LibIHT: A Hardware-Based Approach for Efficient and Evasion-Resistant Dynamic Binary Analysis

Changyu Zhao
University of Wisconsin-Madison
Madison, Wisconsin, USA
changyuz@stanford.edu

Yohan Beugin
University of Wisconsin-Madison
Madison, Wisconsin, USA
ybeugin@cs.wisc.edu

Jean-Charles Noirot Ferrand
University of Wisconsin-Madison
Madison, Wisconsin, USA
jcnf@cs.wisc.edu

Quinn Burke
University of Wisconsin-Madison
Madison, Wisconsin, USA
qkb@cs.wisc.edu

Guancheng Li
Tencent Xuanwu Lab
Beijing, Beijing, China
atumli@tencent.com

Patrick McDaniel
University of Wisconsin-Madison
Madison, Wisconsin, USA
mcdaniel@cs.wisc.edu

ABSTRACT

Dynamic program analysis is invaluable for malware detection, debugging, and performance profiling. However, software-based instrumentation incurs high overhead and can be evaded by anti-analysis techniques. In this paper, we propose LibIHT, a hardware-assisted tracing framework that leverages on-CPU branch tracing features (Intel Last Branch Record and Branch Trace Store) to efficiently capture program control-flow with minimal performance impact. Our approach reconstructs control-flow graphs (CFGs) by collecting hardware generated branch execution data in the kernel, preserving program behavior against evasive malware. We implement LibIHT as an OS kernel module and user-space library, and evaluate it on both benign benchmark programs and adversarial anti-instrumentation samples. Our results indicate that LibIHT reduces runtime overhead by over 150× compared to Intel Pin (7× vs 1,053× slowdowns), while achieving high fidelity in CFG reconstruction (capturing over 99% of execution basic blocks and edges). Although this hardware-assisted approach sacrifices the richer semantic detail available from full software instrumentation by capturing only branch addresses, this trade-off is acceptable for many applications where performance and low detectability are paramount. Our findings show that hardware-based tracing captures control flow information significantly faster, reduces detection risk and performs dynamic analysis with minimal interference.

ACM Reference Format:
Changyu Zhao, Yohan Beugin, Jean-Charles Noirot Ferrand, Quinn Burke, Guancheng Li, and Patrick McDaniel. 2025. LibIHT: A Hardware-Based Approach for Efficient and Evasion-Resistant Dynamic Binary Analysis. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Dynamic binary program analysis plays a critical role in reverse engineering and security evaluation [1, 3, 20, 22], enabling analysts to understand program behavior when source code is unavailable. In security contexts, such analysis is indispensable for identifying vulnerabilities, detecting malware, and verifying that system components adhere to expected behavior. Reverse engineering not only facilitates the discovery of hidden or obfuscated code constructs but also informs the development of robust defensive strategies.

Despite the importance of dynamic analysis, existing tools typically rely on software-based instrumentation methods (e.g.,Intel Pin [18] and DynamoRIO [5]) that inject monitoring code into the target program [4, 10, 21, 27]. While these approaches provide fine-grained visibility into runtime behavior, they incur substantial performance overhead and often perturb the program's natural execution. Furthermore, because software-based instrumentation inserts hooks and modifies the target binary at runtime, these frameworks are routinely subject to anti-analysis and evasion techniques [8, 9, 11, 12, 24, 25, 31], which in turn compromise control-flow reconstruction and the overall analysis reliability.

In this paper, we introduce LibIHT, a hardware-assisted tracing framework that complements traditional dynamic binary instrumentation by addressing its performance and detectability limitations. Rather than injecting instrumentation code at runtime, LibIHT leverages on-CPU branch tracing features built into commodity Intel x86/x64 processors [15]. Specifically, LibIHT uses Last Branch Record (LBR) and Branch Trace Store (BTS) to capture program execution with minimal overhead. By operating directly at the hardware level, LibIHT significantly reduces performance penalties and preserves natural execution. Although this approach records only branch addresses (thus offering lower granularity than full software instrumentation), we hypothesize that such coarse-grained data is sufficiently accurate for effective reverse engineering and dynamic analysis. Notably, while existing hardware-assisted tools for enforcing control flow integrity (CFI) achieve high accuracy and resilience in security enforcement, their designs have predominantly targeted runtime protection rather than assisting reverse engineering [7, 17, 23, 29, 32]. The complexity of harnessing hardware-level tracing and associated trade-offs have thus far prevented a unified solution tailored for reverse engineering. As a result, LibIHT is an ideal solution when analysis scenarios demand high performance and low detectability more than exhaustive semantic information.

We evaluate LibIHT across a diverse set of binaries, including both standard and adversarial executables, assessing its effectiveness in (1) reconstructing accurate control flow graphs, (2) minimizing runtime overhead compared to software-based instrumentation, and (3) maintaining resilience against common anti-analysis techniques. By leveraging the processor's built-in tracing features, LibIHT records execution data at the basic-block level and reconstructs precise control flow with minimal interference. Our results show that LibIHT reconstructs control flow with over 98% accuracy—achieving a mean Jaccard Index of 0.9836, 99.48% block coverage, and 99.69% edge coverage—while reducing average runtime overhead to 7× (versus 237× for DynamoRIO and 1,053× for Intel Pin). In other words, LibIHT delivers high-fidelity analysis at a fraction of the cost: it eliminates intrusive instrumentation and thereby avoids the severe slowdowns of traditional tools. Furthermore, its reliance on hardware-level tracing enables robust analysis that bypasses the diverse set of anti-debugging and anti-instrumentation benchmarks we considered.

Our contributions are as follows:

- We quantify challenges faced by software-based dynamic analysis by measuring both performance overhead and susceptibility to evasion across real-world benchmarks.
- We introduce LibIHT, a hardware-assisted program analysis framework that leverages Intel LBR and BTS to enable efficient and accurate execution tracing.
- We evaluate LibIHT on a diverse set of binaries, demonstrating its effectiveness in reconstructing control flow graphs while maintaining low runtime overhead and resilience against anti-analysis techniques.

The complete LibIHT toolchain (kernel module and user-space library) is released as open-source software at the following url: https://github.com/libiht/libiht

2 BACKGROUND

2.1 Software-Based Dynamic Analysis Tools

Traditional dynamic analysis is typically performed with dynamic binary instrumentation (DBI) frameworks such as Intel Pin [18] and DynamoRIO [5]. These DBI tools work by injecting monitoring code into the target program at runtime to capture execution traces and reconstruct control flow. Figure 1 and Figure 2 illustrate the high-level designs of Pin and DynamoRIO, respectively. Both frameworks employ just-in-time (JIT) compilation and maintain a dedicated code cache to hold instrumented code blocks, thereby enabling detailed runtime monitoring. Although both Intel Pin and DynamoRIO support flexible instrumentation and fine-grained tracing, they suffer from two major drawbacks:

- **Performance Overhead**: Instrumentation can slow down execution by orders of magnitude, making it impractical for real-time or large-scale analysis.
- **Anti-Analysis Evasion**: Many programs, especially malware, employ anti-instrumentation techniques to detect and evade software-based monitoring [8, 9, 11, 12, 24, 25, 31].

Limitations. Although these binary instrumentation frameworks are widely used and effective for detailed runtime program analysis, their design inherently prioritizes fine-grained visibility and flexible instrumentation over execution performance. As our motivating experiment results demonstrate (see Table 1), even simple, deterministic workloads such as `ls`, `dd`, `echo`, `sort`, `wc`, and `cat` (see Section A.1) suffer from significant performance degradation, even without adopting any analysis logic to the binary. For example, while native execution of `echo` requires only 4 ms, running it under Intel Pin increases the runtime to 508 ms, representing a slowdown of over 127×. Similar trends are observed with the other commands, highlighting that the overhead introduced by these frameworks

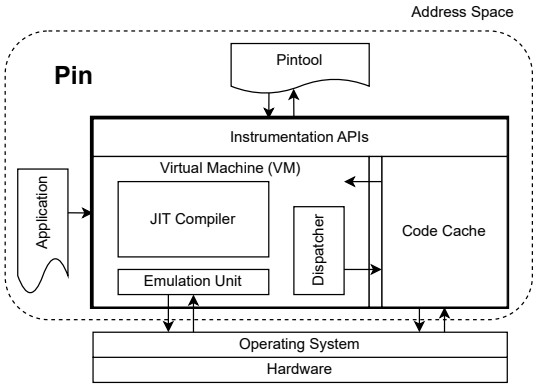

Figure 1: High-level architecture of Intel Pin (adapted from [18]). The tool inserts a virtual machine (VM), JIT compiler, and dispatcher to dynamically transform the target program, storing instrumented code in a dedicated code cache.

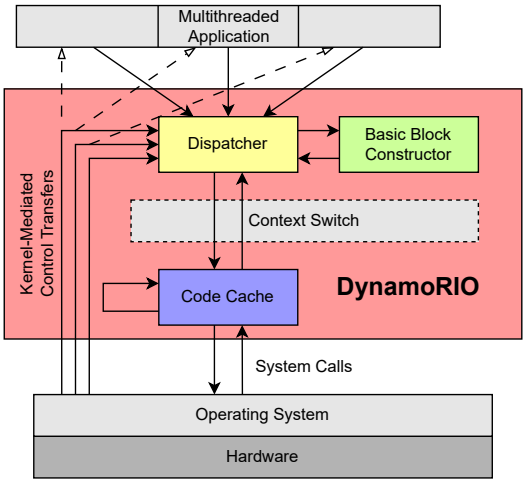

Figure 2: Overview of DynamoRIO's design (adapted from [5]). Similar to Intel Pin, it uses a code cache for instrumented basic blocks, with a dispatcher managing control transfers to and from the running application.

Table 1: Instrumentation overhead for commonly used `coreutils` under Intel Pin and DynamoRIO. Each workload command is listed in Section A.1. All timings are in milliseconds; slowdown factors are shown in parentheses.

Program	Native	Intel Pin	DynamoRIO
`ls`	7 ms	829 ms (118×)	94 ms (13×)
`dd`	14 ms	625 ms (45×)	68 ms (5×)
`echo`	4 ms	508 ms (127×)	54 ms (14×)
`sort`	9 ms	600 ms (67×)	63 ms (7×)
`wc`	8 ms	559 ms (70×)	56 ms (7×)
`cat`	7 ms	497 ms (71×)	55 ms (8x)

can severely distort execution timing. This trade-off renders them impractical for performance-sensitive or real-time systems and may lead to inaccuracies in analyses such as control flow graph (CFG) reconstruction, particularly in security-critical applications where fidelity and stealth are paramount.

2.2 Hardware-Assisted Tracing Mechanisms

To address the limitations of software-based tracing, modern x86 processors provide hardware-level branch tracing mechanisms. These include *Last Branch Record (LBR)*, *Branch Trace Store (BTS)*, and *Intel Processor Trace (Intel PT)* [15]. By capturing control-flow transitions at the processor level, these features enable efficient tracing without instrumenting program code.

LBR. Figure 3 (left) illustrates the Last Branch Record mechanism. LBR uses a small, fixed-size circular buffer of Model Specific Registers (MSRs) to record the source and destination addresses of the most recent branch instructions executed by the CPU. Each new branch pushes the oldest entry out of the buffer, so LBR always reflects the last n (ranging from 4 to 32 base on CPU model) branches taken. Because all operations occur entirely in hardware registers, enabling LBR incurs virtually no runtime overhead. This makes LBR ideal for capturing short-term execution snapshots, such as profiling tight code paths or quickly sampling control-flow behavior without disrupting program throughput.

BTS. Figure 3 (right) shows the Branch Trace Store mechanism. Instead of limited MSRs, BTS writes every branch event into a dedicated in-memory trace ring buffer managed by the CPU's Debug Store facility. When a branch occurs, the processor appends its source and destination addresses to this buffer until either tracing is disabled or interrupted. When the ring buffer nears interrupt threshold, the Debug Store facility raises an interrupt, giving the OS an opportunity to process the collected branch records to prevent data loss. BTS therefore provides complete, long-duration coverage of control-flow transitions, at the cost of extra memory bandwidth and the need to allocate and periodically drain the trace buffer. The ring buffer's size is customizable, but its memory allocation must follow restrictions specified in the Manual (e.g., alignment and placement requirements) [15]. Users must configure properly and retrieve its contents—either on the real time or after execution—to reconstruct a full sequence of branch records for offline analysis.

Intel PT. Intel PT is a third hardware tracing mechanism that delivers instruction-level control-flow information in a highly compact, packetized format. As the CPU runs, it emits packets that encode branch-taken targets (TIP packets) and conditional branch outcomes (TNT packets), along with timing and context-switch markers. These packets are written into a ring buffer in memory, which tools like Linux perf [16] can capture with minimal overhead.

However, turning the raw PT packet stream into a usable control-flow trace requires substantial post-processing. A decoder must decompress each packet, translate packet fields back into concrete instruction addresses, and correlate those addresses with the exact binary image, symbol table, and load offsets in use during tracing. In effect, it must replay the recorded execution against the original executable to reconstruct basic blocks and edges. This decoding step consumes significant CPU cycles and I/O bandwidth, often exceeding the cost of the initial trace collection. As a result, full PT trace decoding can become a performance bottleneck and a memory-pressure concern, limiting its practicality for real-time monitoring or large-scale batch analysis.

3 SYSTEM

In this section, we present a comprehensive description of our approach to leveraging hardware tracing capabilities, addressing existing challenges and outlining our threat model and assumptions. We further describe the detailed system design of LibIHT, including the specific roles and interactions between kernel-space and user-space components, their communication mechanisms, and the workflows that enable efficient control flow analysis.

3.1 Problem Context

In dynamic analysis [3, 20], a fundamental task is the precise inference of a program's control flow. Accurately reconstructing CFG is critical for a broad spectrum of applications including malware detection, reverse engineering, program comprehension, vulnerability assessment, and debugging. Control flow information elucidates both the structural and behavioral properties of a program, enabling analysts to identify executed paths, isolate anomalous or malicious behaviors, and decode intricate logic within binaries lacking source code. This capability is especially crucial in security contexts, where the reconstruction of CFG can expose obfuscated code, hidden functionalities, and subtle malicious actions that may evade conventional static or cursory dynamic analysis.

As discussed in Section 2.1, existing methodologies exhibit significant limitations, underscoring the need for alternative approaches that deliver high-fidelity control flow reconstruction while concurrently minimizing performance overhead and resisting evasion.

3.2 Threat Model

Our threat model considers adversaries who have complete control over the distributed binaries and actively deploy sophisticated anti-analysis techniques to protect the internal logic of their applications from external scrutiny. In practice, adversaries typically attempt to detect and resist traditional software instrumentation through various defensive measures, including debugger detection, extensive code obfuscation, runtime integrity checks, and timing or environmental based evaluations specifically designed

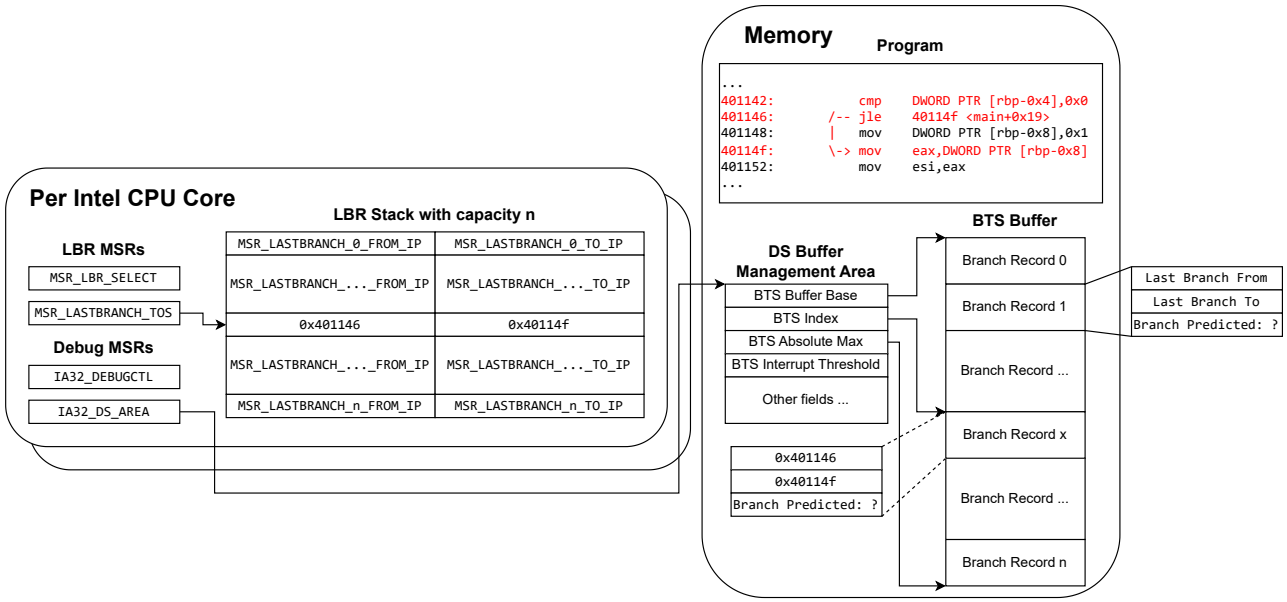

Figure 3: High-level architecture of Intel's LBR and BTS mechanism. LBR maintains a stack of MSRs on each CPU core for the most recent branches, while BTS extends this by logging branch records into a dedicated buffer in memory.

to identify performance overhead associated with instrumentation [8, 9, 11, 12, 24, 25, 31]. Additionally, adversaries may employ dynamic modifications of the program's control flow at runtime, further complicating analysis efforts by obfuscating execution paths and obscuring true program behavior.

In defining our threat model, we specifically assume that adversaries neither intend nor possess the ability to gain kernel-level privileges (ring-0), and we further posit that kernel integrity remains robustly safeguarded by modern OS protections. We also explicitly state that adversaries do not have the capability to successfully compromise or subvert these kernel-level safeguards. Given these clearly defined constraints, adversaries are effectively limited to user-space manipulation only. Recognizing this critical security boundary, LibIHT strategically positions its essential tracing and analysis components within the kernel space, effectively leveraging the user-kernel boundary. By operating in this secure and privileged environment, LibIHT significantly enhances its capability to resist aggressive adversarial attempts at evasion, distortion, or disruption of trace data integrity, thereby preserving the reliability and accuracy of analysis results.

3.3 System Overview

LibIHT is a dynamic binary analysis library framework that reconstructs execution control flow using hardware tracing capabilities provided by modern commodity Intel processors. It employs a dual-component architecture comprising kernel-space and user-space elements. The kernel-space component is responsible for configuring and managing hardware tracing features, while the user-space component interacts with the kernel module to retrieve and analyze trace data. Figure 4 illustrates the high-level architecture of LibIHT and interactions between these components.

Kernel-Space Component. The kernel components, implemented as Linux kernel modules and Windows kernel drivers, operate within privileged ring-0 kernel space. These modules interact directly with Intel processor hardware to access raw hardware-generated trace data (LBR/BTS), manage hardware trace parameters mentioned in Section 2.2, and securely store the generated trace data. By leveraging kernel-level privileges, LibIHT ensures integrity and effectively mitigates detection and evasion attempts from user-space adversaries as studied in Section 3.2.

User-Space Component. User-space components are implemented as a shared library on Linux and a dynamic link library (DLL) on Windows, offering accessible APIs for user-space applications performing security analysis. These components communicate with kernel modules through secure kernel-user interfaces, primarily using device I/O Control (`ioctl`) system calls, to retrieve raw hardware trace data. They abstract the complexity of hardware interactions by exposing high-level APIs, which process raw trace data to reconstruct control flow graphs (CFGs), thus enabling detailed security assessments. This separation between kernel and user spaces maintains strong security boundaries while optimizing usability.

3.4 Interaction and Data Flow

The operational workflow of LibIHT consists of two phases: (1) Data Collection and (2) Control-Flow Reconstruction. In the Data Collection phase, a user-space component initiates a trace session by issuing `ioctl` requests to the kernel module. The kernel module then configures and activates the hardware tracing mechanisms

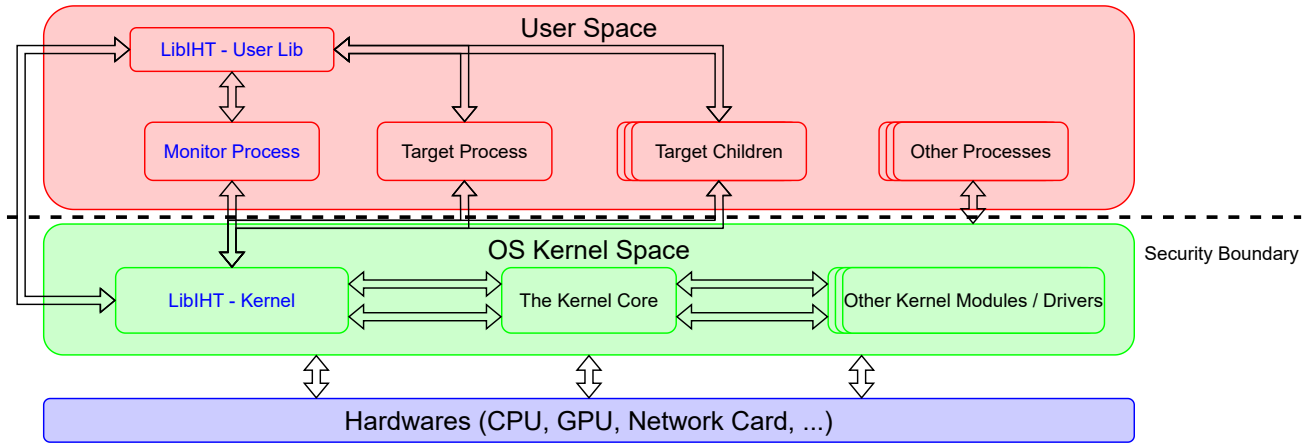

Figure 4: High-level architecture of LibIHT. The kernel-space component manages hardware tracing and collects data for the target process and its children. An optional user-space monitor (e.g., a debugger or custom tool) can analyze the trace data, though it is not required for core operation.

(LBR and BTS). Once tracing is enabled, the processor begins generating branch records in real time as the program executes. The kernel module buffers these trace records in protected kernel memory, preventing any tampering by user-space malware.

In the Control-Flow Reconstruction phase, the user-space analysis library retrieves the buffered trace data via secure `ioctl` calls. The library then parses the sequence of branch records and rebuilds the program's runtime CFG. This reconstruction involves ordering the basic blocks as they were executed and linking them into a CFG that reflects the program's actual paths. The resulting CFG provides critical insight for analysis, enabling detection of hidden behaviors, reverse engineering of program logic, and identification of anomalies. By dividing work between kernel and user space in this manner, LibIHT achieves high efficiency and remains resilient to evasion: the kernel performs low-level data capture with minimal overhead, and user-space performs heavy analysis without risking the integrity of the tracing process.

3.5 Trade-offs and Advantages

In designing LibIHT, we deliberately traded off rich semantic detail for dramatically improved performance and enhanced resistance to anti-analysis measures. Traditional software-based instrumentation tools capture comprehensive execution context-including complete instruction states, register contents, and memory access patterns-that enable highly detailed control-flow reconstruction. In contrast, LibIHT relies exclusively on hardware-level branch tracing, recording only the source and destination addresses of branch events. This means that while LibIHT does not capture the finer details necessary for complete state recovery, it avoids the heavy computational and storage burdens imposed by software instrumentation. For many applications, such as rapid initial reverse engineering or automated malware triage, the overall control-flow information is sufficient to pinpoint suspicious functions or anomalous behavior.

We conceptualize this design decision within a trade-off space defined by two axes: *performance* and *semantic granularity*. On the performance axis, software-based tools often slow down execution by orders of magnitude-our preliminary experiment results show slowdowns exceeding 100× in some cases-rendering them impractical for real-time or large-scale analysis. By shifting instrumentation to the hardware level, LibIHT achieves significantly higher throughput, facilitating rapid analysis and minimally perturbing the program's natural execution flow. On the semantic granularity axis, the cost of this performance gain is the loss of detailed context: LibIHT captures only branch addresses, omitting detailed state information that might be essential for deep forensic analysis. However, for early-stage reverse engineering and dynamic analysis tasks, this level of detail is typically sufficient to flag potential security issues. Importantly, these approaches are complementary. Once LibIHT identifies areas of interest in an execution trace, researchers can then employ detailed software-based instrumentation or static analysis techniques to perform deeper investigation.

Moreover, by operating within kernel space and harnessing intrinsic processor features, LibIHT is inherently more robust against anti-instrumentation evasion techniques. In summary, LibIHT represents a balanced solution that prioritizes rapid, low-overhead trace collection and stealth while accepting a reduction in semantic detail-a trade-off that we argue is both reasonable and practical for a broad range of security and debugging applications.

4 EVALUATION

We apply our approach to answer the following research questions:

- **RQ1 (Benign Setting - Accuracy):** How closely does LibIHT's traced CFG match actual execution paths, and what semantic details are lost when relying on hardware-based tracing instead of software instrumentation? (Section 4.2)
- **RQ2 (Benign Setting - Performance)** How does the performance of LibIHT compare to traditional dynamic analysis frameworks (e.g., Intel Pin and DynamoRIO) in benign scenarios across diverse binaries? (Section 4.3)

- **RQ3 (Adversarial Setting - Resilience):** Under adversarial conditions (e.g., anti-analysis or adaptive attacks), how effectively does LibIHT preserve the accuracy and performance established in RQ1 and RQ2? What techniques can adversaries use to target hardware-based tracing, and how does LibIHT mitigate such attacks? (Section 4.4)

4.1 Experimental Setup

We conducted our experiments on an Intel NUC 11 Essential Kit (model NUC11ATKPE), equipped with an Intel Pentium Silver N6005 processor, 8 GB of memory, and a 512 GB SSD. Ubuntu 24.04 LTS was installed to provide a modern Linux environment with up-to-date kernel support. This system was chosen because it is a commodity Intel platform that offers LBR and BTS capabilities, making it well-suited for evaluating hardware-based control-flow tracing under real-world conditions.

Software Baselines. In our experiments, we compare LibIHT against two widely used state-of-the-art dynamic binary analysis tools as discussed in Section 2.1: DynamoRIO (v 11.3.0) [5] and Intel Pin (v 3.31) [18]. These tools serve as a reference baseline for evaluating LibIHT on both the accuracy and performance overhead.

Benign Benchmark. To comprehensively assess accuracy and performance overhead under normal execution conditions, we construct a benign benchmark suite combining standard, widely-recognized programs. We include several GNU Core Utilities (Coreutils) programs, such as `ls`, `grep`, and `cat`, among others. These utilities are chosen due to their widespread use in daily computing tasks, making them ideal candidates for assessing tracing accuracy and overhead under realistic, routine user-space execution scenarios. Additionally, their control-flow diversity—from simple, linear execution to complex branching and loops—facilitates rigorous assessment of trace completeness and accuracy.

The benign benchmark form a comprehensive set of program for evaluating LibIHT's tracing fidelity and instrumentation overhead in controlled, benign conditions.

Adversarial Benchmark. To evaluate the resilience and robustness of LibIHT in adversarial environments, we construct a carefully curated adversarial testbench consisting of specifically chosen malware Proof-of-Concept (PoC) samples, each featuring known anti-analysis measures [31]. The adversarial benchmarks are selected to represent a broad spectrum of anti-instrumentation and evasive techniques encountered in real-world malicious software [9, 12, 25] including:

Indirect Evasion Techniques. Indirect evasion techniques detect the side effects of instrumentation by examining abnormal processor state or wrong simulation of real instruction execution. For example, the **Functional Limitations (FL)** tests include:

- **FSBase Integrity Check:** Verifies consistency in the `fsbase` register by comparing outputs from `rdfsbase` and `prctl`.
- **RIP Preservation Check:** Ensures that the instruction pointer remains unchanged after operations like system calls.

Direct Evasion Techniques. Direct evasion techniques actively probe for artifacts introduced by instrumentation. These methods are typically subdivided as follows:

- **Code Cache Artifact Detection (CA):** Techniques such as NX Page Execution Detection, Self-Modifying Code (SMC) Detection, and VMLeave Pattern Detection are used to expose shortcomings in handling dynamic code modifications or cached instrumentation artifacts.
- **Environment Artifact Detection (EA):** These methods search for instrumentation-related artifacts, such as environment variables commonly set by DBI frameworks, analysis of mapped files (e.g., inspecting `/proc/self/maps` for known instrumentation libraries), and detection of abnormal memory page permissions.
- **Runtime Just-in-Time Compiler Overhead Detection (RO):** This subcategory involves measuring execution timing anomalies across repeated code path iterations or inspecting dynamic linking behavior to detect irregularities indicative of JIT-based instrumentation.

Collectively, these adversarial benchmarks allow us to rigorously evaluate LibIHT's resilience under sophisticated evasion attempts. Table 2 provides an overview of the adversarial test samples currently employed in our evaluation, along with a brief description of the corresponding techniques.

Metrics. After collecting traces, we evaluate LibIHT along three dimensions:

- *Accuracy of Tracing:* We compare the CFG reconstructed by LibIHT to a ground-truth CFG obtained via static analysis for each program. We use graph similarity metrics Jaccard index and normalized graph edit distance (GED) to quantify structural similarity, and we calculate coverage metrics (the percentage of basic blocks and edges from the ground truth that dynamic analysis tools captured).
- *Performance Overhead:* We measure runtime overhead by comparing execution time with LibIHT tracing against native execution time (no tracing). We report the slowdown factor and record throughput in instructions per second (IPS) as a secondary performance metric.
- *Resilience to Evasion:* We observe whether each adversarial sample successfully detects or evades analysis. Specifically, we note if a program crashes or alters its behavior under LibIHT tracing. If LibIHT is partially evaded, we measure the drop in tracing accuracy and increase in overhead relative to benign conditions.

These metrics and criteria explained in the following subsections will determine how well LibIHT answers our research questions.

4.2 Accuracy in Benign Settings

In this section, we aim to answer **RQ1**, i.e., How accurately do LibIHT 's CFG traces reflect true execution paths, and what semantic details are lost by using hardware-based tracing versus software instrumentation?

To systematically evaluate the accuracy of LibIHT and other dynamic analysis tools, we compare their reconstructed control-flow graphs (CFGs) against a ground-truth CFG obtained via static analysis (using angr [28]). Because dynamic tools emit only a sequence of branch addresses (i.e., a trace of basic-block transitions), we first project that trace back onto the full, static CFG to isolate the exact execution path. Once we have two subgraphs, the ground-truth

Table 2: Overview of Adversarial Benchmark Samples Adopted from [31]

Sample	Category	Subcategory	Technique/Description
`fl_fsbase.c`	Indirect Evasion	FL	FSBase Integrity Detection: detects discrepancies (e.g., comparing `rdfsbase` vs. `prctl`).
`fl_ripsyscall.c`	Indirect Evasion	FL	RIP Preservation Detection: verifies that the instruction pointer remains unchanged after system calls.
`ca_nx.c`	Direct Evasion	CA	NX Page Execution Detection: tests for code cache artifacts via non-executable page checks.
`ca_smc.c`	Direct Evasion	CA	Self-Modifying Code Detection: checks handling of dynamic code modifications.
`ca_vmleave.c`	Direct Evasion	CA	VMLeave Pattern Detection: probes instrumentation layers for abnormal transitions.
`ea_envvar.c`	Direct Evasion	EA	Environment Variable Detection: searches for DBI-specific env vars (Pin, DynamoRIO, etc.).
`ea_mapname.c`	Direct Evasion	EA	Mapped File Name Detection: inspects `/proc/self/maps` for known instrumentation tools.
`ea_pageperm.c`	Direct Evasion	EA	Page Permission Detection: identifies suspicious read-write-execute memory regions.
`ro_jitbr.c`	Direct Evasion	RO	JIT Branch Timing Detection: measures anomalies in repeated code paths to detect JIT overhead.
`ro_jitlib.c`	Direct Evasion	RO	JIT Library Loading Detection: probes behavior of dynamic linking or JIT engines for abnormal latency.

execution path and the traced path, we quantify their similarity and coverage with the following detailed metrics:

- *Jaccard Index:* Measures the overlap between the two subgraphs' node and edge sets, giving a normalized score of similarity between 0 and 1. A higher Jaccard index indicates a larger shared subgraph relative to the union of both, capturing both similarity and diversity in a single value.
- *Normalized GED:* Counts the minimum number of structural edits (node/edge insertions or deletions) required to transform the traced subgraph into the ground-truth subgraph, then divides by the total number of elements in the ground truth. By normalizing to graph size, we account for varying trace complexities and ensure the metric remains comparable across programs of different scales.
- *Basic Block Coverage:* The fraction of ground-truth basic blocks that appear in the traced CFG. This directly measures how comprehensively a tool records every executed block, reflecting path completeness at the node level.
- *Edge Coverage:* The fraction of ground-truth control-flow edges (i.e., transitions between blocks) that are present in the traced CFG. This metric gauges how faithfully the tool preserves the actual execution semantics.

Together, these metrics provide complementary perspectives on tracing fidelity: the Jaccard index and normalized GED capture structural similarity and deviation, while block and edge coverage quantify completeness of execution path recovery. This multidimensional view helps illustrate the trade-offs between semantic detail and performance inherent in hardware-based tracing versus software instrumentation.

Table 3: Mean Control-Flow Graph Reconstruction Metrics on Benign Benchmarks

Metric	LibIHT	Intel Pin	DynamoRIO
Jaccard Index	98.36 ± 0.14%	100%	100%
Normalized GED	0.0231	≈ 0	≈ 0
Block Coverage	99.92 ± 0.04%	100%	100%
Edge Coverage	99.48 ± 0.15%	100%	100%

Table 3 reports the mean and variance of four key CFG reconstruction metrics: Jaccard index, normalized GED, block coverage, and edge coverage. The data was aggregated over all binaries inside benign benchmarks suites. The low standard deviations in each row indicate that there are no significant outliers: each tool behaves consistently across diverse binaries.

Looking first at the software-instrumentation baselines, both Intel Pin and DynamoRIO achieve perfect reconstruction (100% Jaccard, ≈ 0 GED, and full block and edge coverage), reflecting their ability to instrument every executed instruction and record only the target process's control-flow transitions. In contrast, LibIHT attains a Jaccard index of 98.36% (± 0.14%) and a normalized GED of 0.0231, alongside nearly complete coverage (99.92% ± 0.04% of blocks and 99.48% ± 0.15% of edges).

The slight gap between LibIHT and the software baselines does not stem from missing user-mode transitions. Instead, it is caused by a small amount of noise that LBR and BTS record indiscriminately. These extra branches appear because the CPU executes a short sequence of instructions just as it enters and exits kernel mode for system calls or interrupts. At those moments, the privilege

bit has not yet been fully set or cleared. As a result, even though LibIHT disables tracing in privileged mode, it still captures those boundary transitions. This behavior is unavoidable given the hardware's branch-recording granularity, but a simple post-processing step can remove all such kernel entries. By discarding any branch whose address falls outside the target process's user-space region, one can restore a perfect 100% match with the ground-truth CFG. Although this cleanup is not yet automated, LibIHT 's raw traces already achieve over 99% block and edge coverage, demonstrating sufficient precision for most analysis tasks.

Takeaway: Accuracy in Benign Settings

LibIHT delivers near-perfect control-flow reconstruction with minimal variance, validating that its hardware-assisted approach incurs only a modest and easily mitigated cost in metric-level similarity compared to traditional software instrumentation.

4.3 Performance in Benign Settings

In this section, we aim to answer **RQ2**, i.e., How does the performance of LibIHT compare to traditional dynamic analysis frameworks (e.g., Intel Pin and DynamoRIO) in benign scenarios?

To ensure a fair and reproducible comparison, we run each binary in our benchmark suite with a small set of representative argument combinations under six modes: native execution, Intel Pin without tracing (`pin_no_trace`), Intel Pin with tracing(`pin_trace`), DynamoRIO without tracing(`drrun_no_trace`), DynamoRIO with tracing ((`drrun_trace`)), and LibIHT with tracing (`libiht_trace`). Each configuration is executed five times per argument set, and we report the mean and standard deviation across all runs and all argument sets for each binary, thereby capturing both per-run and per-argument variability.

We collect two complementary performance metrics:

- *Execution Slowdown Factor:* Defined as
$$Slowdown = \frac{T_{\text{instrumented}}}{T_{\text{native}}},$$
where T is the wall-clock execution time measured by `perf`. This ratio normalizes results across different program runtimes and isolates the overhead introduced by each instrumentation framework.
- *Instructions Per Second (IPS):* The total executed instruction count on performing the full tracing workload divided by execution time, measured by `perf stat`. IPS quantifies the impact of tracing on CPU throughput, capturing the cost of hardware logging logic (for LibIHT) versus software instrumentation overhead.

By combining slowdown factors with IPS measurements, we both normalize for program length and directly observe the per-instruction cost of each approach. This dual-metric approach ensures that neither long-running workloads nor instruction-dense short workloads bias the comparison.

Figure 5 plots the mean wall-clock time for each Unix utility on a logarithmic y-axis. Across every benchmark, the orange LibIHT bars lie very close to the blue native baseline, whereas both Intel Pin and DynamoRIO with tracing incur slowdowns of one to three orders of magnitude. The error bars reveal that commands involving frequent user–kernel transitions—such as `printf`, `dd`, and `ls`—exhibit somewhat greater variability under LibIHT. This is because LibIHT must temporarily disable and then re-enable hardware tracing around each system call or context switch, and the exact number and timing of those transitions can fluctuate with workload characteristics and OS scheduling. In contrast, CPU-bound utilities show very tight distributions, underscoring that LibIHT 's per-instruction overhead remains minimal and stable even when averaged over multiple argument runs.

Turning to Figure 6, we normalize each mode to native execution by plotting the slowdown factor. It reveals a consistent pattern: The orange bars (LibIHT) cluster around the lower end of the scale, while the red (DynamoRIO with trace) and brown (Pin with trace) bars extend one to two orders of magnitude higher. Dashed horizontal lines mark the mean slowdown for LibIHT (6.58×), DynamoRIO (253.25×), and Pin (1090.34×), clearly illustrating that LibIHT 's overhead remains roughly two orders of magnitude below that of software instrumentation for every benchmark. This normalized view confirms the same performance advantage seen in raw execution times, without any special-casing for particular workloads.

Finally, Figure 7 reports the average IPS observed while each tool executes the full tracing workload. Native runs sustain about 2.2×10^9 IPS, whereas LibIHT records roughly 1.0×10^9 IPS. This gap reflects the inherent cost of on-CPU branch logging rather than any injected code. By comparison, DynamoRIO with tracing and Intel Pin without tracing report higher IPS—around 4.5×10^9 and 3.3×10^9, respectively. It is not because they execute the original program faster but due to the fact that they spend most execution cycles running instrumentation routines such as dynamic dispatch loops, just-in-time compilation, and basic-block construction. Although those routines increase the raw instruction throughput, the sheer volume of instrumentation logic also dramatically increase overall wall-clock time, as shown in Figure 5 and Figure 6. In contrast, LibIHT executes only the application's native branches plus a small logging overhead, yielding much lower overall tracing cost despite a modest per-instruction slowdown.

Takeaway: Performance in Benign Settings

LibIHT delivers dramatically lower end-to-end overhead compare to software-based dynamic analysis tools. Although per-instruction IPS is modestly reduced by hardware logging, the net performance benefit remains substantial across diverse workloads.

4.4 Resilience in Adversarial Settings

In this section, we aim to answer **RQ3**, i.e., Under adversarial conditions (e.g., anti-analysis or adaptive attacks), how effectively does LibIHT preserve the accuracy and performance established in RQ1 and RQ2? What techniques can adversaries use to target hardware-based tracing, and how does LibIHT mitigate such attacks?

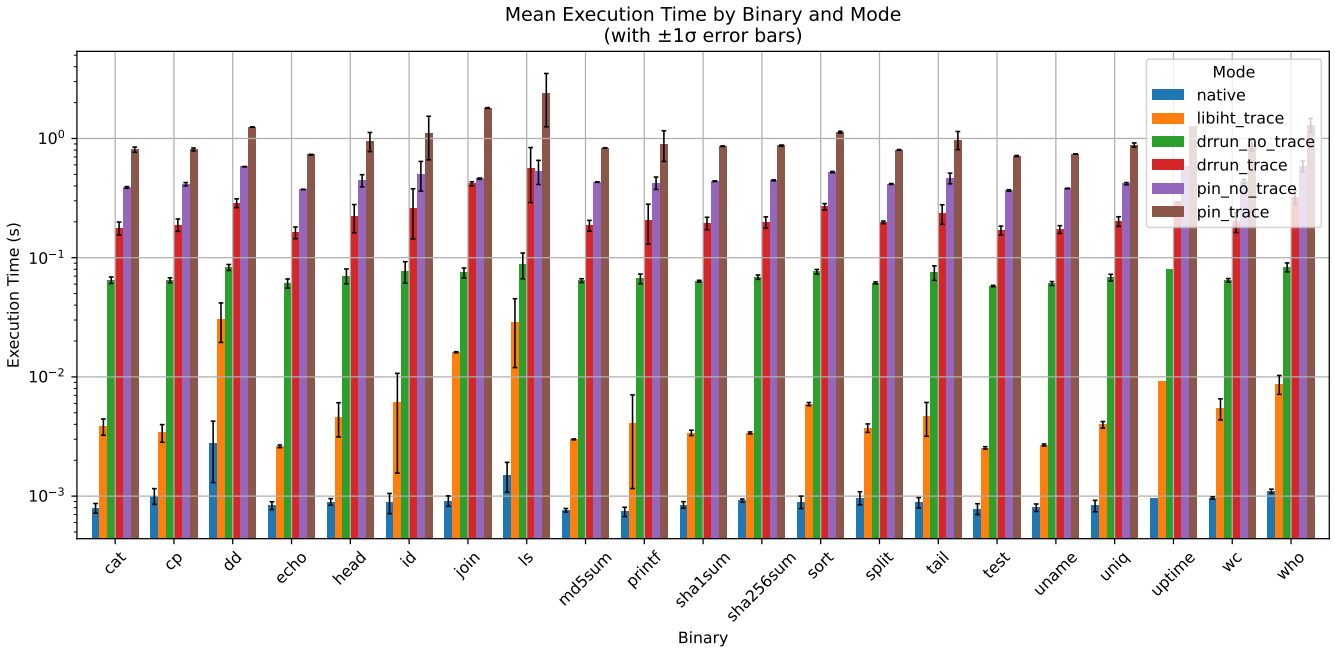

Figure 5: Comparison of mean execution times (in seconds) for standard Unix command-line utilities under six execution modes.

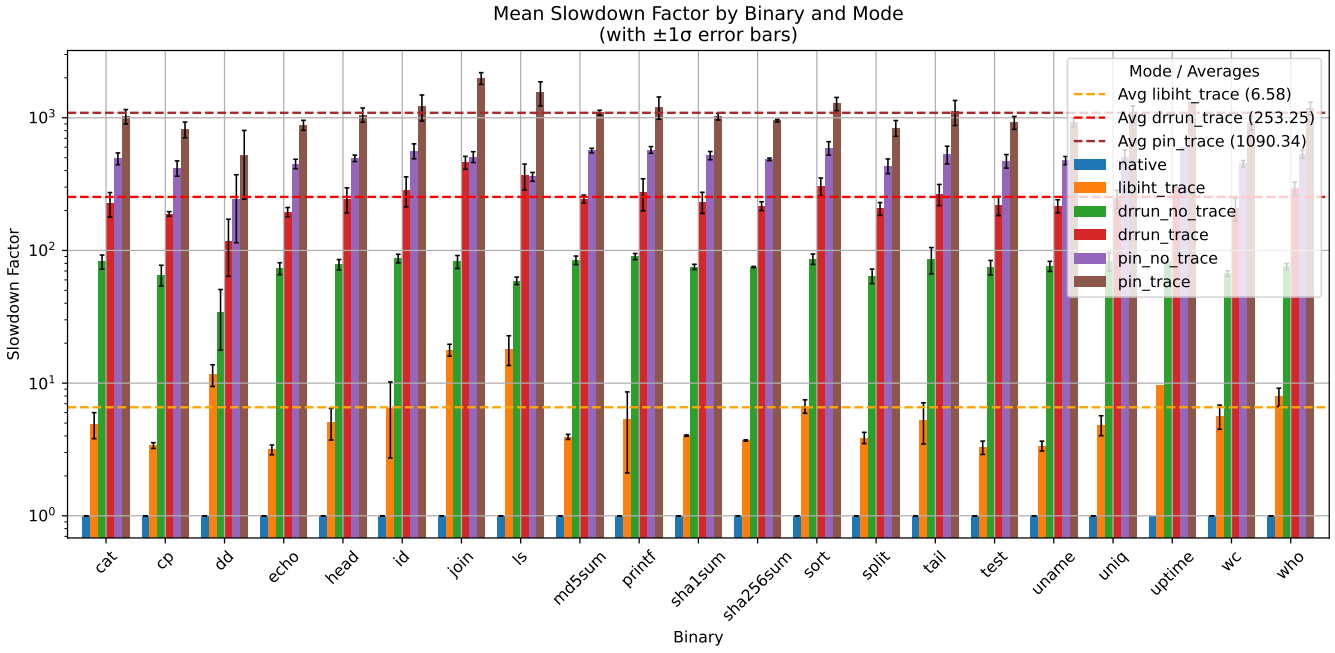

Figure 6: Average slowdown factors for standard Unix command-line utilities relative to native execution across six modes. Dashed horizontal lines denote the overall mean slowdown across the benign benchmark

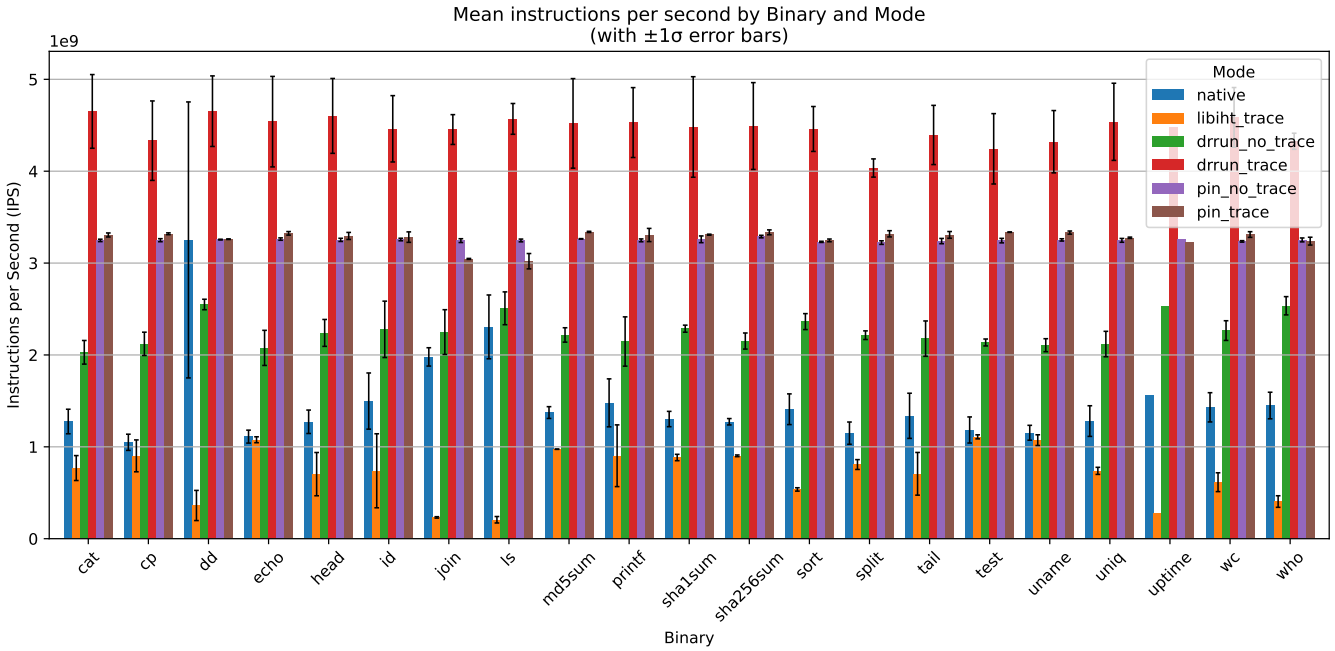

Figure 7: Average instruction per second for standard Unix command-line utilities relative to native execution across six modes.

Table 4: Detection results of dynamic analysis tools by adversarial PoC samples employing anti-debugging and anti-instrumentation techniques. A check mark (✓) indicates the tool was detected by the sample's detection mechanism; a cross mark (✗) indicates the tool was not detected.

Sample	Intel Pin	DynamoRIO	LibIHT
`fl_fsbase.c`	✗	✓	✗
`fl_ripsyscall.c`	✗	✓	✗
`ca_nx.c`	✗	✓	✗
`ca_smc.c`	✓	✗	✗
`ca_vmleave.c`	✓	✗	✗
`ea_envvar.c`	✓	✓	✗
`ea_mapname.c`	✓	✓	✗
`ea_pageperm.c`	✓	✓	✗
`ro_jitbr.c`	✓	✓	✗
`ro_jitlib.c`	✗	✗	✗

We rigorously evaluate resilience through the execution of curated malware PoC samples [12, 31], each embodying specific anti-instrumentation and evasive strategies. First, we systematically document the success or failure of program execution by determining whether each binary detects the presence of tracing, prematurely terminates, or otherwise misbehaves during analysis. For cases of partial tracing success, we further investigate the conditioned accuracy and overhead, assessing the degree to which LibIHT retains control flow tracing accuracy compared to benign scenarios (as described in RQ1). Additionally, we quantify the performance overhead of dynamic binary analysis tools introduced under various adversarial evasion techniques, comparing these results to the overhead metrics established in benign scenarios (as described in RQ2). This comprehensive evaluation illuminates the robustness and practical reliability of LibIHT, highlighting its capability to mitigate adversarial threats effectively in security-critical applications.

Table 4 shows that LibIHT 's hardware-level tracing remains undetected across every anti-instrumentation test, whereas Intel Pin and DynamoRIO are each caught by distinct evasion strategies: the functional-limit checks in `fl_fsbase.c` and `fl_ripsyscall.c` flag only DynamoRIO, illustrating that probes targeting altered processor state affect solely software DBI frameworks; among the code-cache artifact tests, `ca_nx.c` (NX-page check) again detects DynamoRIO, while `ca_smc.c` and `ca_vmleave.c` (self-modifying code and VM-exit patterns) detect only Pin, reflecting tests that exploit Pin's instrumentation model; all three environment artifact checks (`ea_envvar.c`, `ea_mapname.c`, and `ea_pageperm.c`) successfully detect both Pin and DynamoRIO—by identifying DBI-specific environment variables, mapped library names, or abnormal page permissions—but never LibIHT; and in the runtime overhead probes, `ro_jitbr.c` (JIT branch-timing) triggers both Pin and DynamoRIO whereas `ro_jitlib.c` (JIT-library loading) fails to detect any tool, suggesting either unexercised detection logic or insufficient coverage. No adversarial sample ever flags LibIHT, confirming that existing anti-analysis techniques—designed to recognize software-injected hooks, code-cache artifacts, or runtime overhead—are ineffective against hardware-native tracing.

Table 5 then reports, for the samples where each tool successfully executed, two key metrics: block coverage and slowdown factor.

Table 5: Accuracy and Overhead Under Adversarial Benchmarks

	Block Coverage (%)			Slowdown Factor (×)		
Sample	Intel Pin	DynamoRIO	LibIHT	Intel Pin	DynamoRIO	LibIHT
`fl_fsbase.c`	100%	N/A	99.1%	674.09	N/A	1.67
`fl_ripsyscall.c`	100%	N/A	99.3%	518.58	N/A	1.26
`ca_nx.c`	100%	N/A	98.7%	677.57	N/A	1.86
`ca_smc.c`	N/A	100%	98.8%	N/A	170.13	1.84
`ca_vmleave.c`	N/A	100%	99.9%	N/A	1352.34	12.76
`ea_envvar.c`	N/A	N/A	99.9%	N/A	N/A	3.28
`ea_mapname.c`	N/A	N/A	99.9%	N/A	N/A	5.32
`ea_pageperm.c`	N/A	N/A	99.9%	N/A	N/A	5.41
`ro_jitbr.c`	N/A	N/A	99.7%	N/A	N/A	1.51
`ro_jitlib.c`	100%	100%	99.3%	678.28	158.03	14.52

The left half of the table lists the percentage of ground-truth basic blocks captured by each tool—Intel Pin and DynamoRIO maintain nearly 100% coverage when they run, and LibIHT likewise recovers over 98% of blocks across all tests. The right half shows the slowdown factor: Pin slows execution by hundreds to over a thousand times, DynamoRIO by tens to hundreds of times, whereas LibIHT 's slowdown remains below 15× in every scenario. Even in the worst case (`ro_jitlib.c`, where context-switch noise increases overhead to 14.5×), LibIHT outperforms the software baselines by more than an order of magnitude.

Takeaway: Resilience in Adversarial Settings

LibIHT not only evades all common anti-instrumentation checks but also retains high CFG coverage and orders-of-magnitude better performance under adversarial conditions—demonstrating its robustness and efficiency as a hardware-assisted tracing solution.

5 DISCUSSION

The evaluation results suggest that hardware-assisted tracing via LibIHT is a viable and even preferable alternative to traditional software instrumentation in many scenarios. In this section, we reflect on the implications of these findings, discuss the limitations of our approach, and outline opportunities for future work.

Benefits and Implications. An immediate takeaway is that the reduction in overhead (an order of magnitude less than Pin or DynamoRIO) can unlock new uses for dynamic analysis. Techniques that were previously impractical on resource-constrained or real-time systems might become feasible with LibIHT. Moreover, the resilience demonstrated against anti-analysis measures implies that LibIHT could be particularly useful in malware analysis pipelines, where stealth is paramount. By shifting the tracing to hardware and kernel space, we significantly lower the "interference profile" of the analysis tool, which is a new point in the design space for program analysis tools.

Limitations. Despite these advantages, LibIHT is not without trade-offs. One limitation is the loss of certain semantic information - for instance, LibIHT currently logs control-flow branches but not the associated data values or memory accesses. This means analyses that require fine-grained data flow or taint tracking cannot be directly supported in our framework. Another limitation is platform dependence: our implementation relies on Intel-specific features (LBR/BTS). While many modern CPUs have analogous capabilities, additional engineering is required to support other architectures (e.g., ARM's branch record features). A further consideration is that, although LibIHT 's use of LBR/BTS resists all known software anti-instrumentation checks, attackers could eventually target hardware tracing directly-e.g., by overflowing the LBR buffer, exploiting microarchitectural quirks to inject spurious branches, or detecting MSR-access timing anomalies. Over time, such evasion techniques could erode LibIHT 's stealth advantage unless countermeasures (e.g., randomized drain intervals or noise injection) are adopted. We leave the exploration of such targeted attacks to future work.

Future Work. There are several avenues to extend this work. First, enriching trace semantics is a priority: we plan to investigate combining LibIHT with lightweight instrumentation that logs selective additional context (such as function call arguments or memory addresses) to recover some of the lost semantic detail without reintroducing high overhead. Second, porting LibIHT to other platforms (ARM, AMD processors) and evaluating it there would broaden its applicability. Third, a deeper integration with analysis tools (like feeding LibIHT 's output into existing CFG recovery or taint analysis frameworks) could demonstrate end-to-end use cases. Finally, user studies or case studies (e.g., analyzing real malware samples in the wild) would help validate LibIHT 's effectiveness in practical security workflows.

Another promising direction is the use of LibIHT for fuzzing. Modern hardware-assisted fuzzers have employed Intel PT to collect execution coverage efficiently, enabling large-scale exploration of complex programs [6, 26, 30]. However, PT's trace decoding overhead and complexity can introduce bottlenecks in coverage-guided fuzzing workflows. LibIHT, by offering simpler and lower-overhead

control-flow tracing, may serve as an attractive alternative. Future work could benchmark LibIHT against Intel PT-based fuzzers to evaluate trade-offs in accuracy, throughput, and scalability, and to assess whether LibIHT can expand the practicality of hardware-assisted fuzzing in security-critical domains.

Overall, our discussion highlights that LibIHT 's hardware-centric approach shifts some long-standing trade-offs in dynamic analysis. It achieves a new balance between performance, transparency, and fidelity. While there are limitations to address, the approach opens up a promising direction for building low-overhead, anti-evasion program analysis tools.

6 RELATED WORK

In this section, we provide an in-depth review of prior research that utilizes hardware tracing features for program analysis. We categorize the literature into two primary streams: (i) Intel PT-based approaches and (ii) LBR/BTS-based approaches. The following discussion details the advantages and limitations of these approaches and explains how they motivate the design of LibIHT.

6.1 Intel PT-Based Approach

Intel has emerged as a powerful mechanism for capturing fine-grained execution information. Several tools have integrated Intel PT into their frameworks, such as Linux Perf [16] for performance analysis, and specialized systems for control flow integrity monitoring [13, 14]. Intel PT offers high-precision, instruction-level tracing that enables detailed reconstruction of program execution paths-a valuable feature for debugging and reverse engineering.

Despite these strengths, Intel PT-based systems face significant challenges. The voluminous trace data generated requires extensive storage and imposes heavy computational overhead during decoding and analysis. This high data granularity often leads to performance bottlenecks, making real-time analysis difficult, especially on resource-constrained platforms. Although researchers have proposed techniques such as selective tracing and data compression [13, 14] to alleviate these issues, such methods frequently introduce trade-offs that compromise trace fidelity or add complexity to the analysis pipeline. Consequently, while Intel PT delivers unmatched detail, its overhead and scalability concerns restrict its practical applicability in scenarios requiring low-latency or continuous monitoring.

6.2 LBR and BTS-Based Research

An alternative line of research has focused on leveraging the more lightweight hardware features available in modern processors, namely the LBR and BTS. LBR provides a circular buffer that records the most recent branch transitions, offering a succinct snapshot of control-flow events. BTS extends this capability by logging branch events into a dedicated memory buffer over a longer period. These mechanisms naturally produce substantially less data compared to Intel PT, thereby reducing storage and processing overhead.

Prior studies have applied LBR/BTS-based tracing across various domains. For example, Willems et al. [29] employed LBR for malware analysis by detecting anomalies in branch patterns, while other works have explored its use for exploit mitigation [7, 17, 23, 32] and performance profiling [2, 19]. Although these efforts demonstrate that processor-based tracing can effectively capture control-flow anomalies, many existing solutions remain narrowly focused, addressing only specific security threats or performance issues in isolated settings. Moreover, several approaches are presented merely as proof-of-concept prototypes, lacking the robustness and scalability needed for general-purpose program analysis. Common limitations include fixed buffer sizes, constrained tracing durations, and challenges in integrating with binary analysis tools.

LibIHT addresses these limitations by providing a unified, modular framework that integrates both LBR and BTS tracing. Unlike prior approaches that are either tailored to specific applications or exist solely as isolated prototypes, LibIHT offers a robust kernel-space implementation paired with flexible user-space APIs. This design enables efficient branch-level trace capture with low overhead while facilitating seamless integration with diverse security and debugging workflows. In doing so, LibIHT overcomes the issues of scalability, inflexibility, and high resource consumption that have limited earlier tools, thereby advancing the state of the art in hardware-assisted dynamic analysis.

7 CONCLUSION

In this paper, we presented LibIHT, a hardware-assisted dynamic analysis framework that leverages CPU branch tracing features to achieve efficient and stealthy program tracing. LibIHT directly addresses the limitations of software-based instrumentation: in our experiments, it delivered order-of-magnitude performance improvements and resisted a range of anti-analysis techniques, all while accurately reconstructing program control-flow. By relocating tracing into the processor and kernel, LibIHT minimizes its footprint in the target's execution, thereby avoiding the perturbations and detections that plague traditional tools.

Our work contributes a novel perspective on program analysis—one that exploits modern hardware capabilities to balance the long-standing unexplored trade-off between analysis fidelity and overhead. The positive results from LibIHT 's evaluation suggest that hardware-centric tracing can be a practical foundation for building next-generation analysis tools. We believe this approach will enable security researchers and software engineers to analyze programs (including malware) in scenarios that were previously too slow or too risky using conventional methods. As hardware support for tracing continues to evolve, frameworks like LibIHT can be extended and adapted to further close the gap between transparent, high-fidelity analysis and real-world performance constraints.

ACKNOWLEDGMENTS

Funding acknowledgment: This material is based upon work supported by the National Science Foundation under Grant No. CNS-2343611. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work was supported in part by the Semiconductor Research Corporation (SRC) and DARPA.

REFERENCES

[1] Saed Alrabaee, Mourad Debbabi, Paria Shirani, Lingyu Wang, Amr Youssef, Ashkan Rahimian, Lina Nouh, Djedjiga Mouheb, He Huang, and Aiman Hanna. 2020. Binary Analysis Overview. In *Binary Code Fingerprinting for Cybersecurity.*

Vol. 78. Springer International Publishing, Cham, 7–44. https://doi.org/10.1007/978-3-030-34238-8_2 Series Title: Advances in Information Security.

[2] Joy Arulraj, Guoliang Jin, and Shan Lu. 2014. Leveraging the short-term memory of hardware to diagnose production-run software failures. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM, Salt Lake City Utah USA, 207–222. https://doi.org/10.1145/2541940.2541973

[3] Thoms Ball. 1999. The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes* 24, 6 (Nov. 1999), 216–234. https://doi.org/10.1145/318774.318944

[4] Sanjay Bhansali, Wen-Ke Chen, Stuart De Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. 2006. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments*. ACM, Ottawa Ontario Canada, 154–163. https://doi.org/10.1145/1134760.1220164

[5] Derek Bruening and Saman Amarasinghe. 2004. Efficient, transparent, and comprehensive runtime code manipulation. *Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science* (2004). https://www.burningcutlery.com/derek/docs/phd.pdf Publisher: Massachusetts Institute of Technology, Department of Electrical Engineering ….

[6] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. 2019. PTrix: Efficient Hardware-Assisted Fuzzing for COTS Binary. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. ACM, Auckland New Zealand, 633–645. https://doi.org/10.1145/3321705.3329828

[7] Yueqiang CHENG, Zongwei ZHOU, Yu MIAO, Xuhua DING, and Robert H. DENG. 2014. ROPecker: A Generic and Practical Approach For Defending Against ROP Attack. *NDSS Symposium 2014: Proceedings of the 21st Network and Distributed System Security Symposium, San Diego, February 23-26* (Feb. 2014), 1–14. https://doi.org/10.14722/ndss.2014.23156

[8] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. 2019. SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. ACM, Auckland New Zealand, 15–27. https://doi.org/10.1145/3321705.3329819

[9] Daniele Cono D'Elia, Lorenzo Invidia, Federico Palmaro, and Leonardo Querzoni. 2022. Evaluating Dynamic Binary Instrumentation Systems for Conspicuous Features and Artifacts. *Digital Threats: Research and Practice* 3, 2 (June 2022), 1–13. https://doi.org/10.1145/3478520

[10] Peter Feiner, Angela Demke Brown, and Ashvin Goel. 2012. Comprehensive kernel instrumentation via dynamic binary translation. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. ACM, London England, UK, 135–146. https://doi.org/10.1145/2150976.2150992

[11] Ailton Santos Filho, Ricardo J. Rodríguez, and Eduardo L. Feitosa. 2020. Reducing the Attack Surface of Dynamic Binary Instrumentation Frameworks. In *Developments and Advances in Defense and Security*. Vol. 152. Springer Singapore, Singapore, 3–13. Series Title: Smart Innovation, Systems and Technologies.

[12] Ailton Santos Filho, Ricardo J. Rodríguez, and Eduardo L. Feitosa. 2022. Evasion and Countermeasures Techniques to Detect Dynamic Binary Instrumentation Frameworks. *Digital Threats* 3, 2 (Feb. 2022), 11:1–11:28. https://doi.org/10.1145/3480463

[13] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. *ACM SIGPLAN Notices* 52, 4 (May 2017), 585–598. https://doi.org/10.1145/3093336.3037716

[14] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. ACM, Scottsdale Arizona USA, 173–184. https://doi.org/10.1145/3029806.3029830

[15] Intel [n. d.]. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[16] Linux Perf [n. d.]. Perf events and tool security — The Linux Kernel documentation. https://docs.kernel.org/admin-guide/perf-security.html

[17] Weijie Liu, Ximeng Liu, Zhi Li, Bin Liu, Rongwei Yu, and Lina Wang. 2022. Retrofitting LBR Profiling to Enhance Virtual Machine Introspection. *IEEE Transactions on Information Forensics and Security* 17 (2022), 2311–2323. Publisher: IEEE.

[18] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices* 40, 6 (June 2005), 190–200. https://doi.org/10.1145/1064978.1065034 Publisher: Association for Computing Machinery (ACM).

[19] Gabriel Marin, Alexey Alexandrov, and Tipp Moseley. 2021. Break dancing: low overhead, architecture neutral software branch tracing. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM, Virtual Canada, 122–133. https://doi.org/10.1145/3461648.3463853

[20] Nicholas Nethercote. 2004. *Dynamic binary analysis and instrumentation*. Technical Report. University of Cambridge, Computer Laboratory. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-606.html

[21] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM SIGPLAN Notices* 42, 6 (June 2007), 89–100. https://doi.org/10.1145/1273442.1250746 Publisher: Association for Computing Machinery (ACM).

[22] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2015. *Principles of program analysis*. springer.

[23] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2013. Transparent {ROP} exploit mitigation using indirect branch tracing. In *22nd USENIX Security Symposium (USENIX Security 13)*. 447–462.

[24] Seongwoo Park and Yongsu Park. 2024. Detection Techniques for DBI Environment in Windows. *Electronics* 13, 5 (Jan. 2024), 871. https://doi.org/10.3390/electronics13050871 Number: 5 Publisher: Multidisciplinary Digital Publishing Institute.

[25] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontana, Fabio Gritti, and Stefano Zanero. 2017. Measuring and Defeating Anti-Instrumentation-Equipped Malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Michalis Polychronakis and Michael Meier (Eds.). Vol. 10327. Springer International Publishing, Cham, 73–96.

[26] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. {kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels. In *26th USENIX security symposium (USENIX Security 17)*. 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[27] Kevin Scott, Naveen Kumar, Siva Velusamy, Bruce Childers, Jack W. Davidson, and Mary Lou Soffa. 2003. Retargetable and reconfigurable software dynamic translation. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 36–47.

[28] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, and Christopher Kruegel. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 138–157.

[29] Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Thorsten Holz, and Amit Vasudevan. 2012. Down to the bare metal: using processor features for binary analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, Orlando Florida USA, 189–198. https://doi.org/10.1145/2420950.2420980

[30] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. 2018. Ptfuzz: Guided fuzzing with processor trace feedback. *IEEe Access* 6 (2018), 37302–37313. https://ieeexplore.ieee.org/abstract/document/8399803/ Publisher: IEEE.

[31] Zhechko Zhechev. 2018. *Security evaluation of dynamic binary instrumentation engines*. PhD Thesis. Department of Informatics, Technical Universtity of Munich Munich, Germany.

[32] Hongwei Zhou, Keda Kang, and Jinhui Yuan. 2019. HardStack: Prevent Stack Buffer Overflow Attack with LBR. In *2019 International Conference on Intelligent Computing, Automation and Systems (ICICAS)*. IEEE, 888–892.

A APPENDIX

A.1 Workload Commands for Motivating Experiment

In Table 1, we simply measure the runtime overhead introduced by dynamic binary instrumentation tools, we selected a set of commonly used `coreutils` programs that represent typical system workloads, including I/O, file system metadata access, and CPU-bound operations. Each command was chosen to be deterministic and environment-stable.

The exact commands used in the evaluation are listed below:

- `ls`: `ls -lah /bin > /dev/null`
- `dd`: `dd if=/dev/zero of=/dev/null bs=1M count=100`
- `echo`: `echo 'hello world'`
- `sort`: `sort < /etc/passwd > /dev/null`
- `wc`: `wc -l < /etc/passwd > /dev/null`
- `cat`: `cat /etc/passwd > /dev/null`

All output from these commands was suppressed to ensure that measurement reflected only core execution time, without terminal

I/O interference. Each workload was run in three configurations: native, under Intel Pin (`pin -- <command>`), and under DynamoRIO (`drrun -- <command>`).