstraction process.

- **H_0 (Connected Components):** The persistence of 0-dimensional features reveals the clustering structure of the graph. A small number of long bars indicates that the graph has a few very stable, well-separated clusters of nodes. In a call graph, these persistent components would correspond to highly cohesive software modules. In a gate-level netlist, they could identify distinct functional units on the chip. This provides a data-driven method for modularization.[14]
- **H_1 (Loops):** The persistence of 1-dimensional features identifies significant cyclic structures. In a CFG, a long bar in H_1 would correspond to a robust, possibly complex, control loop. In a call graph, it would indicate a significant chain of recursive or mutually recursive function calls. These features often represent core algorithmic or state-machine logic within the system.[14]
- **The Mapper Algorithm:** As a complementary tool, the Mapper algorithm can be used for visualization and exploration. Mapper produces a simplified, one-dimensional graph (a nerve) that summarizes the topology of the high-dimensional point cloud. Each node in the Mapper graph represents a cluster of similar points from the original data, and edges connect nodes that share points. This provides a high-level, intuitive "skeleton" of the system's structure, making it easier to identify global patterns and relationships.[14]

4.4 Application to Malware Analysis

The utility of TDA is not merely theoretical. In the field of cybersecurity, TDA has been successfully applied to malware analysis. By representing malware binaries based on their static or dynamic features (e.g., opcode sequences, API calls), TDA can be used to analyze the resulting point cloud.[40] Studies have shown that the TDA Mapper algorithm is highly effective at clustering malware families and identifying hidden relationships between them, outperforming traditional methods like PCA.[40] Furthermore, persistent homology (via persistence diagrams) is adept at identifying malware samples that belong to multiple classes simultaneously (e.g., a threat that is both a ransomware and a spyware), a task that is difficult for standard classifiers.[40] This demonstrates TDA's power to uncover complex, multi-scale patterns in reverse-engineered artifacts.

The reason TDA is so well-suited for reverse engineering lies in the nature of the engineering process itself. The journey from a high-level design concept to a low-level implementation is a complex, information-destroying process. The architect's clean, modular design is inevitably obscured by implementation details, cross-cutting concerns, and, most significantly, the transformative effects of compilation and synthesis. The resulting binary or netlist is a noisy, distorted reflection of the original intent. Reverse engineering is therefore an ill-posed inverse problem: attempting to recover the original input from a transformed output.[41] TDA, with its inherent robustness to noise and its focus on identifying features that *persist* across multiple scales, provides a mathematical framework for finding the true signal—the resilient aspects of the original design—that has survived this noisy, information-destroying forward process.

Confounding Factors: The Impact of Transformation and Obfuscation on Topological Integrity

5.1 The Observer Effect: How Analysis is Complicated by Code Transformation

A significant challenge in any topological analysis of a reverse-engineered system is that the artifact being analyzed is rarely a direct representation of the original source code or design. It is almost always a transformed version, altered by both benign and malicious processes that

can fundamentally change its structure. Benign transformations, such as compiler optimizations, restructure code to improve machine performance, while malicious transformations, known as obfuscation, restructure code to thwart human and automated analysis.[42] Both processes can severely distort or destroy the meaningful topological features that an analyst seeks to recover, creating a significant "observer effect" where the act of preparing the code for execution makes its original logic harder to observe.

5.2 Compiler Optimizations as Topological Distortion

The primary goal of a modern optimizing compiler is to generate machine code that is smaller, faster, and more energy-efficient, while strictly preserving the program's observable behavior.[43] To achieve this, compilers employ a vast array of aggressive transformations that operate directly on the program's graph representations, such as the CFG. While these transformations are beneficial for performance, they are often detrimental to comprehensibility from a reverse engineering perspective.[43]

Specific Optimizations and Their Topological Impact

- **Function Inlining:** This optimization replaces a function call with the body of the called function. From a topological perspective, this removes a node (the called function) and its associated edges from the call graph, merging its CFG directly into the CFG of the caller.[42] This obscures the original modularity of the design, making the caller's CFG larger and more complex, and makes it harder to identify reusable components.[42]
- **Loop Unrolling:** To reduce the overhead of loop control instructions (counter increments and conditional branches), a compiler may unroll a loop, duplicating its body multiple times.[44] This transforms the cyclic structure of the loop in the CFG into a long, linear sequence of basic blocks, fundamentally altering its topology and making the iterative nature of the algorithm less apparent.
- **Strength Reduction:** This involves replacing a computationally expensive instruction with an equivalent sequence of cheaper instructions. A classic example is replacing integer division by a constant with a series of multiplications, additions, and bit shifts.[46] This transformation increases the complexity of individual basic blocks and can make simple arithmetic operations appear as convoluted and unintuitive sequences of instructions to a reverse engineer.
- **Predicated Execution:** Some architectures allow instructions to be conditionally executed based on the value of a predicate or guard register. Compilers can use this

feature to eliminate conditional branches entirely, transforming a control dependency into a data dependency.[48] For example, instead of a branch that chooses between two paths, the compiler might generate code where instructions for both paths are present, but only one set is enabled for execution based on a predicate. This makes the CFG appear artificially linear and simple, as the true branching logic is hidden within the data flow of the predicate registers, seriously deteriorating the precision of traditional CFG reconstruction.[48]

5.3 Obfuscation as a Direct Attack on Topological Analysis

While compiler optimizations distort topology as a side effect of improving performance, obfuscation techniques are designed with the explicit adversarial goal of destroying a program's structural and semantic clarity to make reverse engineering as difficult as possible.[49] Many of these techniques are direct attacks on the integrity of the topological representations used by analysts.[52]

Key Obfuscation Techniques

- **Control Flow Flattening:** This is one of the most potent anti-topological techniques. It systematically destroys the natural, hierarchical control flow of a program (e.g., nested loops and conditionals). It replaces this structure with a single, large dispatcher loop containing a switch statement. Each original basic block becomes a case within the switch, and at the end of each block, control returns to the central dispatcher, which determines the next block to execute. This transforms a meaningful, structured CFG into a flat, "hub-and-spoke" topology where every logical block is connected only to the central dispatcher, revealing nothing about the program's actual logic flow.[53]
- **Opaque Predicates:** This technique involves inserting conditional branches into the code whose outcome is always known to the obfuscator but is computationally difficult for a static analyzer to determine. For example, a branch might depend on a complex mathematical identity that always evaluates to true. This adds spurious nodes and edges to the CFG, creating a maze of feasible-looking but never-executed paths that can confuse both automated analysis tools and human analysts.[53]
- **Code Virtualization:** This advanced technique replaces the original native instruction set of a program with a custom, unpublished bytecode language. The malware or protected application is then shipped with an embedded virtual machine (VM) or interpreter for this custom language. From a topological perspective, this is a devastating transformation.

The entire original CFG and call graph of the application are replaced by the topology of the VM's main interpreter loop. The analyst is left to reverse engineer not only the application's logic but the entire architecture of the custom VM it runs on, a significantly more complex task.[53]

Table 2: Impact of Transformations on Topological Features

Transformation Type	Specific Technique	Target Graph(s)	Effect on Graph Structure	Implication for Topological Analysis
Compiler Optimization	Function Inlining	Call Graph, CFG	Merges child function's CFG into parent's CFG; removes edge from Call Graph.[42]	Obscures original modularity and design intent; increases complexity of individual function graphs.
Compiler Optimization	Loop Unrolling	CFG	Replaces a cyclic subgraph with a longer, acyclic sequence of duplicated blocks.[44]	Hides the iterative nature of the algorithm; makes loop identification more difficult.
Compiler Optimization	Predicated Execution	CFG	Eliminates conditional branch edges, transforming control dependencies into data dependencies.[48]	Creates an artificially simple CFG that does not reflect the true conditional logic, defeating standard control flow

				analysis.
Obfuscation	Control Flow Flattening	CFG	Destroys hierarchical structure, replacing it with a central dispatcher node connected to all other blocks.[53]	Completely eradicates meaningful topological signatures (loops, components), rendering TDA and manual analysis ineffective.
Obfuscation	Opaque Predicates	CFG	Inserts spurious nodes and edges corresponding to branches that are never taken at runtime.[53]	Pollutes the CFG with false paths, increasing its complexity and misleading static analysis tools.
Obfuscation	Code Virtualization	CFG, Call Graph	Replaces the entire program graph with the graph of the embedded VM's interpreter loop.[53]	Hides the original program's topology entirely, requiring a much more difficult reverse engineering of the VM itself.

5.4 Resilience of the TDA Framework

The TDA framework, particularly persistent homology, is designed to be robust against local

perturbations and noise. It could potentially see through minor obfuscations like dead code insertion, which might appear as short-lived, noisy features in a persistence diagram. However, the large-scale, global topological transformations employed by advanced obfuscation present a more fundamental challenge. A technique like control flow flattening does not add noise to the existing topology; it replaces it entirely with a new, artificial topology. This new structure would have its own topological signature (a single, highly dominant connected component and very few persistent loops), but this signature would describe the obfuscation scheme, not the original program logic.

This reveals a critical limitation: a purely structural analysis like TDA is vulnerable to adversarial manipulations of that structure. In an adversarial context, such as malware analysis or intellectual property protection, topological analysis cannot be applied naively. It must be preceded by a de-obfuscation phase that attempts to reverse these structural transformations and recover a semblance of the original program graph. The analysis of network topology obfuscation schemes, which aim to mislead traffic analysis by manipulating network paths, provides a parallel domain where the goal is to maximize topological distortion to defeat inference.[54] This underscores the fact that topology is both a powerful tool for analysis and a vulnerable target for attack.

Practical Implementation and Tooling

6.1 The Reverse Engineer's Toolkit: An Overview

Bridging the gap between the theoretical framework of topological analysis and the practical task of reverse engineering requires a sophisticated ecosystem of software and hardware tools. This toolkit enables the two core phases of the process: the extraction of topological data from physical or binary artifacts, and the subsequent analysis of that data to build abstract models. The most powerful modern platforms are characterized by their extensibility, which provides the crucial link for integrating advanced analytical methods like TDA into the standard reverse engineering workflow.

6.2 Software Analysis Platforms: IDA Pro and Ghidra

The industry standards for interactive software reverse engineering are Hex-Rays' IDA Pro and the NSA's open-source Ghidra. Both are comprehensive platforms that combine disassemblers, decompilers, and a suite of analysis tools within an integrated environment.[57] Their capabilities for topological analysis are particularly noteworthy.

IDA Pro

IDA Pro is renowned for its powerful and mature feature set, especially its interactive graph visualization capabilities.[58]

- **Interactive Graph View:** IDA's default view for functions is a highly interactive CFG. Edges are color-coded to distinguish control flow paths: green for taken conditional branches, red for not-taken branches (fall-through), and blue for unconditional jumps.[18] This provides immediate visual cues about the program's logic.
- **Graph Manipulation:** Users can manually rearrange the graph layout, and more importantly, group multiple nodes into a single representative node. This feature is essential for manual abstraction, allowing an analyst who has understood a section of code to collapse it, simplifying the graph to focus on other areas. The SimplifyGraph plugin automates this process by identifying and grouping structurally related subgraphs.[60]
- **Xref Graph:** A newer feature, the Xref Graph, allows for the visualization of cross-reference relationships, enabling the creation of custom call graphs or data dependency graphs starting from any point in the binary.[62]
- **Extensibility:** IDA's power is significantly enhanced by its IDAPython scripting API and C++ SDK, which provide programmatic access to the internal database, including the CFG and other structural information. This allows for the automation of analysis and the extraction of graph data for external processing.[58]

Ghidra

Ghidra, a free and open-source alternative, offers a comparable and in some cases superior feature set, with a strong emphasis on collaboration and programmatic analysis.[59]

- **Multiple Graph Views:** Ghidra provides several interconnected views for topological analysis: the **Function Graph** (a CFG similar to IDA's), the **Function Call Graph** (visualizing a function's immediate callers and callees), and **Function Call Trees** (for recursively exploring the entire call chain up or down from a given function).[64]

- **Data Flow Analysis:** A key strength of Ghidra is its built-in data flow analysis. When a user selects a register or variable, the tool can highlight its forward and backward data flow, showing where the value came from and where it is used. This provides a deeper understanding than simple control flow alone.[66]
- **Project-Based Analysis:** Ghidra's architecture allows multiple binaries (e.g., an executable and its libraries) to be loaded into a single project. This facilitates cross-binary analysis, making it easier to trace control and data flow across module boundaries.[63]
- **Extensibility:** Ghidra features a powerful scripting framework supporting Java and Python (via Jython), giving scripts full access to the Ghidra API. This enables the programmatic construction and traversal of call graphs and CFGs, and is the primary mechanism for exporting this data for use with external tools like TDA libraries.[59]

6.3 Hardware Analysis Tools

Hardware reverse engineering requires a combination of physical laboratory equipment and specialized software for analysis.

Physical Analysis Equipment

As detailed in Section III, the initial phase of HRE is dominated by physical processes requiring tools such as chemical etching stations for decapsulation and delayering, and advanced imaging systems like Scanning Electron Microscopes (SEMs) and Transmission Electron Microscopes (TEMs) to visualize the circuit's layers.[29]

Schematic Capture and Netlist Analysis

Once a physical artifact is analyzed, software tools are needed to reconstruct and analyze its logical structure.

- **PCB Reverse Engineering Software:** For Printed Circuit Boards (PCBs), tools like Altium Designer, KiCad, Eagle, and OrCAD are used for schematic capture.[67] The process often involves scanning a high-resolution image of the PCB, importing it into the software, and manually tracing the connections (traces) and placing the components to recreate the

original schematic and layout files.[68]

- **Netlist Analysis Frameworks:** For ICs, after the gate-level netlist is extracted from images, specialized frameworks are required for its analysis. **HAL** is an open-source framework designed specifically for this purpose. It parses netlists from various sources into a graph-based representation and provides tools for traversing and analyzing the gates and nets, with the goal of becoming a hardware-equivalent of IDA or Ghidra.[30]

6.4 Integrating TDA into the Workflow

The critical link enabling the application of the TDA framework to reverse engineering is the ability to export the topological data from analysis platforms into a format consumable by mathematical libraries. The scripting APIs of tools like IDA Pro and Ghidra are the conduits for this integration.

A typical workflow would be:

1. **Data Extraction:** An analyst writes a script (e.g., in IDAPython or Ghidra's Python/Java API) that traverses the internal graph structure of the program being analyzed (e.g., the call graph). The script iterates through each function (node) and its calls (edges) and writes this information to a standard graph format like an adjacency list, GML, or GraphML.[65]
2. **TDA Computation:** A separate program, written in a language with strong scientific computing support like Python or R, loads the exported graph data. It then uses a dedicated TDA library (e.g., GUDHI, Ripser, Dionysus in Python; TDA in R) to perform the pipeline described in Section IV: define a metric, build a filtration, and compute the persistent homology of the data.[13]
3. **Visualization and Interpretation:** The TDA library outputs a persistence diagram or barcode. This output can then be visualized and interpreted by the analyst to identify the robust topological features of the original system, guiding further investigation back in the primary RE tool.

This workflow demonstrates how the extensibility of modern reverse engineering platforms is the key enabler for bridging the gap between practical systems analysis and advanced mathematical theory.

Theoretical Limits and Future Directions

7.1 The Boundaries of Reversibility

While reverse engineering, augmented by advanced topological methods, is a powerful tool for system comprehension, it is constrained by fundamental theoretical limits. These boundaries, rooted in information theory, computer science, and mathematics, dictate that a perfect, unambiguous reconstruction of the original design intent is often impossible. The goal of reverse engineering must therefore be understood not as achieving a single, provably "correct" representation, but as constructing a plausible and useful model that is consistent with the available evidence.

Information-Theoretic Limits

The process of creating software or hardware is inherently information-destroying. This concept is analogous to logical irreversibility in thermodynamics and computation.[70] Many computational operations are not invertible; for example, a logical AND gate with an output of 0 could have had three different input pairs (0,0, 0,1, 1,0). Given only the output, the input cannot be uniquely determined.[70] Similarly, when a designer chooses one implementation strategy over another, or when a compiler discards source-level comments and variable names, information about the design rationale, the alternatives considered, and the original semantics is permanently lost.[3] The final binary or physical chip is a projection of a much richer design space, and this projection cannot be perfectly reversed. The low-level artifact simply does not contain all the information that was present at the design stage.

Computational Limits (Undecidability)

The theory of computation imposes hard limits on what can be known about a program through static analysis. Foundational results, such as the undecidability of the Halting Problem, have direct consequences for reverse engineering.[72] For example, as mentioned previously, the problem of constructing a perfectly precise and complete static call graph is undecidable for any Turing-complete language.[19] It is impossible to create an algorithm that can, for all programs, perfectly resolve all indirect function calls and dynamic dispatches without running the code. This is why static analysis tools must resort to safe *over-approximations*, which is a direct consequence of this fundamental computational

limitation.[19]

The Ill-Posed Nature of Inverse Problems

Mathematically, reverse engineering can be framed as an **inverse problem**: we observe the output of a process (the compiled binary or fabricated chip) and attempt to infer the input that created it (the source code and design intent). Such problems are often **ill-posed**, a term defined by the mathematician Jacques Hadamard as lacking one or more of the following properties: a solution exists, the solution is unique, and the solution is stable (depends continuously on the data).[41] Reverse engineering often fails on the uniqueness criterion. It is entirely possible for two different source code implementations, reflecting different design intents, to be compiled by an optimizing compiler into the exact same machine code. Therefore, there is no unique "source" to be recovered, only a set of possibilities. This ambiguity is a theoretical barrier to achieving a single, definitive reconstruction.

7.2 Practical Limitations

Beyond these theoretical boundaries, reverse engineering faces significant practical challenges.

- **The Human Factor:** Despite advances in automation, reverse engineering remains a profoundly cognitive activity that relies heavily on human expertise, intuition, and domain knowledge.[71] An expert reverse engineer does not analyze every line of code; instead, they leverage experience to recognize patterns, form hypotheses about the program's function, and strategically focus their attention on the most relevant parts of the system.[73] This reliance on human skill makes the process time-consuming, expensive, and difficult to scale.[71]
- **Scale and Complexity:** The sheer size of modern software systems and integrated circuits makes exhaustive analysis intractable. A commercial software application can contain millions of lines of code, and a modern microprocessor contains billions of transistors. Analysts must rely on heuristics, abstraction, and targeted investigation to comprehend such systems, as a complete, bottom-up analysis is not feasible.[3]

7.3 Future Directions

Given these limitations, future advancements in reverse engineering and topological analysis will likely focus on augmenting human expertise and managing complexity, rather than pursuing perfect automation.

- **AI and Machine Learning in Reverse Engineering:** Machine learning models show great promise in automating sub-tasks of the reverse engineering process. This includes identifying compiler-generated code patterns, classifying functions based on their structure, suggesting meaningful variable names, and detecting code that has been obfuscated. These tools can act as intelligent assistants, helping human analysts to navigate complex binaries more efficiently.
- **Formal Methods for De-obfuscation:** The challenge posed by strong obfuscation may be addressed through the increasing use of formal methods. Tools that integrate SMT (Satisfiability Modulo Theories) solvers, such as the gooMBA plugin for IDA Pro, can mathematically prove the equivalence between a complex, obfuscated expression and a simpler canonical form.[58] Expanding this approach to reverse more complex structural obfuscations like control flow flattening is a significant area for future research.
- **Quantitative TDA for Architecture:** The TDA framework proposed in this report is primarily qualitative, identifying the presence of structural features. A promising future direction is the development of quantitative architectural metrics based on these topological features. For example, one could use the properties of a persistence diagram (e.g., the number and length of bars) to generate a "fingerprint" of a system's architecture. Such fingerprints could be used to compare different versions of a software product to track architectural decay, to compare a product to its competitors, or to automatically flag binaries with anomalous topological structures that could indicate the presence of malware or hardware Trojans. This would transform TDA from an exploratory tool into a quantitative measurement system for software and hardware architecture.

Works cited

1. Unlocking Innovation with Reverse Engineering | Lenovo US, accessed September 22, 2025, https://www.lenovo.com/us/en/glossary/reverse-engineering/
2. Achievements and Challenges in Software Reverse Engineering, accessed September 22, 2025, https://cacm.acm.org/research/achievements-and-challenges-in-software-reverse-engineering/
3. A Review of Reverse Engineering Theories and Tools - IJESI, accessed September 22, 2025, https://www.ijesi.org/papers/Vol(2)1/G213538.pdf
4. ghbintellect.com, accessed September 22, 2025, https://ghbintellect.com/hardware-reverse-engineering/#:~:text=As%20the%20term%20implies%2C%20reverse,detection%20to%20legacy%20design%20recovery.
5. Reverse engineering - Wikipedia, accessed September 22, 2025,

https://en.wikipedia.org/wiki/Reverse_engineering
6. Reverse engineering - Siemens Digital Industries Software, accessed September 22, 2025, https://www.sw.siemens.com/en-US/technology/reverse-engineering/
7. What is Reverse Engineering? | Astro Machine Works, accessed September 22, 2025, https://astromachineworks.com/what-is-reverse-engineering/
8. Re-implementing a legacy system | VU Research Portal, accessed September 22, 2025, https://research.vu.nl/files/235493004/Re_implementing_a_legacy_system.pdf
9. Modernization of an Online Giving Software System - Case Study, accessed September 22, 2025, https://www.scnsoft.com/case-studies/modernization-of-an-online-giving-software-system
10. www.geeksforgeeks.org, accessed September 22, 2025, https://www.geeksforgeeks.org/software-engineering/abstraction-levels-in-reverse-engineering/#:~:text=the%20software%20system.-,In%20reverse%20engineering%2C%20abstraction%20levels%20refer%20to%20the%20different%20levels,understand%20and%20analyze%20a%20system.
11. Abstraction Levels in Reverse Engineering - GeeksforGeeks, accessed September 22, 2025, https://www.geeksforgeeks.org/software-engineering/abstraction-levels-in-reverse-engineering/
12. (PDF) DANA Universal Dataflow Analysis for Gate-Level Netlist Reverse Engineering, accessed September 22, 2025, https://www.researchgate.net/publication/346706480_DANA_Universal_Dataflow_Analysis_for_Gate-Level_Netlist_Reverse_Engineering
13. Topological data analysis - Wikipedia, accessed September 22, 2025, https://en.wikipedia.org/wiki/Topological_data_analysis
14. View of A User's Guide to Topological Data Analysis | Journal of Learning Analytics, accessed September 22, 2025, https://learning-analytics.info/index.php/JLA/article/view/5196/6089
15. Demystifying Control Flow Graphs (CFG) in Software Engineering ..., accessed September 22, 2025, https://can-ozkan.medium.com/demystifying-control-flow-graphs-cfg-in-software-engineering-a4203279b7a5
16. Reverse engineering using graph queries - SciSpace, accessed September 22, 2025, https://scispace.com/pdf/reverse-engineering-using-graph-queries-igc37mahy5.pdf
17. Flow Graph in Code Generation - GeeksforGeeks, accessed September 22, 2025, https://www.geeksforgeeks.org/compiler-design/flow-graph-in-code-generation/
18. Igor's tip of the week #23: Graph view - Hex-Rays, accessed September 22, 2025, https://hex-rays.com/blog/igors-tip-of-the-week-23-graph-view
19. Call graph - Wikipedia, accessed September 22, 2025, https://en.wikipedia.org/wiki/Call_graph

20. An Empirical Study of Static Call Graph Extractors. - ResearchGate, accessed September 22, 2025, https://www.researchgate.net/publication/220403777_An_Empirical_Study_of_Static_Call_Graph_Extractors
21. Call Graph Soundness in Android Static Analysis - University of Washington, accessed September 22, 2025, https://homes.cs.washington.edu/~rjust/publ/call_graph_soundness_issta_2024.pdf
22. Static and Dynamic Reverse Engineering Techniques for Java Software Systems - Trepo, accessed September 22, 2025, https://trepo.tuni.fi/bitstream/handle/10024/67001/951-44-4811-1.pdf?sequence=1
23. en.wikipedia.org, accessed September 22, 2025, https://en.wikipedia.org/wiki/Call_graph#:~:text=Call%20graphs%20can%20be%20dynamic,possible%20run%20of%20the%20program.
24. Call Graph Soundness in Android Static Analysis - arXiv, accessed September 22, 2025, https://arxiv.org/html/2407.07804v1
25. Static vs. Dynamic Analysis: Choosing the Right Path in Reverse Engineering, accessed September 22, 2025, https://blog.aspiresys.com/software-product-engineering/static-vs-dynamic-analysis-choosing-the-right-path-in-reverse-engineering/
26. Understanding Software Dependency Graphs | Blog | VulnCheck, accessed September 22, 2025, https://www.vulncheck.com/blog/understanding-software-dependency-graphs
27. Software Dependency Graphs: Definition, Use Cases, and Implementation - PuppyGraph, accessed September 22, 2025, https://www.puppygraph.com/blog/software-dependency-graph
28. Manage software dependencies with graph visualization - Linkurious, accessed September 22, 2025, https://linkurious.com/blog/manage-software-dependencies-with-graph-visualization/
29. GHB Intellect Explains Hardware Reverse Engineering, accessed September 22, 2025, https://ghbintellect.com/hardware-reverse-engineering/
30. HAL - CAD for Assurance, accessed September 22, 2025, https://cadforassurance.org/tools/reverse-engineering-and-visualization/hal/
31. Practical partial hardware reverse engineering analysis - University of Cambridge, accessed September 22, 2025, https://www.repository.cam.ac.uk/bitstreams/ed1678d3-e510-41c0-8cfd-5ae507d38a40/download
32. Silicon Investigations IC & PCB Reverse Engineering Services, accessed September 22, 2025, https://www.siliconinvestigations.com/abt/abtrv.html
33. What Are the Different Types of Reverse Engineering Tools?, accessed September 22, 2025, https://ablengineering.com.au/What-Are-the-Different-Types-of-Reverse-Engineering-Tools-~69

34. Reverse Engineering of Integrated Circuits using Cross-Sectional Transmission Electron Microscopy—A Morphological and Structural Study | Request PDF - ResearchGate, accessed September 22, 2025, https://www.researchgate.net/publication/277593350_Reverse_Engineering_of_Integrated_Circuits_using_Cross-Sectional_Transmission_Electron_Microscopy-A_Morphological_and_Structural_Study
35. Reverse engineering Flash EEPROM memories using Scanning ..., accessed September 22, 2025, https://www.cl.cam.ac.uk/~sps32/cardis2016_sem.pdf
36. Circuit extraction - Wikipedia, accessed September 22, 2025, https://en.wikipedia.org/wiki/Circuit_extraction
37. Gate-Level Netlist Reverse Engineering for Hardware Security ..., accessed September 22, 2025, https://www.cerc.utexas.edu/utda/publications/C200.pdf
38. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries* - Chair of Programming Languages and AI - LMU Munich, accessed September 22, 2025, https://www.plai.ifi.lmu.de/publications/vmcai09-cfr.pdf
39. An Introduction to Topological Data Analysis: Fundamental and Practical Aspects for Data Scientists - Frontiers, accessed September 22, 2025, https://www.frontiersin.org/journals/artificial-intelligence/articles/10.3389/frai.2021.667963/full
40. (PDF) Reliable Malware Analysis and Detection using Topology ..., accessed September 22, 2025, https://www.researchgate.net/publication/365081565_Reliable_Malware_Analysis_and_Detection_using_Topology_Data_Analysis
41. Reverse engineering and identification in systems biology: strategies, perspectives and challenges - PMC - PubMed Central, accessed September 22, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC3869153/
42. obfuscation - What kinds of steps can I take to make my C++ ..., accessed September 22, 2025, https://reverseengineering.stackexchange.com/questions/118/what-kinds-of-steps-can-i-take-to-make-my-c-application-harder-to-reverse-engi/119
43. CcNav: Understanding Compiler Optimizations in Binary Code, accessed September 22, 2025, http://hdc.cs.arizona.edu/papers/vast_2020_ccnav.pdf
44. Enhancing Performance Through Control-flow Unmerging and Loop Unrolling - OSTI, accessed September 22, 2025, https://www.osti.gov/servlets/purl/2325331
45. [2507.18792] Decompiling Rust: An Empirical Study of Compiler Optimizations and Reverse Engineering Challenges - arXiv, accessed September 22, 2025, https://arxiv.org/abs/2507.18792
46. Applied Compiler Optimizations for Proving Code - Commit @ CSAIL — Home, accessed September 22, 2025, https://commit.csail.mit.edu/papers/2025/Ricardo_Ruiz_SB_Thesis.pdf
47. Reverse Engineering Optimizations: Division By Multiplication - Low Level Pleasure, accessed September 22, 2025, https://repnz.github.io/posts/reversing-optimizations-division/
48. (PDF) Reconstructing Control Flow from Predicated Assembly Code., accessed

September 22, 2025, https://www.researchgate.net/publication/220905052_Reconstructing_Control_Flow_from_Predicated_Assembly_Code
49. Code Obfuscation: A Comprehensive Guide Against Reverse, accessed September 22, 2025, https://doverunner.com/blogs/code-obfuscation-guide-against-reverse-engineering-attempts/
50. How Obfuscation Works in Software Development | by Endurance, the Martian - Medium, accessed September 22, 2025, https://medium.com/@hendurhance/how-obfuscation-works-in-software-development-7f52edfff520
51. Top 7 Source Code Obfuscation Techniques - Zimperium, accessed September 22, 2025, https://zimperium.com/blog/top-7-source-code-obfuscation-techniques
52. Code Obfuscation against Static and Dynamic Reverse Engineering - ResearchGate, accessed September 22, 2025, https://www.researchgate.net/publication/220722099_Code_Obfuscation_against_Static_and_Dynamic_Reverse_Engineering
53. Obfuscation Techniques: Enhancing Security Measures - VMRay, accessed September 22, 2025, https://www.vmray.com/malware-obfuscation-techniques/
54. [2508.12852] RoTO: Robust Topology Obfuscation Against Tomography Inference Attacks, accessed September 22, 2025, https://www.arxiv.org/abs/2508.12852
55. Secure and Energy-Efficient Network Topology Obfuscation for Software-Defined WSNs - University of Waterloo, accessed September 22, 2025, https://uwaterloo.ca/scholar/sites/ca.scholar/files/sshen/files/manaf2023secure.pdf
56. EigenObfu: A Novel Network Topology Obfuscation Defense Method | Request PDF, accessed September 22, 2025, https://www.researchgate.net/publication/385916300_EigenObfu_A_Novel_Network_Topology_Obfuscation_Defense_Method
57. GhIDA: Ghidra decompiler for IDA Pro - Cisco Blogs, accessed September 22, 2025, https://blogs.cisco.com/security/talos/ghida-ghidra-decompiler-for-ida-pro
58. IDA Pro: Powerful Disassembler, Decompiler & Debugger - Hex-Rays, accessed September 22, 2025, https://hex-rays.com/ida-pro
59. Ghidra - Wikipedia, accessed September 22, 2025, https://en.wikipedia.org/wiki/Ghidra
60. Graph view | Hex-Rays Docs, accessed September 22, 2025, https://docs.hex-rays.com/user-guide/disassembler/graph-view
61. FLARE IDA Pro Script Series: Simplifying Graphs in IDA | Mandiant | Google Cloud Blog, accessed September 22, 2025, https://cloud.google.com/blog/topics/threat-intelligence/flare-ida-pro-script-series-simplifying-graphs-ida
62. Graphs | Hex-Rays Docs, accessed September 22, 2025, https://docs.hex-rays.com/user-guide/user-interface/menu-bar/view/graphs
63. What is the difference between Ghidra and Ida? - Information Security Stack

Exchange, accessed September 22, 2025, https://security.stackexchange.com/questions/204876/what-is-the-difference-between-ghidra-and-ida
64. Introduction to Ghidra, accessed September 22, 2025, https://ghidra.re/ghidra_docs/GhidraClass/Beginner/Introduction_to_Ghidra_Student_Guide.html
65. Callgraphs with Ghidra, Pyhidra, and Jpype - clearbluejar, accessed September 22, 2025, https://clearbluejar.github.io/posts/callgraphs-with-ghidra-pyhidra-and-jpype/
66. GHIDRA VS IDA PRO: A COMPARISON OF TWO POPULAR ..., accessed September 22, 2025, https://osintteam.blog/ghidra-vs-ida-pro-a-comparison-of-two-popular-reverse-engineering-tools-55223fad9193
67. PCB copy software and tools for engineers - Wonderful PCB, accessed September 22, 2025, https://www.wonderfulpcb.com/blog/pcb-copy-software-tools-for-engineers-reverse-engineering/
68. Reverse engineer pcb to schematic - schematic script - Layout - KiCad.info Forums, accessed September 22, 2025, https://forum.kicad.info/t/reverse-engineer-pcb-to-schematic-schematic-script/44525
69. PCB Reverse Engineering - ScanCAD Intl., accessed September 22, 2025, https://scancad.net/pcb-reverse-engineering/
70. Reverse Engineering | American Scientist, accessed September 22, 2025, https://www.americanscientist.org/article/reverse-engineering
71. Reverse Engineering Cognition - MITRE Corporation, accessed September 22, 2025, https://www.mitre.org/sites/default/files/publications/pr-15-2630-reverse-engineering-cognition.pdf
72. Isn't compiler engineering just a combinatoral optimization problem? - Reddit, accessed September 22, 2025, https://www.reddit.com/r/Compilers/comments/1m4ugnb/isnt_compiler_engineering_just_a_combinatoral/
73. RE-Mind: a First Look Inside the Mind of a Reverse Engineer - USENIX, accessed September 22, 2025, https://www.usenix.org/system/files/sec22summer_mantovani.pdf
74. If you learn reverse engineering,anything is open source,is it true? - Reddit, accessed September 22, 2025, https://www.reddit.com/r/ReverseEngineering/comments/15tgdji/if_you_learn_reverse_engineeringanything_is_open/
75. Topology Analysis of Software Dependencies - McGill School Of ..., accessed September 22, 2025, https://www.cs.mcgill.ca/~martin/papers/tosem2008.pdf