ERCIYES UNIVERSITY
FACULTY OF ENGINEERING

TRACEFLOW: RUNTIME BEHAVIOR ANALYSIS OF LINUX BINARIES WITH DYNAMORIO

Prepared by

Ahmet Göker
1030510380

Supervisor
Res. Asst. Fuat TÖRE

Computer Engineering
Undergraduate Thesis

October 2025
KAYSERI

ACKNOWLEDGMENTS

I would like to thank my supervisor, Res. Asst. Fuat TÖRE, for his guidance during this study. I am also grateful to the Department of Computer Engineering at Erciyes University for providing the academic setting in which this work was developed.

This revised manuscript was prepared with particular care for research integrity. Preserved outputs are reported as preserved outputs, proposed corrections are identified as proposals, and no experiment is described as reproduced when the required binary, build record, and raw trace were unavailable.

Ahmet Göker

Kayseri, October 2025

TRACEFLOW: RUNTIME BEHAVIOR ANALYSIS OF LINUX BINARIES WITH DYNAMORIO

Ahmet Göker
Erciyes University, Computer Engineering
Supervisor: Res. Asst. Fuat TÖRE

ABSTRACT

This thesis examines a DynamoRIO based dynamic binary instrumentation prototype for observing the runtime behavior of Linux binaries. The prototype counts executed application instructions, reports memory read and write operands, follows entry to and return from the main function, and labels a small set of external call targets. Preserved experimental outputs were re-examined, and the implementation visible in source screenshots was audited against the documented DynamoRIO interfaces and reference clients.

Two experimental records could be verified from the available evidence. The external call run counted 76 handler events, although only eight named targets were printed individually. The memory run recorded 2,274 executed application instructions, 359 read operands, and 353 write operands. These totals correspond to 15.8 reads and 15.5 writes per 100 instructions. Their sum is not a count of unique instructions because a read-modify-write instruction may contribute to both categories.

The audit found that the memory callback receives the instruction program counter rather than the effective data address, that main-function scope is represented by a process-global Boolean value, and that external calls are classified from the low 16 bits of a target address. The preserved results therefore support a bounded prototype observation. They do not establish malware detection accuracy, classification performance, or low overhead. This thesis separates observation from inference and proposes an auditable redesign based on effective address computation, thread-local scope state, module-relative target resolution, raw trace retention, and repeated measurements.

Keywords: dynamic binary instrumentation, DynamoRIO, Linux, runtime analysis, memory tracing

TABLE OF CONTENTS

TRACEFLOW: RUNTIME BEHAVIOR ANALYSIS OF LINUX BINARIES WITH DYNAMORIO

4. CHAPTER
EXPERIMENTAL METHOD AND EVIDENCE

5. CHAPTER
RESULTS

6. CHAPTER
DISCUSSION

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ABI	Application Binary Interface
ASLR	Address Space Layout Randomization
BB	Basic Block
DBI	Dynamic Binary Instrumentation
ELF	Executable and Linkable Format
GOT	Global Offset Table
PC	Program Counter
PIE	Position Independent Executable
PLT	Procedure Linkage Table
RMW	Read-Modify-Write
TLS	Thread-Local Storage

INTRODUCTION

Software rarely reveals its complete behavior through source code alone. The final executable contains compiler decisions, library bindings, loader activity, and control transfers that may not be obvious in the source representation. Source code may also be unavailable. Runtime observation is therefore useful in performance work, compatibility testing, reverse engineering, and defensive security analysis. Dynamic binary instrumentation, abbreviated as DBI, provides this observation by placing analysis code into an executing binary without requiring the original program to be rebuilt. DynamoRIO is one established DBI framework. It copies application code into a managed code cache, transforms that code, and preserves application semantics while client callbacks collect observations [1–3].

This thesis studies TraceFlow, a small DynamoRIO client prototype for Linux binaries. The preserved implementation has four intended functions. It counts application instructions, identifies read and write operands, marks entry to and return from the program's `main` function, and reports selected external call targets. These functions are useful as a teaching instrument because they expose the boundary between a static instruction, a runtime event, and an interpretation made by an analyst. They also create several opportunities for subtle error. A program counter is not a data address. A memory operand is not necessarily a unique memory instruction. A target address suffix is not a stable symbol identity. A process-global flag does not model thread-local control flow.

The original project materials included source screenshots and two preserved output runs, but they did not include a complete source tree, build manifest, versioned binary, command transcript, or raw trace. This evidence boundary determines what can be claimed. The numbers in the preserved outputs can be transcribed and checked for arithmetic consistency. The visible code can be audited against the documented DynamoRIO interfaces. The experiments cannot be described as independently reproduced because

the necessary artifacts were not retained. This distinction is central to the revised study.

Problem statement

A runtime tracing tool can produce convincing output while measuring something different from its labels. The error may remain unnoticed when the output contains plausible hexadecimal addresses and familiar function names. TraceFlow’s memory callback illustrates the problem. The visible instrumentation passes the application instruction address obtained from `instr_get_app_pc`. The reporting code describes that value as a memory address. DynamoRIO provides a separate helper, `drutil_insert_get_mem_addr`, for calculating an operand’s effective address [4]. Without that calculation, the trace identifies the instruction responsible for an access but not the location accessed.

Scope is another source of ambiguity. The preserved client uses a Boolean variable named `in_main`. Entry to `main` sets the variable and a return event clears it. An instrumentation callback then decides whether a basic block should be instrumented. This design combines translation time and execution time state. A translated block may remain in DynamoRIO’s cache and execute later under a different state. The global variable is also shared by all threads. DynamoRIO’s extension manager provides thread-local and callback-ordering facilities intended for such cases [5]. A reliable design must say whether scope is decided while a block is translated or while it executes, and it must preserve that decision separately for each thread.

The final ambiguity concerns external calls. The prototype labels targets from the low 16 bits of a callee address and compares those bits with a fixed interval associated with one observed executable. Address Space Layout Randomization changes load addresses between processes. Position Independent Executables and shared libraries further separate an absolute address from its file offset. The ELF and System V AMD64 specifications describe symbol resolution through the Procedure Linkage Table and Global Offset Table [6, 7]. A stable classifier must first identify the containing module and then derive a module-relative offset or resolved symbol. The low bits alone are not a general substitute.

Aim and research questions

The aim of this thesis is to evaluate what the preserved TraceFlow prototype actually measures and to define a technically sound path from that prototype to an auditable Linux runtime tracer. The goal is not to claim a completed malware detector. Malware analysis requires controlled samples, ground truth, containment, repeatable acquisition, and documented handling procedures [8]. None of those requirements is satisfied by a pair of benign demonstration runs. The defensible contribution is narrower and more useful: a source-aware audit, a precise interpretation of the surviving measurements, and a corrected instrumentation design.

The study addresses four research questions:

1. What runtime quantities are directly supported by the preserved output, and which claims cannot be recovered from it?
2. Does the visible instrumentation implement the meanings assigned to its fields, especially effective memory address and main-function scope?
3. How should the prototype be redesigned using documented DynamoRIO facilities so that its records are thread-aware, address-correct, and reproducible?
4. What can the numerical results show about the observed test runs without extending them into unsupported performance or malware claims?

Scope and contribution

The scope is Linux on the x86-64 architecture, ELF executables, DynamoRIO client instrumentation, direct and indirect control transfers, and memory operands visible to the instrumentation API. Kernel execution, hardware tracing, packed malware unpacking, inter-process correlation, and statistical malware classification are outside the evaluated system. The study refers to defensive security use cases, but the preserved test artifacts are treated as benign programs. This keeps the analysis legal, bounded, and reproducible in principle.

The first contribution is an evidence ledger. Each reported result is classified as directly

observed, arithmetically derived, inferred, proposed, or unavailable. This prevents a reconstructed explanation from being mistaken for an experimental fact. The second contribution is a code audit that maps visible implementation choices to DynamoRIO semantics. The third contribution is a corrected architecture. It uses the staged event model from `drmgr`, register management from `drreg`, effective address calculation from `drutil`, and per-thread buffers similar to the official memory-trace examples [5, 9, 10]. The fourth contribution is a set of reporting rules for reading the preserved measurements without overstating them.

Figure I.1 summarizes this thesis's evidence chain. The hand-drawn appearance is deliberate. It distinguishes conceptual reasoning from the exact quantitative charts used later. The drawing is generated by deterministic TikZ code, not by an image model. Its slight rotations, doubled outlines, and uneven connectors are fixed in the source, so the same build produces the same strokes.

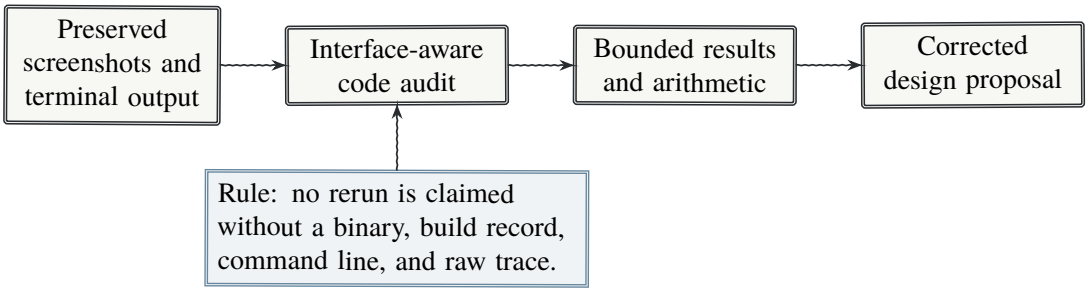

Figure I.1. Evidence chain used throughout the thesis. Conceptual figure, not an empirical result.

Research integrity and limitations

The manuscript uses three safeguards. First, preserved numbers remain unchanged. If an output line says that a handler ran 76 times, the thesis reports 76 rather than forcing the value to agree with a list of eight displayed functions. Second, derived values show their formula and denominator. The memory-event rate is calculated per 100 executed application instructions. It is not called a percentage of instructions with memory access because the numerator counts operands and may count one instruction more than once. Third, all proposed corrections are grammatically separated from implemented behavior. Words such as *should*, *would*, and *proposed* indicate design work that was not present in

the preserved executable.

Several limitations remain. Exact DynamoRIO, compiler, Linux distribution, and kernel versions were not preserved. The target binaries and their cryptographic hashes were not retained. The output includes one run for each demonstrated mode, so variance and confidence intervals cannot be calculated. Native and instrumented runtimes were not recorded, which means instrumentation overhead cannot be quantified. The screenshots expose only portions of the client, so the audit cannot establish whole-program compilation or absence of unseen logic. These limitations do not invalidate the screenshots. They restrict the questions the screenshots can answer.

Thesis organization

Chapter 1 presents dynamic binary instrumentation, Linux executable loading, related tools, and the distinction between program counters and effective addresses. Chapter 2 reconstructs the TraceFlow design from the preserved materials. Chapter 3 audits the visible code and records the severity and consequence of each finding. Chapter 4 defines a reproducible experimental method and an evidence model. Chapter 5 reports the exact preserved results. Chapter 6 discusses interpretation, validity, overhead, security relevance, and engineering priorities. Chapter 7 concludes the study and lists concrete recommendations. The appendices preserve the visible source excerpts, corrected design fragments, screenshots, and a submission-oriented reproducibility checklist.

1. CHAPTER

TECHNICAL BACKGROUND AND RELATED WORK

1.1. Dynamic binary instrumentation

Dynamic binary instrumentation observes and modifies executable code while a program runs. A DBI system normally gains control before an application code fragment executes, decodes the fragment, inserts analysis operations, and places the resulting fragment in a managed code cache. Later executions can use the cached fragment without repeating every translation step. This architecture gives the client access to runtime control flow and operands while keeping the target binary unchanged on disk. The framework must preserve registers, flags, stack state, signals, system interaction, and control transfers well enough that the application behaves as if no instrumentation were present [1, 2, 11].

DynamoRIO exposes this mechanism through callbacks and extension libraries. A client can register for basic-block construction, thread initialization, module loading, process exit, and other events. It can inspect an instruction list and insert clean calls or inline analysis. The code cache means there are two relevant times. Translation time is when a block is decoded and instrumented. Execution time is when a thread executes the transformed block. A client that mixes state from these times can create behavior that depends on cache history rather than only on application execution.

Pin and Valgrind are established alternatives. Pin provides a high-level instrumentation interface and supports instruction, trace, routine, and image concepts [12]. Valgrind translates machine code into an intermediate representation and is well known for heavyweight analyses such as memory checking [13]. QEMU emphasizes system and user-mode emulation across architectures [14]. Compiler-assisted tools such as

AddressSanitizer insert checks at build time and can offer lower cost when source and a compatible toolchain are available [15]. These systems serve overlapping but different conditions. TraceFlow uses DynamoRIO because its research question concerns a binary that can be instrumented without source recompilation.

Table 1.1. Position of TraceFlow among related runtime analysis approaches.

Approach	**Input requirement**	**Primary strength**	**Relevance to this thesis**
DynamoRIO	Executable binary	Transparent runtime instrumentation with extensible callbacks and a code cache	Framework used by TraceFlow
Pin	Executable binary	Mature instrumentation abstractions for instruction and routine analysis	Comparable DBI design point
Valgrind	Executable binary	Rich intermediate representation and heavyweight checking	Illustrates the cost and capability tradeoff
AddressSanitizer	Source or compiler IR	Efficient compiler-inserted memory checks	Useful when rebuilding the target is possible
QEMU	Binary or guest system	Cross-architecture translation and emulation	Broader than TraceFlow's same-architecture tracing scope

1.2. Basic blocks and instrumentation granularity

A basic block is a sequence of instructions with one entry and a control transfer at the end. DynamoRIO commonly offers the block as an instruction list during construction. A client can count blocks, inspect every application instruction, or insert analysis at selected points. The chosen granularity affects meaning and overhead. Counting a block once is not equal to counting its instructions. Inserting one callback for every operand can produce several callbacks for one instruction. Calling a formatted output function from every callback adds cost that belongs to the tracer rather than the target.

The official examples separate instrumentation from reporting. For memory tracing, the reference design records compact entries in a per-thread buffer and performs heavier processing when the buffer is full or the thread exits [10, 16]. This pattern reduces transitions into clean-call code and avoids a shared output operation on every event. It

also makes the event schema explicit. A record can contain an instruction address, effective address, access type, size, thread identifier, and sequence number. TraceFlow's preserved code takes the simpler instructional path of per-event clean calls. That choice is acceptable for a prototype, but its overhead and concurrency consequences must be reported.

1.3. Instruction address and effective address

The instruction program counter and a memory operand's effective address answer different questions. The program counter identifies where the instruction resides in executable memory. The effective address identifies the data location selected by that instruction during a particular execution. On x86-64, the effective address may depend on a base register, an index register, a scale, a displacement, and in some cases the instruction pointer [17]. The same static instruction can access a different data location on every loop iteration.

Consider the instruction in Equation 1.1. If the scale is four, the base register contains `0x7000`, the index contains three, and the displacement is eight, the effective address is `0x7014`. The instruction address could be `0x401230`; it does not participate in this example's data-address calculation.

$$EA = Base + Index \times Scale + Displacement \tag{1.1}$$

DynamoRIO exposes the application program counter through `instr_get_app_pc`. It exposes an effective-address calculation helper through `drutil_insert_get_mem_addr`, which writes the computed address to a reserved destination register before the application instruction runs [4]. A correct memory tracer may retain both fields because both are useful. The program counter supports attribution to code. The effective address supports analysis of the data region touched. Labeling one as the other removes this distinction.

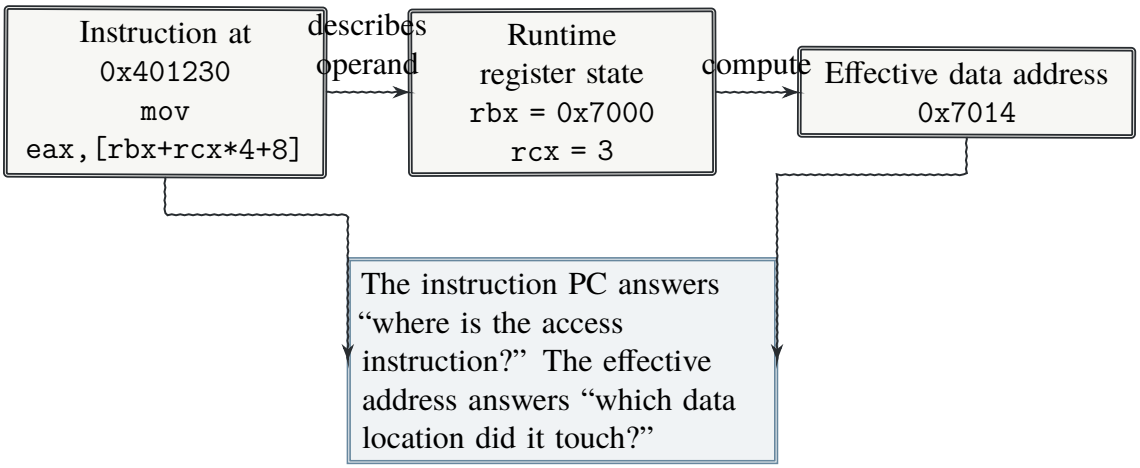

Figure 1.1. Program counter and effective address are related but distinct observations.

1.4. ELF loading, PLT, and GOT

Linux commonly stores executables and shared objects in ELF format. The file contains segments used by the loader, sections used by linkers and tools, symbol information when available, and relocation records. A dynamically linked call may transfer through a Procedure Linkage Table entry. The corresponding Global Offset Table entry is updated with a resolved address according to the loader's relocation process. The exact sequence depends on linkage mode, compiler options, and whether lazy binding is enabled [6, 7].

Address Space Layout Randomization changes where modules are mapped. The process mapping list in `/proc/<pid>/maps` reports address intervals, permissions, file offsets, device and inode information, and optional pathnames [18]. A runtime target should therefore be represented as at least a module identity plus a module-relative offset. If symbols are available, the tracer may add a symbol name. An absolute address is still useful within one run, but it is not a stable cross-run identifier.

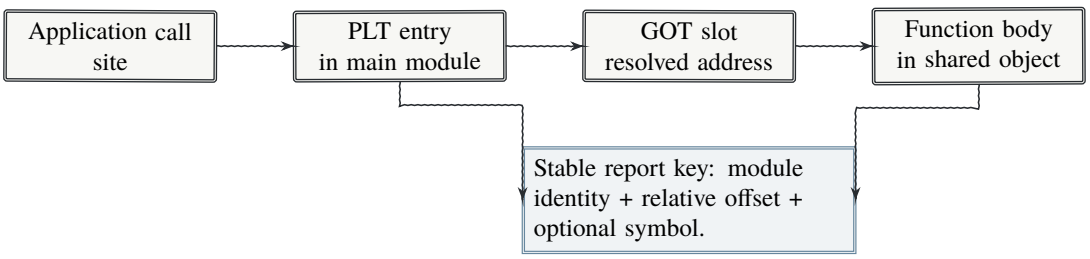

Figure 1.2. Simplified dynamic call resolution path for a Linux ELF program.

Taking only `target & 0xFFFF` discards the module identity and most address bits. It may appear to produce a file-like offset in one small non-PIE executable, but it is not equivalent to subtracting the module base. Two unrelated targets can share the same low 16 bits. A function may also be reached through its resolved library address rather than through a PLT stub. These cases make suffix-based naming unsuitable for general tracing.

1.5. Control-flow scope and concurrency

Limiting tracing to `main` can reduce loader noise and focus a teaching experiment on application logic. The boundary is less simple than it first appears. The main function calls libraries, callbacks may re-enter the program, signals may interrupt it, and additional threads may begin while the original thread is still within `main`. A single Boolean variable can represent only one process-wide state. It cannot record nested entry depth or assign state to a particular thread.

DynamoRIO's `drmgr` library offers a staged callback system and thread-local storage. Its insertion phase can reserve resources and place runtime tests in a defined order [5]. The `drreg` extension manages scratch registers and application values when analysis code must compute addresses or branch on runtime state [9]. For a main-scoped tracer, a per-thread depth or scope flag is safer than a global variable. If the policy is specifically "events executed by the initial thread after main entry and before its matching return," the tool should say so. If calls made by `main` are included, the instrumentation must remain active in those callees while the runtime scope flag is set.

Translation-time filtering creates another issue. Suppose a library block is translated before `main`, while a global flag is false. If the instrumentation callback returns without inserting analysis, the cached block can later execute during `main` without trace operations. The inverse can also occur if a block is first translated while the flag is true and later executes after the scope has ended. A runtime predicate inserted into the block avoids this cache-history dependence. Alternatively, the client can flush relevant code when scope changes, but flushing is more disruptive and requires careful synchronization.

1.6. Runtime tracing in defensive security

Runtime observation is useful in defensive analysis because it can reveal behavior that static inspection misses, such as dynamically selected targets, unpacked code, memory permission changes, and process interaction. The MITRE ATT&CK entry for `/proc/<pid>/mem` illustrates one Linux-specific process-injection path and the importance of observing process memory access in context [19]. A generic operand tracer, however, does not by itself detect that technique. It must associate the access with a system interface, target process, permissions, and behavior sequence.

NIST guidance treats malware handling as a controlled incident-response activity that includes preparation, detection, containment, eradication, and recovery [8]. A research system that claims malware detection also needs representative samples, labels, a decision rule, and evaluation metrics. TraceFlow's surviving evidence contains no malware corpus and no classifier. The appropriate security claim is therefore that DBI can support later behavioral analysis by producing runtime evidence. Whether that evidence supports a detector is a separate experimental question.

1.7. Chapter summary

DBI provides detailed runtime visibility, but the meaning of a record depends on when and how it was captured. Program counters, effective addresses, absolute targets, module-relative offsets, translation state, and execution state must remain distinct. The next chapter reconstructs TraceFlow using those distinctions and defines the boundary between visible implementation and proposed architecture.

2. CHAPTER

TRACEFLOW SYSTEM DESIGN

2.1. Reconstruction basis

TraceFlow is reconstructed from the thesis PDF, source-code screenshots, and terminal-output screenshots supplied with the project. The reconstruction uses only elements visible in those materials. It does not assume the contents of missing functions or recover a compilable repository. This matters because a screenshot can establish that a line was shown, but it cannot establish which revision produced a binary or whether another source file changed its behavior.

The visible client follows a conventional DynamoRIO lifecycle. Initialization registers callbacks and initializes symbol support. A module-load callback searches for the main executable and looks up `main`. Basic-block instrumentation iterates over instructions, inserts clean calls, and updates counters. A return or control-transfer path changes the intended main-function scope. An exit callback prints aggregate statistics. The two preserved runs demonstrate an external-call mode and a memory-operand mode.

Table 2.1 records the available artifacts and the claims each can support. This ledger is used later when assigning confidence to a result.

Table 2.1. Artifact ledger for the reconstructed prototype.

Artifact	Status	Supported use
Thesis PDF	Available	Project narrative, visible code fragments, and preserved figures
Source screenshots	Available, partial	Line-level semantic audit of the visible functions
Terminal screenshots	Available	Exact displayed addresses, counters, and named targets
Complete source tree	Unavailable	Compilation and whole-program review cannot be claimed
Target binaries	Unavailable	Rerun, hash verification, and disassembly comparison cannot be claimed
Build manifest and command log	Unavailable	Toolchain and flag reconstruction cannot be claimed
Raw event trace	Unavailable	Event-level reaggregation and sequence analysis cannot be claimed

2.2. Observed component model

Figure 2.1 presents the observed design. Solid conceptual boxes represent components visible in the preserved materials. The event collector receives instruction or control-flow events. Shared state contains counters, output handles, `main` metadata, and the global scope flag. The reporting path writes individual lines and final totals. Because the surviving screenshots cover only part of the implementation, the figure does not add an unseen database, classifier, model, or user interface.

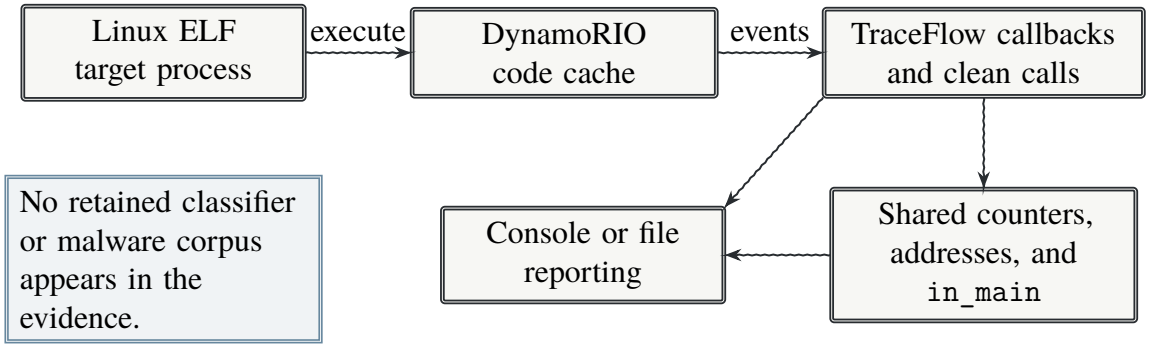

Figure 2.1. Observed TraceFlow component model reconstructed from preserved materials.

2.3. Event taxonomy

The thesis separates event names from their validated meaning. An *instruction event* is

a callback associated with an application instruction execution. A *read-operand event* corresponds to an operand that the decoder marks as a memory source. A *write-operand event* corresponds to an operand marked as a memory destination. A single instruction can contain more than one relevant operand. A read-modify-write operand can contribute to both categories. The terms therefore describe instrumentation events rather than a partition of all executed instructions.

An *external-call handler event* is an invocation of the prototype's external-call callback. The preserved total is incremented before the visible target-name filters. It is broader than the set of eight printed targets. The term *external* is retained because it appears in the prototype, but its exact predicate cannot be proven from the incomplete source. A robust implementation would define whether external means outside the current basic block, outside the main executable module, or resolved to a shared object.

A *main-scope event* is intended to occur after entry to `main` and before return from `main`. In the preserved implementation, this is an intended meaning rather than a fully reliable invariant. The global flag and translation-time gate weaken the invariant. Chapters 3 and 4 treat this scope as a testable property rather than an assumption.

2.4. Address model

TraceFlow handles four address concepts. The module base is the load address reported for the main executable. The `main` address is obtained through symbol lookup. The instruction program counter identifies the code instruction currently being processed. The call target identifies a destination of a control transfer. A fifth concept, the effective data address, is required for memory tracing but is not actually calculated by the visible code.

The preserved external-call run reports a module base of `0x000079a333400000` and a `main` address of `0x000079a333401269`. Their difference is `0x1269`. This arithmetic is module-relative and remains consistent within the screenshot. The target classifier then takes a different path and masks callee addresses to 16 bits. That operation happens to yield values such as `0x10e0` and `0x1170`, but it should not be confused with subtraction of the module base.

$$Offset_{module} = Address_{runtime} - Base_{module} \tag{2.1}$$

Equation 2.1 defines the appropriate starting point when an address belongs to a known module. The calculation must verify membership first. If $Address_{runtime}$ lies outside the module interval, subtraction can produce a meaningless value. The result should also carry a module build identity or file hash when cross-run comparison is required.

2.5. Proposed record schema

The preserved console lines are readable, but they are not sufficient as a durable trace format. A corrected design should write structured raw records and generate human-readable summaries afterward. Table 2.2 defines a minimal record. Fields marked optional apply only to relevant event types.

Table 2.2. Proposed versioned event record for an auditable TraceFlow implementation.

Field	Representation	Purpose
schema_version	Unsigned integer	Allows parsers to reject or migrate incompatible records
run_id	String or UUID	Associates events with one invocation and metadata file
sequence	64-bit integer	Preserves total event order as emitted by the tracer
thread_id	Integer	Separates thread-local execution and scope
event_type	Enumeration	Instruction, memory read, memory write, call, return, or module event
instruction_pc	64-bit address	Identifies the instruction responsible for the event
effective_address	64-bit address, optional	Identifies accessed data after runtime calculation
access_size	Integer, optional	Records the operand width in bytes
target_address	64-bit address, optional	Records a control-transfer destination
module_id	String, optional	Associates an address with a loaded binary image
module_offset	64-bit integer, optional	Supports ASLR-independent comparison
main_depth	Integer	Makes runtime scope and nesting explicit

Run metadata should be stored separately but referenced by `run_id`. It should include the UTC timestamp, host architecture, Linux distribution and kernel, DynamoRIO version, client commit, compiler and flags, command line, environment variables that affect loading, target path, cryptographic target hash, output hash, and status code. These fields

are not decorative. Without them, a later reviewer cannot determine whether two output differences came from the target, the tracer, the loader, or the environment.

2.6. Corrected processing pipeline

The proposed pipeline in Figure 2.2 retains the prototype's educational purpose while correcting its measurement model. Module and symbol callbacks build a module table. A thread initialization callback allocates thread-local state and a buffer. Basic-block analysis identifies instrumentation sites. The insertion stage reserves registers and inserts runtime code. Memory operands use `drutil_insert_get_mem_addr`. Control transfers retain the full target and resolve it against the module table. A runtime scope value is read from thread-local storage. The fast path writes compact records; a flush path persists batches and updates summary counters.

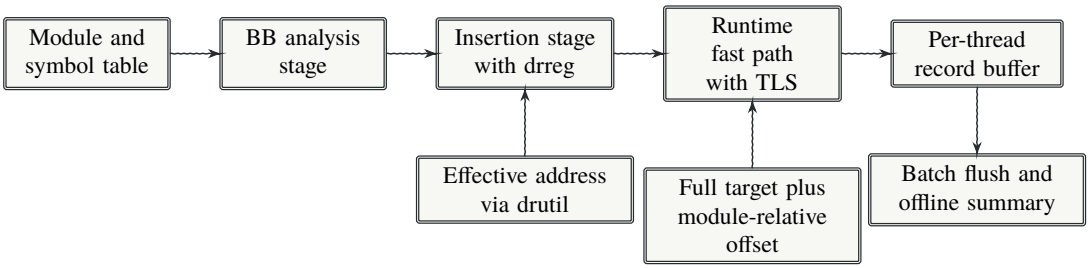

Figure 2.2. Proposed corrected TraceFlow processing pipeline.

2.7. Requirements

The reconstructed and corrected designs can be evaluated against explicit requirements:

1. Every recorded memory access shall distinguish instruction PC from effective data address.
2. Every event shall carry a thread identity, and scope shall be tracked per thread.
3. Every call target shall retain its full runtime address. Module-relative fields shall be added only after module-membership validation.
4. Aggregate counters shall be derived from the same raw records used for detailed reporting.

5. Unnamed and filtered events shall remain countable so that totals reconcile.

6. The tracer shall not print from the per-operand fast path unless a diagnostic mode explicitly enables it.

7. Each run shall produce metadata and cryptographic hashes sufficient for later comparison.

8. Failure to initialize required extensions, open output, resolve mandatory symbols, or flush records shall produce an explicit nonzero status or error record.

These requirements make the meaning of success observable. A console filled with addresses is not sufficient. A successful run is one in which the records conform to the schema, totals reconcile, failures are visible, and another researcher can identify the software and target that produced them.

2.8. Chapter summary

TraceFlow's preserved structure is a compact DynamoRIO client with shared state and per-event reporting. The available evidence supports a useful reconstruction but not a build claim. The corrected design adds address correctness, thread-local scope, full target identity, buffered records, and run metadata. The next chapter applies these requirements directly to the visible implementation.

3. CHAPTER

IMPLEMENTATION AND CODE AUDIT

3.1. Audit method

The audit compares visible source expressions with the semantics documented by DynamoRIO and the Linux executable model. A finding requires a specific code location, an expected property, an observed deviation, and an effect on the reported results. Severity reflects measurement validity rather than exploitability. A critical finding changes what a central output field means. A high finding can make inclusion or classification dependent on address layout, thread scheduling, or code-cache history. A medium finding weakens robustness, reconciliation, or reproducibility. A low finding affects diagnostics or presentation without changing the central count.

The source is available only as images embedded in the original PDF. Listings 3.1 and 3.2 transcribe the relevant visible expressions. Whitespace and comments are shortened to fit the page, but the operands that drive the findings are retained. Appendix APPENDIX-2 provides clean typeset plates and preserves the unaltered source images in the accompanying source package. No claim is made that these fragments form a complete or currently compilable client.

3.2. Memory operand instrumentation

Listing 3.1 shows the central memory instrumentation pattern. For every source operand that is a memory reference, the client inserts a clean call with three arguments. The first argument is built from `instr_get_app_pc(inst)`. The same pattern is used for destination operands. The second argument is an operand size, and the third distinguishes

a read from a write.

```
if (!in_main)
    return DR_EMIT_DEFAULT;

if (instr_reads_memory(inst)) {
    for (int i = 0; i < instr_num_srcs(inst); i++) {
        if (opnd_is_memory_reference(instr_get_src(inst, i))) {
            dr_insert_clean_call(
                drcontext, bb, inst, (void *)memref_event, false, 3,
                OPND_CREATE_INTPTR(
                    (ptr_int_t)instr_get_app_pc(inst)),
                OPND_CREATE_INT32(
                    drutil_opnd_mem_size_in_bytes(
                        instr_get_src(inst, i), inst)),
                OPND_CREATE_INT32(0));
        }
    }
}
/* The destination loop repeats the pattern with type 1. */
```

Listing 3.1. Transcription of the visible memory instrumentation pattern.

3.2.1. Finding A1: the recorded address is an instruction PC

Severity: critical. The first callback argument is the application program counter. It is not the effective address of the source or destination operand. The original explanation calls this value the address at which the memory operation occurs, which can be read as the data location. The implementation instead identifies the code location of the instruction. Every execution of the same static instruction therefore reports the same value, even when loop iterations access different array elements.

The size field does not repair the missing address. It reports the width of the operand that the decoder can determine for the instruction. The read or write label also remains useful. The existing record can validly be interpreted as “a memory operand of this size was associated with the instruction at this PC.” It cannot be interpreted as “this data address was read or written.”

DynamoRIO’s `drutil_insert_get_mem_addr` helper is designed to calculate the effective address before the application instruction executes [4]. It needs a destination register and a scratch register whose use must not corrupt application state. The `drreg` extension manages such register reservations in staged instrumentation [9]. A corrected

design should record both the instruction PC and calculated effective address.

3.2.2. Finding A2: string and complex memory operations need expansion policy

Severity: medium. The visible loops inspect decoded source and destination operands. Some x86 instructions represent repeated string operations or more complex memory behavior that cannot be treated as a single ordinary operand event if the research question concerns each dynamic access. The official utilities include expansion support for repeated string operations, and the official memory-trace sample documents special handling for string and scatter or gather forms [4, 10]. TraceFlow does not state an expansion policy. Counts should therefore be described as events generated by the chosen instrumentation loop, not complete counts of all micro-level memory transfers.

3.2.3. Finding A3: read and write counts are not disjoint

Severity: medium. The read loop and write loop are independent. A read-modify-write instruction may satisfy both predicates, and a memory operand can appear as both source and destination. The preserved totals are valid as operand-event counters but their sum is not a count of unique memory instructions. Reporting separate event rates is appropriate. Reporting a combined rate is also possible if it is explicitly called *operand events per instruction*.

3.3. Main-function scope

3.3.1. Finding A4: translation-time gating depends on cache history

Severity: high. The visible insertion callback begins with `if (!in_main) return DR_EMIT_DEFAULT`. This condition is evaluated while DynamoRIO constructs or instruments a block. The resulting block can remain in the code cache. If it is first translated before main entry, it may contain no TraceFlow calls when it later executes within the intended scope. If first translated while the flag is true, the inserted calls remain even if the block later executes after main returns.

The defect is not that a condition exists. It is that an execution-state condition controls permanent instrumentation at translation time. The corrected design should instrument relevant sites independently of the current global flag and insert a fast runtime test of thread-local scope. A code-cache flush could force translation after scope changes, but it would be more complex and would still require a clear multi-thread policy.

Figure 3.1 illustrates the problem with two executions of the same block.

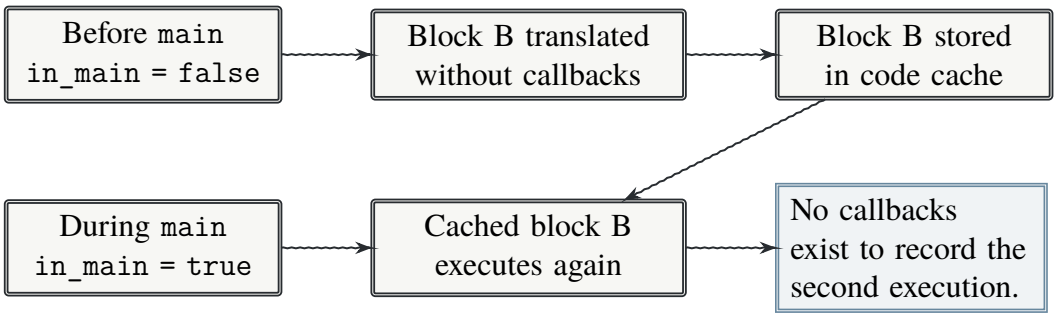

Figure 3.1. A translation-time scope decision can survive into a different execution scope.

3.3.2. Finding A5: process-global scope is not thread-safe

Severity: high. A process-global Boolean cannot represent per-thread entry to main, nested calls, or concurrent execution. A worker thread can execute while the initial thread is inside main and inherit the same process-wide true value. Another callback can change the flag while the first thread is still active. Shared counters and a shared log also require synchronization or per-thread aggregation. The visible fragments do not establish either.

The corrected state should live in DynamoRIO thread-local storage. A depth value is preferable to a Boolean if recursive or nested scope transitions are possible. A policy decision is still needed for threads created from main. The tracer could include every thread that begins while the process is in application scope, or only include threads whose own scope state is explicitly activated. Either policy can be implemented, but it must be documented and tested.

3.4. External-call classification

Listing 3.2 transcribes the visible handler. The callback first checks `in_main`, stores the callee, looks up the containing module, masks the target to 16 bits, and labels offsets in

a fixed interval. The aggregate external-call counter is incremented after the module and range conditions.

```
static void external_call_event(app_pc caller, app_pc callee) {
    if (!in_main) return;

    current_call = callee;
    module_data_t *mod = dr_lookup_module(callee);
    if (mod != NULL) {
        ptr_uint_t offset = (ptr_uint_t)callee & 0xFFFF;
        if (offset >= 0x1000 && offset <= 0x1180) {
            update_call_count(callee);
            const char *func_name = get_plt_name(offset);
            /* formatted output omitted */
        }
        dr_free_module_data(mod);
    }
    stats.ext_calls++;
}
```

Listing 3.2. Transcription of the visible external-call handler.

3.4.1. Finding A6: target suffix is not a module-relative offset

Severity: critical. Masking an address with `0xFFFF` retains only its low 16 bits. A module-relative offset requires subtracting the start of the module that contains the address, after confirming containment. The visible `dr_lookup_module(callee)` call finds a module, but its returned start address is not used in the calculation. As a result, the name mapping can collide across modules and can change when layout or linkage changes.

The fixed range from `0x1000` through `0x1180` appears tailored to the PLT layout of one observed target. That is a useful demonstration fixture but not a generic classifier. A robust tracer should retain the full target address, module path or build identity, and module-relative offset. It can request a symbol through `drsyms` when available and fall back to the numeric module-relative form when symbols are stripped.

3.4.2. Finding A7: aggregate and named totals use different predicates

Severity: high. The code increments `stats.ext_calls` for every callback that passes the initial main-scope test, including targets outside the fixed offset range and targets for which module lookup fails. The eight printed functions are updated only inside the range. The preserved total of 76 therefore must not be described as the sum of the eight displayed

rows. The screenshot shows one occurrence for each named row, leaving 68 handler events that cannot be classified from the retained output.

This mismatch is not necessarily a counter bug. It is a reporting bug if the total is presented without its broader predicate. A corrected summary should contain mutually reconcilable categories such as *named target*, *unnamed target in main module*, *target in another module*, and *unresolved target*. Their sum should equal the all-call total.

3.4.3. Finding A8: shared current target and output require synchronization

Severity: medium. The assignment to `current_call`, counter updates, and writes to a shared output handle are visible without a synchronization policy. Multiple threads can overwrite the shared target and interleave log lines. DynamoRIO clients often avoid this by buffering per thread and merging after execution. If atomic counters are used, their exact guarantees should be stated. If a lock protects reporting, its overhead should be measured.

3.5. Initialization and failure handling

The visible initialization calls `drsym_init()` without checking its return value. It registers exit and basic-block events, obtains the main module, performs a symbol lookup for `main`, frees the module data, and calls `drsym_exit()`. Because the visible symbol lookup occurs before shutdown, immediate shutdown does not invalidate that particular lookup. It would prevent later symbol queries unless the library were initialized again.

3.5.1. Finding A9: required initialization results are not enforced

Severity: medium. Symbol initialization, event registration, extension initialization, and output creation can fail. A required failure should stop the client with an explicit diagnostic. The preserved code prints a message when the main module or symbol is unavailable, but the complete failure path and process status are not visible. A stripped binary may not expose `main` in the expected symbol table. A fallback could accept an explicit module-relative entry offset, use debug information if available, or run in a documented whole-module mode.

3.5.2. Finding A10: entry-point formatting and terminology are imprecise

Severity: low. The visible format string prefixes `%p` with literal `0x`. Common C libraries already render pointers with this prefix, which can produce `0x0x...`. More importantly, an ELF entry point and the start of `main` are different concepts. The code prints `module->entry_point`, then separately resolves `main`. The report should use distinct labels.

3.6. Overhead and observer effects

The prototype inserts a clean call for every relevant operand and another for every application instruction. If the handlers perform formatted output, the analysis can dominate the target's runtime. This does not change the arithmetic of a final software counter, but it can alter thread interleavings, timing-sensitive behavior, cache state, and system interaction. Dynamic instrumentation is always an observer with cost. Sound evaluation measures the cost rather than calling it negligible.

A corrected benchmark should run the same target natively, under DynamoRIO with an empty client, under TraceFlow with counting only, under TraceFlow with buffered recording, and under any verbose diagnostic mode. Each configuration should be repeated after warm-up. Wall-clock time, user and system CPU time, exit status, record count, bytes written, and peak resident memory should be retained. `perf stat` can add hardware and software counter context where permissions and the host configuration allow it [20]. Median and dispersion should be reported rather than a single fastest run.

3.7. Audit summary

Table 3.1. Summary of implementation findings.

ID	Finding	Severity	Consequence
A1	Instruction PC reported as memory address	Critical	Data locations cannot be recovered from the preserved field
A2	Complex memory operations lack an expansion policy	Medium	Counts are instrumentation events, not proven micro-access totals

ID	Finding	Severity	Consequence
A3	Read and write categories overlap	Medium	Their sum is not a unique-instruction count
A4	Main scope gates translation	High	Inclusion can depend on code-cache history
A5	Main scope is process-global	High	Multi-thread attribution and nesting are unreliable
A6	Low 16 bits used as target offset	Critical	Names can collide or change with layout
A7	Total counter is broader than named output	High	The 76-event total cannot be reconciled with eight displayed rows
A8	Shared target, counters, and log lack a visible concurrency policy	Medium	Events and lines may race or interleave
A9	Required initialization results are not fully enforced	Medium	Partial operation may be mistaken for complete tracing
A10	Pointer formatting and entry terminology are imprecise	Low	Diagnostics can be confusing but central totals remain unchanged

The audit changes the interpretation of the prototype without discarding its educational value. Its counters demonstrate how instrumentation callbacks can be inserted and summarized. Its memory "address" field should be renamed *instruction PC*. Its PLT labels should be treated as target-specific demonstrations. Its main scope should be treated as an intention that needs runtime and multi-thread tests.

3.8. Corrective implementation pattern

Listing 3.3 is a schematic design excerpt, not a claim about the preserved executable. It shows the required separation of fields. Resource reservation, error branches, special-instruction expansion, buffer capacity checks, and restoration are abbreviated. A production implementation should follow the complete official sample and extension contracts for the selected DynamoRIO release.

```
/* Analysis stage: identify a memory operand and its static PC. */
opnd_t ref = instr_get_src(inst, operand_index);
```

```
app_pc pc = instr_get_app_pc(inst);

/* Insertion stage: reserve registers through drreg. */
reg_id_t reg_addr, reg_scratch;
reserve_two_registers(drcontext, bb, inst,
                      &reg_addr, &reg_scratch);

/* Compute the dynamic data address before the instruction. */
bool ok = drutil_insert_get_mem_addr(
    drcontext, bb, inst, ref, reg_addr, reg_scratch);
if (ok) {
    /* Runtime code also reads scope from thread-local storage. */
    insert_buffer_write(bb, inst, EVENT_READ, pc,
                        reg_addr, operand_size(ref, inst));
}

restore_reserved_registers(drcontext, bb, inst,
                           reg_addr, reg_scratch);
```

Listing 3.3. Schematic corrected pattern for a memory event. This is a proposal, not preserved code.

For a target record, the corresponding pattern retains the full callee and performs module lookup without truncation. If $base \leq callee < end$, it calculates $callee - base$. Symbol lookup is optional enrichment rather than the sole identity. Unknown targets remain in the raw trace. These rules ensure that a failed name lookup does not silently remove an otherwise valid control-transfer observation.

3.9. Chapter summary

The audit identified two critical semantic errors and several scope, reconciliation, concurrency, and reproducibility weaknesses. None of the findings requires altering the preserved numerical totals. They require narrower labels and prevent unsupported conclusions. Chapter 4 turns the audit into an experimental plan that could verify the corrected system.

4. CHAPTER

EXPERIMENTAL METHOD AND EVIDENCE

4.1. Study design

This work has two methodological layers. The first is a retrospective evidence study. It analyzes the surviving screenshots and code fragments without changing their content. The second is a prospective validation protocol for the corrected design. The protocol specifies the artifacts and tests needed before stronger claims can be made. Keeping these layers separate avoids an improper substitution in which a plausible redesign is presented as if it had produced the old outputs.

The retrospective layer answers descriptive questions. It transcribes exact counters, verifies address subtraction, groups memory-map regions, and checks whether table totals reconcile. It also uses interface documentation to interpret the visible expressions. It cannot estimate run-to-run variation or overhead. The prospective layer defines controlled fixtures with known behavior. It can test field meaning, scope, thread attribution, target identity, error handling, and performance when implemented and executed in a retained environment.

4.2. Evidence classification

Every substantive statement is assigned one of five evidence classes:

1. **Observed:** printed directly in a supplied screenshot or visible in a source image.
2. **Derived:** calculated from observed values with a displayed formula.
3. **Documented:** supported by an official API, ABI, or system reference.

4. **Inferred:** a likely interpretation that is not uniquely established by the retained artifacts.

5. **Proposed:** a design, experiment, or code change that was not part of the preserved run.

An unavailable value is not assigned a plausible substitute. For example, the unknown DynamoRIO version is listed as unavailable rather than guessed from screenshot styling. This policy makes the study less dramatic but more useful. It tells a future maintainer exactly where new evidence would change the conclusion.

Table 4.1. Evidence rules applied to central thesis statements.

Statement	**Class**	**Rule**
The memory run printed 359 reads	Observed	May be quoted exactly with screenshot provenance
The read rate is 15.8 per 100 instructions	Derived	Formula and rounding must be shown
`instr_get_app_pc` returns an application PC	Documented	Cite the framework interface and distinguish it from effective address
The target range represents the PLT of the test binary	Inferred	State that the fixed addresses suggest this purpose but do not prove it generically
Thread-local buffering will reduce per-event output work	Proposed	Describe as a design expectation pending measurement

4.3. Preserved experiment record

The available experiment record contains two output modes. The external-call mode reports the main module base, the main address, a relative main offset, individual lines for selected call targets, and a total callback count. The memory mode reports memory-map regions, individual instrumentation output, and final counts for read operands, write operands, and executed application instructions. The thesis PDF does not preserve the complete invocation command or environment.

The retrospective procedure is therefore:

1. Preserve the supplied PDF unchanged and record its file identity in the project manifest.

2. Extract embedded images without resampling where possible.

3. Transcribe numerical values twice and compare the transcriptions.
4. Recalculate only quantities whose inputs are present, such as module-relative offset and event rates.
5. Do not infer missing event rows from an aggregate total.
6. Link every reconstructed evidence plate to its original PDF page in Appendix APPENDIX-2 and retain the unaltered image in the source archive.

This procedure protects the source evidence while allowing a corrected interpretation. The numerical charts in Chapter 5 are exact redrawings of transcribed values. Their geometry is not hand-sketched because sketch distortion would be inappropriate for quantitative comparison.

4.4. Prospective validation environment

A reproducible rerun should use a dedicated Linux environment rather than the analyst's daily workstation. The environment can be a virtual machine or an isolated physical host. It should have no access to personal files or production credentials. Benign fixtures are sufficient for validating an instrumentation engine. If later research introduces malicious samples, it requires additional containment, network control, authorization, and incident-handling procedures consistent with institutional policy and NIST guidance [8].

Before each run, the harness should record:

- operating-system image identifier, distribution release, kernel, architecture, and CPU model;
- DynamoRIO release and build identity;
- TraceFlow source commit, compiler, compile flags, and linked extension versions;
- target source commit where available, compiler, flags, binary format, PIE status, and SHA-256 hash;

- exact command line, working directory, relevant environment variables, and standard-input fixture;
- start and end time, exit status, standard output, standard error, raw trace path, and hashes of every result file.

The invocation should be stored as data and executed by the harness. A representative shape is `drrun -c <client> <client-options> -- <target> <target-options>`. This is not asserted to be the original command. It is a template to be filled with paths and options from the retained build. Quoting must be handled by the harness so that the metadata exactly matches the executed argument vector.

4.5. Validation fixtures

Small deterministic programs provide stronger semantic tests than a complex application whose correct event sequence is unknown. Each fixture should emit a checksum so that instrumentation-induced functional changes are visible. Source, binary, disassembly, and expected trace properties should be retained together.

4.5.1. Effective-address fixture

The first fixture allocates a fixed array, performs reads and writes at known indices, and prints the array base and final checksum. The test harness compares TraceFlow effective addresses against the known array interval and a debugger-assisted reference. The same static load instruction should be observed with different effective addresses across loop iterations. The instruction PC should remain constant for that instruction. This fixture directly detects the A1 defect.

4.5.2. Read-modify-write fixture

The second fixture applies an operation such as increment to a memory location. Disassembly confirms an instruction form that reads and writes the same operand. The expected result is one read-operand event and one write-operand event associated with the same instruction execution and effective address. A unique-instruction summary should

count the instruction once. This verifies that raw event categories and derived summaries remain distinct.

4.5.3. Code-cache scope fixture

The third fixture places a helper function in a constructor that runs before `main`, then calls the same helper during `main`. This encourages the same code to be translated before and during the intended scope. A correct runtime predicate records the in-scope execution regardless of the earlier translation. Running a reversed or cache-flushed variant helps confirm that results do not depend on translation order. This fixture directly tests A4.

4.5.4. Thread fixture

The fourth fixture creates two worker threads with distinct memory ranges and event counts. Each thread writes a deterministic marker and joins cleanly. The trace must associate every event with the correct DynamoRIO thread identifier and buffer. Repeated runs should produce the same per-thread totals even if interleaving changes. Log records must remain parseable with no partial or interwoven lines.

4.5.5. Target-resolution fixture

The fifth fixture calls known imported functions from PIE and non-PIE builds. It also loads a small shared object built by the project. The expected report contains full target addresses, containing-module identities, and module-relative offsets. Cross-run comparison uses module identity plus offset and should remain stable when ASLR changes absolute addresses. A separate stripped build tests numeric fallback when symbols are unavailable.

4.5.6. Failure fixtures

Failure behavior is tested by using an unwritable output path, a truncated configuration, a binary without an accessible `main` symbol, a full output buffer during process termination, and deliberate extension-initialization failure where the test interface permits it. The client

must emit a clear error and avoid presenting a partial trace as complete. These tests are essential because silent data loss can invalidate a large run while leaving its summary plausible.

Table 4.2. Prospective semantic validation matrix.

Fixture	**Property under test**	**Acceptance condition**
Indexed array	Effective address	Dynamic addresses fall in expected elements while the responsible PC remains stable
Memory increment	RMW classification	Paired read and write events share PC and data address; unique count remains one
Constructor plus main helper	Runtime scope	In-main execution is recorded independent of first translation time
Two workers	Thread attribution	Records and totals are correct per thread with no corrupted output
PIE, non-PIE, shared object	Target identity	Module plus offset is stable across randomized load addresses
Stripped executable	Symbol fallback	Numeric records remain complete when names are unavailable
Output and initialization failures	Completeness signaling	Run is explicitly marked failed or incomplete

4.6. Quantitative measures

For a run with I executed application instructions, R read-operand events, and W write-operand events, the normalized event rates are:

$$Rate_{read} = \frac{R}{I} \times 100 \tag{4.1}$$

$$Rate_{write} = \frac{W}{I} \times 100 \tag{4.2}$$

$$Rate_{operand} = \frac{R + W}{I} \times 100 \tag{4.3}$$

The unit for each value is events per 100 executed application instructions. The combined rate can exceed 100 for instruction sets or workloads with multiple memory operands. It is not a probability and not a percentage of unique instructions.

For timing, let T_n be native runtime, T_d be runtime under DynamoRIO with an empty client, and T_t be TraceFlow runtime. Two useful ratios are:

$$Overhead_{total} = \frac{T_t}{T_n} \tag{4.4}$$

$$Overhead_{client} = \frac{T_t}{T_d} \tag{4.5}$$

The first includes the framework and client. The second isolates the client relative to the framework baseline more closely. Neither value can be calculated from the preserved evidence. A future experiment should report the median and an interval such as the interquartile range from a predeclared number of repetitions. Statistical tests are unnecessary when the objective is a deterministic semantic property. For performance comparisons, effect size and variability are more informative than an isolated significance threshold [21].

4.7. Reconciliation checks

Raw records and summaries should satisfy invariants after every complete run. The number of read records must equal the reported read total. The same rule applies to writes, instructions, and calls. Named, unnamed, unresolved, and filtered call categories must sum to the all-call total. Per-thread totals must sum to the process total. Sequence numbers must be unique within their documented domain. Each module-relative offset must lie within the module interval recorded for the event time.

The harness should fail a run when an invariant is violated. It should not silently adjust a summary to match expectations. A discrepancy is evidence about the tracer. The 76 versus eight relationship in the preserved external-call output demonstrates why reconciliation categories matter. The aggregate is not wrong merely because the visible named rows sum to eight. The report is incomplete because it does not show where the other 68 events went.

4.8. Threats to validity

Construct validity concerns whether a field measures its stated concept. The program-counter issue is a construct-validity failure. Validation fixtures with known addresses directly address it. **Internal validity** concerns whether observed changes result from the instrumentation rather than environment drift, caching, or compiler changes. Fixed images, hashes, and controlled variants reduce this risk. **External validity** concerns generalization beyond the small fixtures. Passing the fixtures establishes semantic correctness for tested instruction and binary forms, not complete coverage of all Linux software. Larger open-source workloads can be added only after the core semantics pass.

Conclusion validity concerns the strength of numerical claims. One preserved run supports a description of that run, not a distribution. No confidence interval or malware-detection rate can be computed. **Reproducibility validity** concerns whether another researcher can reconstruct the setup. The current evidence is insufficient, which is why the prospective manifest includes versions, hashes, commands, and raw records.

4.9. Chapter summary

The method preserves a strict boundary between retrospective evidence and prospective validation. It defines deterministic fixtures for each important code-audit finding, normalizes counters without changing their meaning, and requires reconciliation between raw events and summaries. Chapter 5 applies the retrospective portion to the exact values that survived.

5. CHAPTER

RESULTS

5.1. Reporting boundary

This chapter reports only values retained in the supplied thesis PDF and arithmetic derived from those values. It does not present a new execution of TraceFlow. Exact quantitative charts use straight axes and undistorted geometry. Conceptual figures elsewhere use the deterministic sketch style. This visual distinction prevents a hand-drawn design language from implying uncertainty in exact counts or, conversely, from making a conceptual diagram look like measured data.

5.2. Main-module and symbol result

The external-call screenshot reports a main-module base of `0x000079a333400000` and a main-function address of `0x000079a333401269`. Subtracting the module base yields `0x1269`. Equation 5.1 confirms the printed offset.

$$\texttt{0x000079a333401269} - \texttt{0x000079a333400000} = \texttt{0x1269} \tag{5.1}$$

This is a valid module-relative calculation for the two displayed addresses. It shows that symbol lookup and base addition produced an internally consistent `main` location in that run. It does not validate the separate low-bit calculation used for external-call labels.

5.3. External-call result

The preserved output displays eight named PLT targets. Each displayed target has a count

of one. The handler's final total is 76. Table 5.1 reproduces the named rows.

Table 5.1. Named external-call targets retained in the preserved output.

Displayed target	**Displayed offset**	**Displayed count**
write	`0x10e0`	1
getpid	`0x10f0`	1
strlen	`0x1100`	1
printf	`0x1120`	1
lseek	`0x1130`	1
close	`0x1140`	1
open	`0x1150`	1
fstat	`0x1170`	1
Named rows shown		8
Handler events reported		76

The difference is 68 events. The source audit explains why the values need not match: `stats.ext_calls` is incremented outside the visible module-range and name filters. The retained output does not provide identities or categories for those 68 events. They are therefore labeled *not represented among the retained named rows*, not *unknown functions* and not *malicious calls*.

Figure 5.1 shows the reconciliation using exact integer values. The shaded segment does not infer a target class. It only represents the arithmetic remainder between the reported total and the sum of displayed named counts.

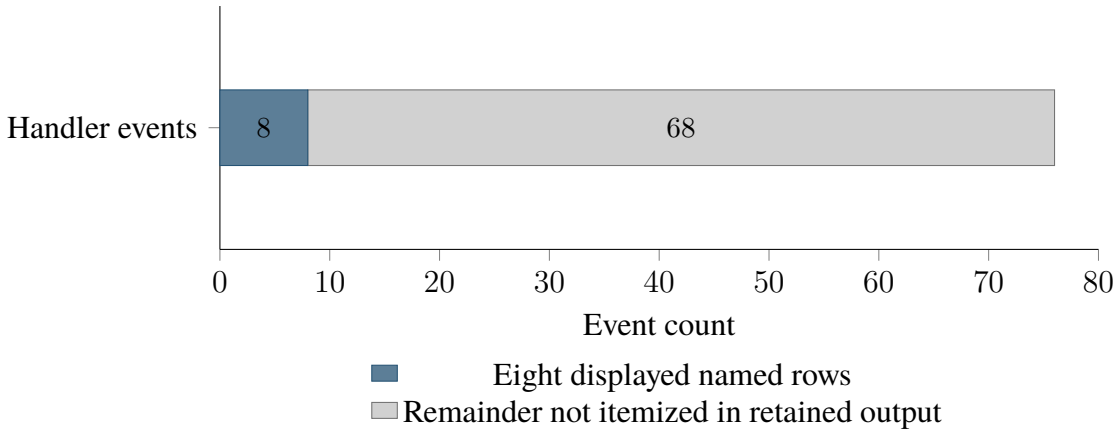

Figure 5.1. Exact reconciliation of the 76 external-call handler events.

The target list shows that the demonstration program used common file, process, and

string-related library interfaces. It does not establish frequency beyond one displayed instance each, because the screenshot may represent filtered output and the aggregate predicate is broader. No sequence, caller identity, return value, argument value, or error code is retained. A behavior narrative such as “the program opened a file, inspected it, and wrote data” would be plausible but not uniquely supported. Many benign programs call the same interfaces.

5.4. Memory and instruction counts

The memory-mode output reports 359 read-operand events, 353 write-operand events, and 2,274 executed application instructions. Figure 5.2 plots these counts on a common linear scale.

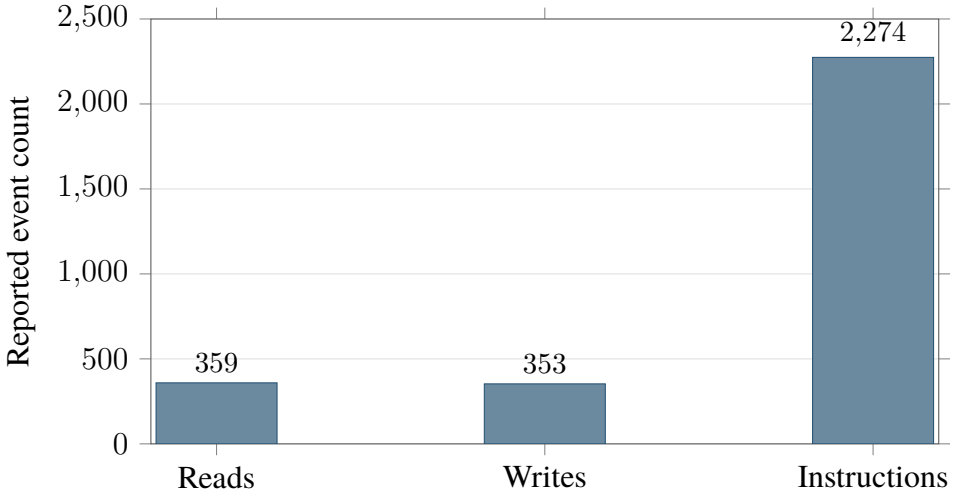

Figure 5.2. Exact counters in the preserved memory-mode summary.

The read and write counts are close, with six more read events than write events. That difference is descriptive only. It does not establish a read-dominant workload because there is one run, no variance estimate, and no event-level trace. The instruction count is the denominator for normalized operand-event rates. It is not an independent hardware retired-instruction measurement.

Applying Equations 4.1 through 4.3 gives:

$$Rate_{read} = \frac{359}{2274} \times 100 = 15.787\ldots \approx 15.8, \tag{5.2}$$

$$Rate_{write} = \frac{353}{2274} \times 100 = 15.523\ldots \approx 15.5, \tag{5.3}$$

$$Rate_{operand} = \frac{359 + 353}{2274} \times 100 = 31.310\ldots \approx 31.3. \tag{5.4}$$

Figure 5.3 presents the rounded values. The combined bar counts operand events, including any overlap caused by read-modify-write instructions.

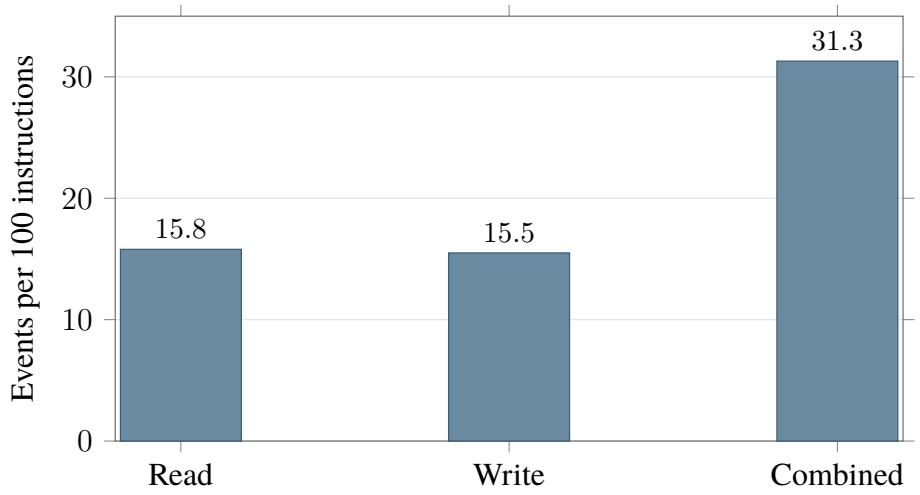

Figure 5.3. Derived operand-event rates from the preserved memory run.

The rates support a compact comparison between future runs if the same instrumentation semantics are maintained. They should not be compared with hardware load/store counters without accounting for different event definitions. The prototype loops over decoder operands and inserts software callbacks. A hardware counter may count retired loads, stores, cache events, or micro-operations under architecture-specific rules.

5.5. Memory-map result

The preserved memory-map screenshot lists mappings for the test process. Contiguous rows have been grouped by displayed pathname or label, and their displayed interval lengths have been summed. Table 5.2 reports the resulting sizes. This is a descriptive view of the captured map, not a complete measure of resident memory. A mapping's virtual size does not say how many pages are resident or private.

Table 5.2. Grouped virtual mapping sizes transcribed from the preserved screenshot.

Displayed mapping group	**Grouped size (KiB)**
libc	2,068
ld-linux-x86-64.so.2	232
stack	132
anonymous mappings	72
mem_test	20
vvar	16
vdso	8
vsyscall	4
Total of displayed groups	2,552

Figure 5.4 uses a logarithmic horizontal axis because the largest displayed group is more than 500 times the smallest. Every bar retains its exact label. The logarithmic scale is stated in the axis title so that visual length is not mistaken for a linear proportion.

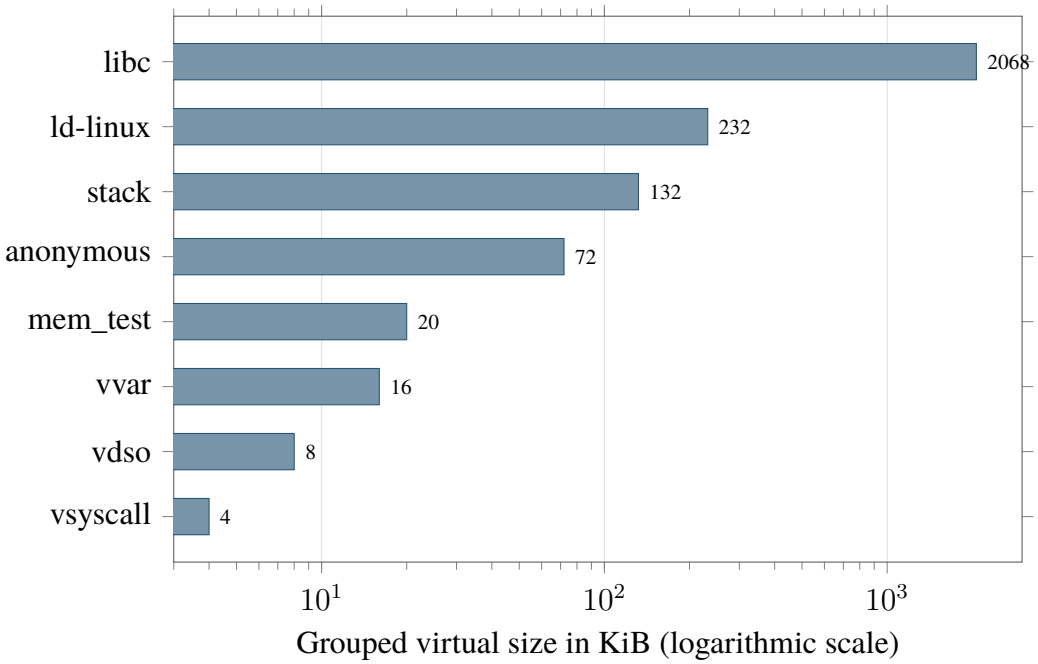

Figure 5.4. Exact grouped mapping sizes from the preserved map.

The library mapping dominates the displayed virtual intervals, followed by the dynamic loader and stack. This is normal for a small dynamically linked program and does not indicate that the target spent most of its time in libc. Mapping size, access count, and execution time are separate quantities. The screenshot contains permissions, but the

grouped chart does not infer writable-executable regions because the exact row-level permission analysis was not retained as structured data.

5.6. Direct answers to the research questions

For the first research question, the evidence directly supports the displayed addresses, 76 external-call handler events, eight named rows with count one, 359 read-operand events, 353 write-operand events, 2,274 executed application instructions, and the transcribed mappings. It supports the derived rates and the `main` offset. It does not support call-sequence reconstruction, data-address reconstruction, repeated-run variability, overhead, or malware-detection accuracy.

For the second question, the visible code does not implement the advertised effective data address. Its scope and target labeling also lack the invariants required for general use. For the third question, the documented correction is a staged, thread-local, buffered design with explicit effective-address calculation and module-relative target resolution. For the fourth question, the numbers describe one bounded demonstration. They show near-equal read and write operand-event counts and a larger instruction-event denominator. They do not show that the program is compute-intensive, memory-intensive, benign, or malicious.

5.7. Chapter summary

The preserved results are internally useful when their predicates are stated accurately. Arithmetic confirms the main offset and the normalized memory-event rates. The call total does not reconcile with the named list because the source uses different filters. The next chapter discusses what these results mean for engineering, validity, and defensive analysis.

6. CHAPTER

DISCUSSION

6.1. What the prototype demonstrates

TraceFlow demonstrates the basic mechanics of a DynamoRIO client. It registers callbacks, obtains module information, resolves a main symbol in the preserved initialization path, inspects decoded instructions, inserts clean calls, and aggregates runtime events. These are meaningful implementation steps. The output proves that a demonstration binary executed under an instrumented environment and that the client emitted counters and target labels.

The prototype also demonstrates why instrumentation research needs a precise data dictionary. The same hexadecimal field can be interpreted as an instruction PC, effective address, absolute call target, module-relative offset, or PLT entry. These meanings are not interchangeable. A small labeling error can propagate into figures and conclusions that look technically detailed but answer the wrong question. The most valuable outcome of the audit is therefore not a larger feature list. It is a measurement model in which every field has one declared origin and unit.

The preserved memory counts remain useful after relabeling. They describe how many callbacks the client generated under its decoder and scope policy. The instruction addresses could help find hot access sites if full records were available. The missing effective addresses prevent data-region analysis, but they do not erase the instruction-attribution purpose. Similarly, the fixed PLT map can teach how a particular target binary is laid out. It should be presented as a target-specific map rather than a

reusable symbol resolver.

6.2. Meaning of the call result

The external-call total of 76 and the eight named rows measure different sets. This is a reporting-design lesson. A reader naturally expects a table under a total to partition that total. The source shows that the total includes handler events outside the naming interval. The difference of 68 should remain visible as a reconciliation category. Hiding it would give a cleaner chart but a weaker thesis.

The displayed names are common C library interfaces. Their presence alone has little security specificity. `open`, `fstat`, `lseek`, `write`, and `close` may occur in an ordinary utility, a test program, an installer, or malicious software. A behavioral claim would need arguments, return values, file identities, calling context, ordering, and perhaps system-call confirmation. Function names are features, not verdicts.

PLT interception also has coverage limits. A compiler may inline code, bind a function directly, use an alternative interface, or issue a system call without a libc wrapper. Lazy binding can cause early transfers through resolver paths. Indirect calls may target a resolved address in a shared object. A general call tracer should record control transfers first and enrich them with symbols later. Making a name a prerequisite for retaining the event reverses that priority.

6.3. Meaning of the memory result

The read rate of 15.8 and write rate of 15.5 per 100 instructions are normalized software-event rates. They allow scale-independent comparison only when the instrumentation policy is unchanged. A later version that expands string operations or deduplicates read-modify-write instructions would create a different metric. Schema versioning and manifest metadata are therefore necessary for longitudinal use.

The close values do not establish a balanced memory workload. The prototype counts operands visible to its loops, not bytes transferred, cache misses, unique addresses, or time waiting on memory. A 64-byte vector access and a 1-byte scalar access can each

contribute one event. Operand size was passed to the callback, but no retained aggregate by size is available. The event rates are best treated as a smoke-test signature for this one target and instrumentation version.

An effective-address implementation would allow richer but still bounded analyses. Records could be grouped by stack, heap, main module, shared library, anonymous mapping, or memory-mapped file using the map intervals valid at event time. Sequentiality and reuse distance could be studied. Accesses to executable and writable regions could be flagged for review. These analyses require dynamic map updates, because modules and mappings can appear or disappear while the program runs. A single startup map is not enough for a long-lived process.

6.4. Scope semantics

The phrase “inside main” needs an operational definition. One option is lexical module scope: instrument instructions whose addresses lie in the main executable. Another is dynamic call scope: include all instructions executed after entry to `main` and before its matching return, including shared-library callees. A third option is thread-rooted scope: include the initial thread and explicitly propagated worker-thread activity. The preserved global flag appears closest to dynamic call scope, but its translation-time use undermines that intent.

For behavior analysis, dynamic call scope is often more informative because external work performed on behalf of main remains visible. It also includes loader or library behavior that may dominate output. The tracer should therefore record both scope state and module identity. An offline query can then select “events during main in any module” or “events executed from the main module” without rerunning the target.

Signals and nonlocal exits complicate a simple entry-return pair. A process may call `exit`, raise a fatal signal, unwind, or never return from main. The runtime should flush per-thread buffers from supported exit callbacks and mark whether the scope closed normally. Depth state must be reset during thread teardown. These details matter because an incomplete final buffer can bias the last part of a trace without an obvious error.

6.5. Performance implications

Per-event clean calls are easy to understand and appropriate for proving that a callback works. They are not a good default for large traces. Each transition must preserve enough application state for the analysis routine, and formatted output adds conversion, locking, and I/O. The observer may slow the program enough to alter scheduling and timeout behavior. A timing-sensitive target can follow a different path under instrumentation.

The proposed per-thread buffer reduces but does not remove overhead. Inline code writes a fixed-size record and advances a pointer. A clean call occurs only on buffer flush or exceptional handling. The record format should be compact enough to limit bandwidth but explicit enough to remain parseable. Binary records can be converted to text offline. If loss is permitted for performance, the tracer must count dropped events and mark the run. Silent sampling or loss is unacceptable.

Framework baseline measurement separates client cost from the cost of running under DynamoRIO. Comparing TraceFlow only with native execution would attribute both costs to the client. Comparing a verbose logger only with a buffered logger would hide the framework baseline. The five-configuration plan in Chapter 3 provides an interpretable ladder. The test should also confirm functional output and checksum equality, because a fast run that changed program behavior is not a valid performance result.

6.6. Comparison with established tools

DynamoRIO's official `memtrace_simple` and `drcachesim` infrastructure already demonstrate address-correct, buffered tracing patterns [10, 16]. TraceFlow should reuse those patterns instead of creating a competing low-level buffering mechanism without need. Its distinct value can be the narrower educational report that connects main scope, module-relative call targets, and memory categories in one auditable schema.

Dr. Memory shows that a practical memory checker needs substantially more semantics than a stream of operands, including allocation state, shadow metadata, and detailed error reporting [22]. Valgrind similarly illustrates the capability and cost of an intermediate-representation-based checking framework [13]. TraceFlow is not a substitute

for those tools. It is a smaller observability client.

Pin exposes comparable opportunities for routine and instruction instrumentation [12]. The design principles found in this audit are not DynamoRIO-specific: translation-time and runtime state must be distinguished, application state must be preserved, addresses need module context, and output counters need reconcilable predicates. DynamoRIO-specific extensions provide practical mechanisms for satisfying those principles.

6.7. Defensive security relevance

A corrected TraceFlow can support defensive research in three roles. First, it can produce a compact execution inventory for a benign or authorized sample. Second, it can identify code locations associated with selected memory and control-flow events for later reverse engineering. Third, it can export structured records to a separate analysis pipeline. None of these roles requires the tracer itself to label a program as malware.

A detector would require another layer. It would define features, training or rule-construction data, ground truth, a decision threshold, and evaluation against separate samples. False positives and false negatives would need reporting. The environment would need controls for evasion and instrumentation detection. The current study intentionally stops before that layer. Calling the prototype "AI" would be inaccurate because no learned model, training procedure, inference path, or evaluated classifier is present in the evidence.

Memory access can contribute to security findings when combined with stronger context. For example, writes to executable anonymous memory followed by control transfer may deserve review. Access to another process through `/proc/<pid>/mem` is relevant to a documented ATT&CK technique [19]. A generic read or write operand in one process does not establish either behavior. System calls, mapping permissions, target process identity, and temporal sequence are required.

6.8. Research validity

The greatest threat to this thesis is artifact incompleteness. The revised text mitigates it through explicit evidence classes and by avoiding rerun language. It cannot recover a lost repository or raw trace. A future repository release should include the source, build configuration, fixture programs, expected outputs, manifest schema, and scripts for running and validating the tests. Release tags and hashes should connect the manuscript to a specific state.

The second threat is implementation drift. Official APIs evolve, and a design excerpt based on current documentation may need adjustment for another release. The source should pin a supported DynamoRIO version and test against it in continuous integration. Documentation URLs in the bibliography provide conceptual grounding, but a build manifest must still identify the exact installed headers and binaries.

The third threat is target representativeness. Small fixtures are intentionally narrow. They can prove that a field is calculated correctly for specific instruction forms. They cannot prove coverage of every x86-64 instruction or every ELF linking pattern. Validation should grow from unit fixtures to open-source workloads, then to authorized security corpora if that becomes the research objective. Each stage should add claims only after its evidence exists.

6.9. Engineering priorities

Table 6.1 orders the recommended work. Priority is based on how directly an issue changes measurement meaning and how much later work depends on it.

Table 6.1. Recommended engineering priorities for a corrected TraceFlow.

Order	Change	Reason
1	Separate instruction PC and effective address	Restores the central meaning of memory records
2	Replace global scope with runtime TLS state	Removes cache-history and multi-thread ambiguity
3	Retain full targets and module-relative identities	Makes call records stable under ASLR and across modules
4	Add per-thread buffers and reconciliation counters	Reduces observer cost and makes data loss visible
5	Add manifests, hashes, fixtures, and CI validation	Turns demonstrations into reproducible experiments
6	Measure native, framework, and client overhead	Quantifies observer effects after semantics are correct
7	Add richer behavioral analyses	Builds features on validated records rather than unstable fields

The ordering is important. Performance optimization before address correctness can make an invalid measurement faster. A polished security dashboard before raw-record reconciliation can make missing events harder to notice. The first release criterion should be semantic acceptance tests, followed by completeness and performance.

6.10. Ethics and safe use

The instrumentation techniques in this thesis are dual-use, but the described workflow is defensive and laboratory-safe. It does not provide persistence, evasion, unauthorized access, or destructive behavior. Test programs should be owned by the researcher or distributed with permission. System-wide tracing and analysis of third-party software should follow applicable licenses, institutional policy, and law. Malicious samples, if ever introduced, should be handled only in an approved isolated environment by authorized personnel.

Privacy also matters. Raw traces may contain addresses, paths, arguments, or data-derived values. A published dataset should avoid personal file paths and secrets. Manifests should include enough system information for reproducibility without exposing credentials or unnecessary personal identifiers. Access control and retention rules should be defined

before collection, not after a trace has been shared.

6.11. Chapter summary

TraceFlow is best understood as an instructional DBI prototype whose preserved outputs support bounded runtime observations. The code audit narrows several field meanings and exposes a clear engineering path. Address correctness, runtime thread-local scope, target identity, and reconciled buffering are prerequisites for further claims. The final chapter states the conclusions and concrete recommendations.

7. CHAPTER

CONCLUSION AND RECOMMENDATIONS

7.1. Conclusion

This thesis evaluated TraceFlow as a DynamoRIO based runtime-analysis prototype for Linux binaries. The evaluation used the supplied thesis PDF, embedded source screenshots, and preserved terminal output. It did not claim a new execution because the complete source tree, target binaries, build record, exact command, environment versions, and raw trace were unavailable.

The preserved evidence supports several exact results. In the external-call run, the main module was displayed at `0x000079a333400000`, `main` at `0x000079a333401269`, and their difference at `0x1269`. The handler reported 76 events. Eight PLT target names were shown with one count each. The other 68 events cannot be categorized from the retained output. In the memory run, the client reported 2,274 executed application instructions, 359 read-operand events, and 353 write-operand events. These values correspond to 15.8 reads, 15.5 writes, and 31.3 combined operand events per 100 instructions.

The code audit changed the meaning assigned to important fields. The memory callback receives `instr_get_app_pc`, so the displayed address is the location of the instruction rather than the effective address of the accessed data. The instrumentation callback filters on a process-global `in_main` value at translation time, which makes coverage sensitive to code-cache history and unsuitable for reliable multi-thread attribution. The external-call handler masks the callee to its low 16 bits instead of calculating a verified module-relative offset. Its aggregate counter is also broader than its name-reporting filter.

These findings do not make the prototype worthless. They define what it is: a compact

DBI demonstration that counts callbacks and associates them with instruction sites and a target-specific PLT map. They also define what it is not. The preserved system is not an evaluated malware detector, an artificial-intelligence classifier, a validated effective-address tracer, or a measured low-overhead production tool.

The corrected design proposed in this thesis uses DynamoRIO's documented extension model. It calculates effective addresses with `drutil`, reserves application-safe registers with `drreg`, manages staged events and thread-local state with `drmgr`, retains full targets plus verified module-relative offsets, and writes versioned records through per-thread buffers. A run manifest binds those records to software versions, commands, hashes, and target identity. Deterministic fixtures verify semantics before larger workloads are introduced.

7.2. Recommendations

The following recommendations should be completed in order:

1. Recreate the client in a version-controlled repository and preserve the original screenshot-based revision as a historical reference.

2. Implement separate `instruction_pc` and `effective_address` fields. Use official effective-address helpers and special-instruction handling.

3. Replace process-global main scope with an explicit runtime policy stored per thread. Test the policy with constructor, re-entry, worker-thread, signal, and abnormal-exit fixtures.

4. Remove low-bit target classification. Store the full address, containing module identity, module interval, and verified relative offset. Add a symbol name only when resolution succeeds.

5. Define a versioned binary or structured record schema and derive every summary from retained raw records. Include reconciliation categories for unnamed and unresolved calls.

6. Buffer events per thread, count dropped records, and make flush failure visible. Keep verbose per-event printing as an optional diagnostic mode.

7. Create a run manifest containing the OS, kernel, DynamoRIO release, client commit, compiler flags, command, environment, target hash, result hashes, exit status, and completeness status.

8. Run semantic acceptance fixtures before performance tests. Then compare native, empty-framework, counting, buffered, and verbose configurations with repeated measurements.

9. If the research later addresses malware detection, define a separate approved study with a contained environment, representative labeled corpus, feature definition, decision rule, and false-positive and false-negative evaluation.

7.3. Final statement

Reliable runtime analysis begins with a simple discipline: every output field must correspond to the event it claims to describe. TraceFlow provides a useful foundation for learning that discipline. With address-correct records, thread-aware runtime scope, stable module identity, and reproducible experiments, it can become a defensible analysis tool. Until those changes are implemented and rerun, the present numerical results should remain attached to the preserved prototype and its stated limitations.

REFERENCES

1. Bruening, D., Garnett, T., and Amarasinghe, S., 2003. An Infrastructure for Adaptive Dynamic Optimization, *International Symposium on Code Generation and Optimization*, 265–275, doi: `https://doi.org/10.1109/CGO.2003.1191551`.

2. Bruening, D., Zhao, Q., and Amarasinghe, S., 2012. Transparent Dynamic Instrumentation, *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, 133–144, doi: `https://doi.org/10.1145/2151024.2151043`.

3. DynamoRIO Project, 2025. DynamoRIO Overview, DynamoRIO Project, 11.3.0 edition, version 11.3.0, release 11.3.0-1.

4. DynamoRIO Project, 2025. Utility Routines for Auxiliary Library Use, DynamoRIO Project, 11.3.0 edition, version 11.3.0, release 11.3.0-1.

5. DynamoRIO Project, 2025. DynamoRIO Extension Event Manager, DynamoRIO Project, 11.3.0 edition, version 11.3.0, release 11.3.0-1.

6. Tool Interface Standards Committee, 1995. Executable and Linking Format Specification, Tool Interface Standards Committee, 1.2 edition.

7. Matz, M., Hubicka, J., Jaeger, A., and Mitchell, M., 2013. System V Application Binary Interface: AMD64 Architecture Processor Supplement, Linux Foundation, 0.99.6 edition.

8. Souppaya, M. and Scarfone, K., 2013. Guide to Malware Incident Prevention and Handling for Desktops and Laptops, National Institute of Standards and Technology.

9. DynamoRIO Project, 2025. DynamoRIO Register Management, DynamoRIO Project, 11.3.0 edition, version 11.3.0, release 11.3.0-1.

10. DynamoRIO Project, 2025. DynamoRIO Sample Use Cases, DynamoRIO Project, 11.3.0 edition, version 11.3.0, release 11.3.0-1.

11. Bruening, D., 2004. Efficient, Transparent, and Comprehensive Runtime Code Manipulation, Doktora Tezi, Massachusetts Institute of Technology.

12. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., and Hazelwood, K., 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 190–200, doi: `https://doi.org/10.1145/1065010.1065034`.

13. Nethercote, N. and Seward, J., 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation, **ACM SIGPLAN Notices**, **42**(6):89–100, doi: `https://doi.org/10.1145/1250734.1250746`.

14. Bellard, F., 2005. QEMU, a Fast and Portable Dynamic Translator, *USENIX Annual Technical Conference, FREENIX Track*, 41–46.

15. Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D., 2012. AddressSanitizer: A Fast Address Sanity Checker, *USENIX Annual Technical Conference*, 309–318.

16. DynamoRIO Project, 2025. DynamoRIO Cache Simulator and Memory Tracing Infrastructure, DynamoRIO Project, 11.3.0 edition, version 11.3.0, release 11.3.0-1.

17. Intel Corporation, 2025. Intel 64 and IA-32 Architectures Software Developer's Manual, Intel Corporation.

18. Colomar, A. and man-pages contributors, L., 2025. proc_pid_maps(5): Mapped Memory Regions, Linux man-pages project, 6.15 edition, linux man-pages release 6.15.

19. MITRE ATT&CK, 2025. Technique T1055.009: Process Injection, Proc Memory, The MITRE Corporation, enterprise ATT&CK STIX data snapshot dated 28 October 2025; technique created 14 January 2020.

20. Linux kernel contributors, 2025. perf-stat(1): Run a Command and Gather Performance Counter Statistics, Linux perf project, 6.15 edition, linux kernel release 6.15.

21. Cohen, J., 1994. The Earth Is Round ($p < .05$), **American Psychologist**, **49**(12):997–1003.

22. Bruening, D. and Zhao, Q., 2011. Practical Memory Checking with Dr. Memory, *International Symposium on Code Generation and Optimization*, 213–223, doi: `https://doi.org/10.1109/CGO.2011.5764689`.

23. Erciyes University Department of Computer Engineering, 2025. Undergraduate Thesis Writing Guide and LaTeX Format, Erciyes University, official guide linked from the department announcement updated 31 October 2025.

24. Erciyes University Department of Computer Engineering, 2025. Design Project and Undergraduate Thesis Interim Reports Announcement, Erciyes University, official department announcement updated 31 October 2025.

APPENDIX-1

CODE AUDIT RECORD AND CORRECTED DESIGN EXCERPTS

1.1. Status of the source material

The original project source was supplied as screenshots within the thesis PDF. The excerpts in this appendix are either direct transcriptions of visible lines or schematic corrections. They are not presented as a complete buildable repository. A later implementation should copy the corrected ideas into a new version-controlled client and validate them against the exact DynamoRIO release selected by that project.

1.2. Visible initialization excerpt

Listing APPENDIX-1.1 transcribes the visible initialization logic with long output strings shortened. It shows that symbol lookup occurs during client initialization. The return from `drsym_init` is not checked in the screenshot.

```
DR_EXPORT void dr_client_main(client_id_t id, int argc,
                              const char *argv[]) {
    drsym_init();
    dr_register_exit_event(event_exit);
    dr_register_bb_event(event_basic_block);

    module_data_t *module = dr_get_main_module();
    if (module != NULL) {
        printf("Module entry point: 0x%p\n",
               module->entry_point);

        size_t offset;
        drsym_error_t sym_res = drsym_lookup_symbol(
            module->full_path, "main", &offset,
            DRSYM_DEMANGLE);

        if (sym_res == DRSYM_SUCCESS) {
            app_pc main_addr = module->start + offset;
            printf("Found main at offset 0x%x (address %p)\n",
```

```
                    (unsigned int)offset, main_addr);
        }
        dr_free_module_data(module);
    }
    drsym_exit();
}
```

Listing APPENDIX-1.1. Transcription of the visible initialization and main-symbol lookup.

Important review points are the unchecked library initialization, the distinction between ELF entry point and `main`, the pointer-format prefix, the lack of a visible stripped-binary policy, and immediate symbol-library shutdown after the initial lookup. The module-data object is released in the visible success path, which is appropriate for the copied data returned by the API.

1.3. Proposed event types

Listing APPENDIX-1.2 is a schematic record definition. Fixed-width integer encoding, byte order, packing, and serialization must be specified in the schema document. Directly dumping a compiler-dependent C structure is not a portable file format.

```
typedef enum {
    TRACE_INSTRUCTION = 1,
    TRACE_MEMORY_READ = 2,
    TRACE_MEMORY_WRITE = 3,
    TRACE_CALL = 4,
    TRACE_RETURN = 5,
    TRACE_MODULE = 6
} trace_event_type_t;

typedef struct {
    uint32_t schema_version;
    uint32_t event_type;
    uint64_t sequence;
    uint64_t thread_id;
    uint64_t instruction_pc;
    uint64_t effective_address;
    uint64_t target_address;
    uint64_t module_offset;
    uint32_t access_size;
    uint32_t main_depth;
    uint32_t flags;
    uint32_t module_id;
} trace_event_t;
```

Listing APPENDIX-1.2. Schematic in-memory event model for a corrected client.

The flags field can identify whether effective address, module offset, or symbol enrichment

is valid. Zero should not be overloaded to mean “missing” because zero can be a valid offset. The module table should map the numeric module identifier to a path, load interval, file identity, and load sequence. Paths included in shared data should be normalized or redacted according to the publication policy.

1.4. Proposed thread-local state

Listing APPENDIX-1.3 shows the minimum runtime state. It is intentionally schematic. Exact allocation, registration, and callback signatures must follow `drmgr` for the selected release.

```
typedef struct {
    uint64_t thread_id;
    uint32_t main_depth;
    trace_event_t *buffer_begin;
    trace_event_t *buffer_next;
    trace_event_t *buffer_end;
    uint64_t dropped_events;
    bool flush_failed;
} trace_thread_state_t;

static bool
runtime_scope_is_active(trace_thread_state_t *state) {
    return state != NULL && state->main_depth > 0;
}
```

Listing APPENDIX-1.3. Schematic per-thread state and scope policy.

The entry callback increments depth for the thread that enters the configured root. The matching return decrements it without underflow. Thread creation policy is explicit. If worker events should be included, the parent or process state can mark a new thread according to a documented rule. The output still records the worker’s own thread identifier.

1.5. Proposed target normalization

Listing APPENDIX-1.4 replaces low-bit masking with verified containment. The helper retains the absolute address even when lookup fails. Symbol names are added offline or through a checked enrichment path.

```
target_identity_t
normalize_target(app_pc callee) {
    target_identity_t out = {0};
    out.absolute = (uint64_t)(ptr_uint_t)callee;

```

```
    module_data_t *mod = dr_lookup_module(callee);
    if (mod == NULL) {
        out.flags |= TARGET_UNRESOLVED;
        return out;
    }

    if (callee >= mod->start && callee < mod->end) {
        out.module_offset =
            (uint64_t)(callee - mod->start);
        out.module_id = intern_module(mod);
        out.flags |= TARGET_HAS_MODULE_OFFSET;
    }

    dr_free_module_data(mod);
    return out;
}
```

Listing APPENDIX-1.4. Schematic target normalization without address truncation.

This pattern avoids collisions introduced by `callee & 0xFFFF`. It also allows a report to group repeated calls across ASLR-randomized runs when the module identity is the same. If the binary changes, its build identity or hash changes, preventing an offset from being compared across incompatible code silently.

1.6. Audit acceptance checklist

Before calling the corrected code complete, a reviewer should verify the following items against source and test output:

1. all extension initialization and registration results are checked;
2. cleanup occurs once and only after the last use of each extension;
3. application registers and flags are preserved on every instrumentation path;
4. an effective-address failure produces a marked record or explicit counter rather than a false address;
5. repeated string and complex memory operations follow a documented expansion policy;
6. read and write event semantics are documented for read-modify-write instructions;
7. main-scope decisions occur at runtime and are stored per thread;

8. full targets are retained before any symbol or module filter;
9. per-thread buffers cannot overrun, and dropped or failed records are counted;
10. process exit, thread exit, signals, and abnormal termination have defined flush behavior;
11. raw totals reconcile with every human-readable summary;
12. fixtures run in automated tests and compare target checksums as well as trace contents;
13. exact source, client binary, target binary, and output hashes are retained.

1.7. Suggested repository layout

The following layout keeps implementation, fixtures, schemas, and manuscripts separate. It is a proposal rather than a claim about the missing original repository.

```
traceflow/
  CMakeLists.txt
  client/
    traceflow.c
    instrumentation.c
    record_buffer.c
    module_table.c
  include/
    trace_schema.h
  fixtures/
    indexed_array.c
    read_modify_write.c
    cache_scope.c
    threads.c
    target_calls.c
  tests/
    run_semantic_tests.py
    validate_trace.py
```

```
schema/
  trace-record-v1.md
  run-manifest-v1.schema.json
docs/
  experiment-protocol.md
thesis/
  official-erciyes-latex-source/
```

A continuous-integration job should build the client and fixtures in a pinned Linux image, run semantic tests, validate raw records, and archive manifests for failed runs. Performance experiments are better run on a controlled host because shared CI hardware produces noisy timing.

APPENDIX-2

PRESERVED EVIDENCE AND REPRODUCIBILITY RECORD

2.1. Typeset source and output plates

The plates below remove only the original editor chrome, black frames, and unused margins. Visible statements and reported values are retained, and no new run is claimed. Unaltered PNG files and the deterministic renderer remain in the source package. The full memory-map image is archived; its checked totals appear in Table 5.2 and Figure 5.4.

Initialization and main-symbol lookup SOURCE TRANSCRIPTION

```
DR_EXPORT void dr_client_main(client_id_t id, int argc,
                              const char *argv[]) {
    drsym_init();
    dr_register_exit_event(event_exit);
    dr_register_bb_event(event_basic_block);
    module_data_t *module = dr_get_main_module();
    if (module != NULL) {
        size_t offset;
        drsym_error_t sym_res = drsym_lookup_symbol(
            module->full_path, "main", &offset, DRSYM_DEMANGLE);
        if (sym_res == DRSYM_SUCCESS) {
            app_pc main_addr = module->start + offset;
            printf("Found main at offset 0x%x (address %p)\n",
                   (unsigned int)offset, main_addr);
        }
        dr_free_module_data(module);
    }
    drsym_exit();
}
```

Typeset from the retained source image. Formatting normalized; executable semantics unchanged.

Figure APPENDIX-2.1. Typeset transcription of the initialization and main-symbol lookup excerpt visible in original PDF page 38.

External-call handler SOURCE TRANSCRIPTION

```c
static void
external_call_event(app_pc caller, app_pc callee)
{
    if (!in_main)
        return;

    current_call = callee;
    module_data_t *mod = dr_lookup_module(callee);
    if (mod != NULL) {
        ptr_uint_t offset = (ptr_uint_t)callee & 0xFFFF;
        if (offset >= 0x1000 && offset <= 0x1180) {
            update_call_count(callee);
            const char *func_name = get_plt_name(offset);

            dr_fprintf(log_file, "\nExternal CALL from %p to %s@plt\n",
                       caller, func_name);
            dr_fprintf(log_file, "  Base address: %p\n", program_base);
            dr_fprintf(log_file, "  Relative offset: 0x%04x\n",
                       (unsigned int)offset);
            dr_fprintf(log_file, "  Absolute address: %p\n", callee);
        }
        dr_free_module_data(mod);
    }
    stats.ext_calls++;
}
```

Typeset from the retained source image. Formatting normalized; executable semantics unchanged.

Figure APPENDIX-2.2. Typeset transcription of the external-call handler visible in original PDF page 51.

External-call evidence reconciliation EVIDENCE SUMMARY

```
Program base address: 0x000079a333400000
Successfully instrumented main function at 0x000079a333401269
Entering main function

Named rows retained in the output:
open@plt    target=0x000079a333401150    offset=0x1150    count=1
strlen@plt  target=0x000079a333401100    offset=0x1100    count=1
write@plt   target=0x000079a3334010e0    offset=0x10e0    count=1
lseek@plt   target=0x000079a333401130    offset=0x1130    count=1
fstat@plt   target=0x000079a333401170    offset=0x1170    count=1
close@plt   target=0x000079a333401140    offset=0x1140    count=1
getpid@plt  target=0x000079a3334010f0    offset=0x10f0    count=1
printf@plt  target=0x000079a333401120    offset=0x1120    count=1

Displayed named rows: 8
Total external calls: 76
Difference not itemized by the retained output: 68
```

Reconciled from the retained output. No new execution was performed; 68 = 76 - 8.

Figure APPENDIX-2.3. Reconciled evidence summary from the preserved external-call output on original PDF page 52. The difference is $76 - 8 = 68$.

Memory operand instrumentation SOURCE TRANSCRIPTION

```
if (!in_main)
    return DR_EMIT_DEFAULT;

if (instr_reads_memory(inst)) {
    for (int i = 0; i < instr_num_srcs(inst); i++) {
        if (opnd_is_memory_reference(instr_get_src(inst, i))) {
            dr_insert_clean_call(
                drcontext, bb, inst, (void *)memref_event, false, 3,
                OPND_CREATE_INTPTR(
                    (ptr_int_t)instr_get_app_pc(inst)),
                OPND_CREATE_INT32(
                    drutil_opnd_mem_size_in_bytes(
                        instr_get_src(inst, i), inst)),
                OPND_CREATE_INT32(0));
        }
    }
}

if (instr_writes_memory(inst)) {
    for (int i = 0; i < instr_num_dsts(inst); i++) {
        if (opnd_is_memory_reference(instr_get_dst(inst, i))) {
            dr_insert_clean_call(
                drcontext, bb, inst, (void *)memref_event, false, 3,
                OPND_CREATE_INTPTR(
                    (ptr_int_t)instr_get_app_pc(inst)),
                OPND_CREATE_INT32(
                    drutil_opnd_mem_size_in_bytes(
                        instr_get_dst(inst, i), inst)),
                OPND_CREATE_INT32(1));
        }
    }
}

if (instr_is_app(inst)) {
    dr_insert_clean_call(
        drcontext, bb, inst, (void *)exec_event, false, 1,
        OPND_CREATE_INTPTR((ptr_int_t)instr_get_app_pc(inst)));
}
```

Typeset from the retained source image. Formatting normalized; executable semantics unchanged.

Figure APPENDIX-2.4. Typeset transcription of the memory-instrumentation excerpt visible in original PDF page 63.

Preserved memory-mode run OUTPUT TRANSCRIPTION

```
Memory WRITE at 0x000077cd83487cb3 (size=4)
Instruction EXECUTE at 0x000077cd83487cb3
Instruction EXECUTE at 0x000077cd83487cb9
Memory READ at 0x000077cd83487cbd (size=8)
Instruction EXECUTE at 0x000077cd83487cbd
Memory READ at 0x000077cd83487cbe (size=8)
Instruction EXECUTE at 0x000077cd83487cbe
Memory READ at 0x000077cd83487cc0 (size=8)
Instruction EXECUTE at 0x000077cd83487cc0
Memory READ at 0x000077cd83487cc2 (size=8)
Instruction EXECUTE at 0x000077cd83487cc2
Memory READ at 0x000077cd83487cc4 (size=8)
Instruction EXECUTE at 0x000077cd83487cc4
Memory READ at 0x000077cd83487cc6 (size=8)
Instruction EXECUTE at 0x000077cd83487cc6
Memory READ at 0x000077cd83487cc7 (size=8)
Instruction EXECUTE at 0x000077cd83487cc7
Instruction EXECUTE at 0x000077cd7f601160
Memory READ at 0x000077cd7f601165 (size=8)
Instruction EXECUTE at 0x000077cd7f601165
Memory READ at 0x000077cd7f601166 (size=8)
Instruction EXECUTE at 0x000077cd7f601166
Instruction EXECUTE at 0x000077cd8342a1ca

Memory Statistics in main:
Total reads: 359
Total writes: 353
Total executes: 2274
```

Typeset from the retained output image. Addresses, labels, counters, and ordering are unchanged.

Figure APPENDIX-2.5. Typeset transcription of the preserved memory-mode output and counters from original PDF page 65.

2.2. Run manifest template

Table APPENDIX-2.1 lists the minimum metadata required for a future rerun. Values are intentionally blank where no retained evidence exists.

Table APPENDIX-2.1. Reproducibility manifest template.

Field	Required value
Run identifier	Unique identifier generated before execution
UTC start and end	ISO 8601 timestamps
Host and isolation	VM or host image identity and containment policy
Distribution and kernel	Exact release strings
CPU and architecture	Model, architecture, and relevant enabled features
DynamoRIO	Release, build identity, installation path, and binary hash

Field	Required value
TraceFlow	Source commit, dirty-state indicator, compiler, flags, and client hash
Target	Path, source commit if available, compiler, flags, PIE status, and SHA-256 hash
Invocation	Exact argument vector, working directory, and relevant environment
Input	File or standard-input fixture identity and hash
Output	Raw trace, standard output, standard error, and their hashes
Result	Exit status, signal if any, completeness status, and flush errors
Counts	Instructions, reads, writes, calls, unresolved records, and dropped records
Validation	Schema result, reconciliation result, and target checksum comparison

2.3. Submission checklist

The thesis source uses the Erciyes University Department of Computer Engineering LaTeX class and the writing guide linked from the department's October 2025 project announcement [23, 24]. Administrative requirements can change, so the student should confirm the following items directly with the department before formal submission:

1. Obtain the project number from the department and confirm the required PDF filename.
2. If the department later requires a signed jury or approval form, attach the completed official form separately. It is intentionally absent from this English-only manuscript.
3. Confirm the supervisor and department-chair titles against the department directory on the submission date.
4. Run the institution's required similarity check and confirm the applicable threshold. References, unavoidable technical identifiers, and official forms should be handled according to department policy rather than manually altered to manipulate the score.
5. Verify that every source image remains legible in the final PDF and that no private path, credential, or personal contact detail appears.

6. Archive the submitted PDF, its LaTeX source package, bibliography, evidence images, and the exact final file hash.

2.4. Known unavailable fields

At the time of this revision, the project number, jury names, approval date, complete source repository, target binaries, build commands, and exact experiment environment were not available. They have not been invented. The thesis remains complete as an evidence-based audit and design study, but any future claim of reproduced execution depends on restoring or recreating those technical artifacts and running the protocol in Chapter 4.

CURRICULUM VITAE

PERSONAL INFORMATION

Name	Ahmet Göker
Student number	1030510380

EDUCATION

Degree	**Institution**	**Period**
B.Sc., Computer Engineering	Erciyes University, Kayseri	2021 to 2025

ACADEMIC INTERESTS

Dynamic binary instrumentation, Linux systems programming, runtime analysis, reverse engineering, and defensive security research.

Only information supported by the supplied thesis materials has been included. Private contact details and unsupported biographical fields have intentionally been omitted.