RTrace: Towards Better Visibility of Shared Library Execution

Huaifeng Zhang
Chalmers University of Technology
and the University of Gothenburg
huaifeng@chalmers.se

Ahmed Ali-Eldin
Chalmers University of Technology
and the University of Gothenburg
ahmed.hassan@chalmers.se

Abstract—**Software supply chain security has become a critical concern in recent years. Modern software systems increasingly depend on third-party dependencies to accelerate development. Shared libraries are the prevalent form of software sharing and hence, of third-party dependencies in modern software systems. As more attacks target the software supply chain, understanding the behavior of these dependencies is essential for identifying vulnerabilities and malicious code. Hence, accurately tracing function calls within shared libraries is critical for effective software security analysis. However, existing library function tracers often fail to meet this need. As we show in this work, state-of-the-art library function tracers are limited in effectiveness and scalability, missing a significant number of function calls and failing with more complex workloads, resulting in incomplete or misleading views of runtime behavior.**

In this paper, we present RTrace, a tracing tool designed to address the limitations of existing solutions. We analyze the root causes of why widely used tracers miss function calls and identify common pitfalls such as relying on incorrect symbol information and inability to monitor early or indirect function invocations. RTrace overcomes these challenges by incorporating comprehensive runtime monitoring, function boundary detection, and support for implicit and unconventional function calls. We compare RTrace to four state-of-the-art tracers, namely, `ltrace`, `drltrace`, `ldaudit`, and `IntelPT`. Our evaluation across 21 applications and 92 shared libraries shows that RTrace significantly outperforms existing tools in detecting function call. RTrace achieves an F1-score of at least 0.92 on all benchmarks, whereas the best existing tracer reaches only 0.74, providing more accurate visibility into shared library runtime behavior. Finally, we show how RTrace can be used to assist in detecting malicious package and in vulnerability analysis by providing a more complete view of shared library function usage.

I. Introduction

After many high-profile vulnerabilities were discovered recently, software supply chain security has become a critical concern. Today, more than 95% of enterprises actively or passively depend on third-party libraries to reduce development efforts and accelerate feature implementation [1]. Such widespread dependencies in modern software systems make vulnerabilities in third-party components potentially more severe than those in the software itself. A recent study shows that 99% of vulnerabilities in a software system originate from their dependencies [2]. A single vulnerability in a dependency can propagate through the software supply chain, impacting thousands of systems and applications that rely on it, either directly or transitively.

The growing complexity of direct and transitive dependencies in modern software systems makes it increasingly difficult to determine which components and code are actually executed in a given application [3]. This lack of visibility into runtime behavior creates a serious security blind spot, as malicious or unexpected code could be executed without the developer's awareness [4]. Therefore, the need for effective software visibility tracking - knowing exactly what code is present and executed – is more important and more challenging than ever [5]. Tracing techniques play a crucial role in addressing this challenge, offering insights not only at the system level (e.g., network traffic, file access, and syscalls), but also at the application level, including function calls, data flow, and control flow [6], [7].

In this paper, we focus on the problem of tracing shared library function calls. Shared libraries are widely-used to package and distribute code, allowing developers to reuse existing code and share functionality across different applications in Linux and Unix-like operating systems [8]. A shared library contains compiled executable code that can be loaded into memory and executed by multiple programs at runtime. A single program may depend on tens and even hundreds of shared libraries directly and transitively. For example, training a deep learning model with TensorFlow can involve almost 400 shared libraries [9]. Therefore, tracing function calls to shared libraries is a powerful technique for enhancing software transparency. It allows developers and auditors to gain fine-grained visibility into program behavior and to verify that only the intended library code is executed.

Tracing shared library function calls is a long-standing topic, and numerous tools have been developed over the years to support this task. However, the ability of these tools to meet the demands of modern software supply chain security has not been examined. While existing tools from both industry [10], [11], [12] and academia [7], [13]—such as `ltrace`, which remains widely used—offer various tracing capabilities, we find that they are insufficient for reliably tracing *all* function calls within

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.231243
www.ndss-symposium.org

shared libraries. In particular, many calls go undetected. This poses a serious risk for security and provenance analysis, as any undetected call may reflect an undocumented dependency or even a stealthy malicious payload. A compelling example is the recent XZ backdoor vulnerability (CVE-2024-3094) [14], which leveraged indirect function calls to replace a legitimate OpenSSH function with a malicious one. Such cases underscore the need for more precise and reliable library call tracing.

In this paper, we identify several common pitfalls that cause existing tracers to miss function calls in shared libraries. We then present `RTrace`, a new library function call tracer designed to overcome these limitations. `RTrace` provides a more complete and accurate view of shared library function usage during program execution. `RTrace` operates in two modes: *light mode* and *rich mode*. The light mode focuses on lightweight and accurate detection of whether a function has been executed, with minimal runtime overhead. The rich mode provides a more comprehensive view by generating function call graphs and capturing detailed runtime information for each function call, including arguments, return values, and other runtime data.

For a given program running specific workloads, `RTrace` begins by detecting all shared libraries loaded at runtime, thereby avoiding missed dependencies due to dynamic loading. For each detected library, `RTrace` does not rely solely on symbol table metadata (which may be incorrect or missing); instead, it performs an adaptive function boundary detection step to infer accurate function boundaries. It also instruments any branch-related instructions that may lead out of a function. To address calls that occur during library loading—such as resolution of indirect function calls, `RTrace` is built on top of DynamoRIO [15], an instrumentation framework that instruments code before it is executed, allowing it to capture function calls that occur during the loading of shared libraries. We evaluate `RTrace` on a suite of 21 benchmarks involving 92 shared libraries and show that it achieves an F1 score of at least 0.92, significantly outperforming existing tracers, whose best-case performance reaches only 0.72.

Beyond traditional use cases, we envision `RTrace` can be used by software developers and auditors to inspect, understand, and verify the behavior of shared libraries in their applications. We identify two key anticipated use cases for `RTrace`: (1) *pre-alert detection*: the light mode of `RTrace` can be used to proactively detect suspicious behavior in shared libraries by comparing runtime characteristics, such as the number of function calls—across different versions. Anomalies, such as a sudden increase in invoked functions within a library with stable APIs, can serve as early warning signals for potential security threats. (2) *post-incident analysis*: In the aftermath of a security incident, the comprehensive runtime information provided by the rich mode of `RTrace` allows analysts to reconstruct execution flows and understand the mechanisms used by injected or hidden malicious code. To summarize, our key contributions are as follows:

- We assess the effectiveness of four widely used library call tracers, namely, `ltrace`, `drltrace`, `ldaudit`, and `IntelPT`, and show that they often miss a substantial number of shared library function calls, raising awareness about the limitations of existing, widely used tracers.
- We identify and analyze the common pitfalls in existing tracers that lead to missed function calls in shared libraries, including incorrect symbol metadata, overlooked indirect calls, and unconventional function calls, etc.
- We design and implement `RTrace`, a library call tracer that overcomes these issues and provides comprehensive and accurate function-level tracing for shared libraries.
- We evaluate `RTrace` against the four tracers using 21 applications and show how `RTrace` can be used to assist security analysis.
- We open-source `RTrace`[1].

II. Background and Related Work

A. Supply Chain Security

Software developers usually rely on software frameworks and third-party libraries to accelerate the development process. This style of software development forms what is commonly referred to as the *software supply chain*, in which software projects depend on other software projects, libraries, and frameworks that collectively contribute to the final software product [16].

In recent years, there has been a substantial increase in the number of attacks targeting this supply chain, particularly through the exploitation of open-source third-party components [16], [17], [3], [18], [2]. Attackers inject vulnerabilities or malicious code into widely used packages, which are then propagated throughout the ecosystem, leading to the compromise of countless systems [19]. Several high-profile incidents—such as the SolarWinds breach [20], the log4j vulnerability [21], and the recent XZ backdoor [14], have underscored the urgent need for better visibility and control over software dependencies.

Malicious package detection in software supply chains, particularly for Python, NPM, and RubyGems has gained more attention in recent years. Many approaches have been proposed to detect such threats in package registries [22], [23], [24], [25], [26], with some integrated into CI/CD pipelines to block malicious packages before deployment [25], [26]. These methods commonly rely on metadata, static analysis, or dynamic analysis. Their dynamic analysis focuses on system calls or high-level package API behavior [25], [26]. However, shared library calls are often overlooked, even though high-level languages like Python and JavaScript rely on shared libraries as their low-level runtime. Understanding how packages invoke and interact with these libraries offers a more complete view of their runtime behavior and adds a valuable dimension to malicious package detection.

B. Shared Library

In Linux, shared libraries typically are compiled in the Executable and Linkable Format (ELF) and uses the `.so` suffix [27]. Shared libraries in the ELF format are divided

[1]https://github.com/negativa-ai/rtrace

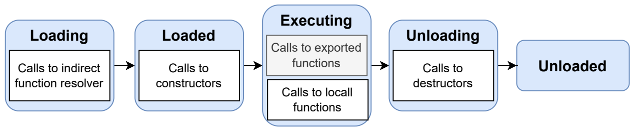

Loading
Calls to indirect function resolver
Loaded
Calls to constructors
Executing
Calls to exported functions
Calls to locall functions
Unloading
Calls to destructors
Unloaded

Fig. 1: The lifecycle of shared libraries during program execution. Traditional tracers focus on the calls to exported functions during the executing but ignore all the other phases.

into sections, among which the following are most relevant to our work: `.text` contains the executable code; `.init` and `.fini` contain the constructor and destructor functions for library initialization and cleanup, respectively; `.symtab` stores both local and global symbols of functions, variables, etc.; `.dynsym` is a subset of `.symtab`, containing only global symbols used for dynamic linking. Each symbol includes metadata such as its name (`st_name`), address (`st_value`), size (`st_size`), type and binding (`st_info`), etc. These fields are crucial for library call tracing, as they provide information about the functions and their locations in memory.

Modern system advancements have significantly increased the complexity of shared library execution. Figure 1 illustrates the typical lifecycle of shared libraries during program execution. When a program starts, the dynamic linker loads the required shared libraries into memory, initiating a sequence of phases: loading, execution, and unloading. During the loading phase, calls to indirect function resolvers (e.g., for GNU `IFUNC` symbols) may be triggered. Once the library is loaded but before the main application logic begins, constructor functions defined in the library are executed. In the execution phase, the program invokes exported functions from shared libraries, which may call local functions within the same library or functions in other libraries. Finally, during the unloading phase—typically when the program exits—destructor functions are invoked to perform any necessary cleanup.

C. Library Call Tracing

Shared library function call tracing is widely used in various domains, including software security analysis [7], debugging [28], performance analysis [29] and software debloating [30], [13], [31]. One of the most widely used tools for this purpose on Linux systems is `ltrace` [10]. It leverages the Linux `ptrace` interface to instrument the target process by setting software breakpoints (`INT3`) at entries in the Procedure Linkage Table (PLT). When a program invokes a function from a shared library, the call is redirected through the PLT, allowing `ltrace` to intercept and log the call when the PLT entry is executed. Apart from `ltrace`, several binary instrumentation frameworks such as DynamoRIO [15] and Intel PIN [32] have enabled the development of more advanced tracing tools. For example, `drltrace`[11], based on DynamoRIO, and `TinyTracer`[33], based on PIN, are designed to monitor and trace shared library function calls during program execution.

The `rtld-audit` API of the dynamic linker provides another tracing mechanism [12]. It offers hooks such as `la_symbind`, which are triggered when the dynamic linker resolves symbols, enabling interception and logging of function calls at link time. Another approach by Wang et al.[7] employs a custom dynamic loader to perform library call tracing. Additionally, Intel Processor Trace (`IntelPT`) [34] provides hardware-assisted tracing capabilities, allowing for detailed tracing of function calls and control flow in applications.

Despite these various approaches, all tools we examined have significant limitations; These tools heavily rely on symbol table information to determine which functions to trace and often miss calls that occur outside the main execution phase of the shared library lifecycle (Figure 1). As a result, they frequently miss functions due to missing or inaccurate symbol data and implicit calls triggered during the loading or unloading phases. Such missed functions can be part of benign initialization routines or exploitations for malicious purposes. Some tools also incur substantial runtime overhead, making them impractical for complex applications, as we show in Section V.

D. Binary Analysis

Function boundary detection in binaries aims to identify the start and end of functions in compiled binaries without relying on source code. Research in this area can be broadly classified into three categories: control flow graph (CFG)-based methods [35], [36], [37], [38], pattern-based methods [39], [40], [41], and machine learning-based methods [42], [43], [44]. Among these, two techniques are particularly relevant to our work: FunSeeker [41] and Nucleus [36]. FunSeeker uses a pattern-based approach to detect `endbr` instructions in binaries that support Intel's Control-flow Enforcement Technology (CET) [45], a modern hardware feature designed to mitigate control-flow hijacking attacks. Nucleus, on the other hand, utilizes a CFG-based approach that identifies function boundaries through connected component analysis of the binary's control flow graph. Unlike FunSeeker, Nucleus is more general and applicable to binaries without CET support.

Function signature inference aims to deduce a function's signature, including arity (the number of arguments) and types (both arguments and return value), from a compiled binary. Static analysis, such as control-flow, data-flow and value set analysis, is commonly used to infer function signatures [40], [38] and variable type [46]. Binary analysis platforms, like Ghidra [40] and Angr [38], leverage known library signatures and perform static data-flow analysis to infer likely types by examining how values are propagated and used. De Goër et al.[47] propose a lightweight dynamic analysis approach that monitors runtime behavior to recover function parameters and return values, focusing on three basic types {`addr, int, float`}. Hybrid methods that combine static and dynamic analysis can improve inference accuracy, as demonstrated in works like TypeSqueezer[48]. Most of these methods focus on System V AMD64 ABI [49] calling convention, which is commonly used on Linux x86-64 systems.

III. A Call for a Better Tracer

As discussed earlier, advancements in modern systems have significantly increased the complexity of shared library execution, introducing many implicit function calls that occur outside the direct control of the application. Existing tracers have not evolved to accommodate these changes, and as a result, they often fail to provide accurate and comprehensive visibility into shared library behavior at runtime. To give a simple example, Listing 2 in the appendix illustrates a shared library that defines several functions, some of which are implicitly invoked. When a program calls the function `indirect_func`, it triggers the execution of all functions shown in the listing. However, to the best of our knowledge, no existing tracer is capable of detecting all these calls.

We have identified the following key limitations of traditional tracers that contribute to these blind spots. Firstly, traditional tracers typically rely on symbol tables (`.symtab` and `.dynsym` sections) to determine which function to trace. For instance, `drltrace` uses the `.dynsym` section to trace calls to exported functions, but fails to capture local function calls within shared libraries. While tracing interactions between the main program and shared libraries is helpful, developers are usually aware of these calls. What is more critical for security and provenance is understanding what happens inside the library—how functions call each other, and how they interact with other libraries. This level of tracing is challenging because local functions are typically only listed in `.symtab`, which is often missing in stripped libraries. Without this symbol information, traditional tracers cannot identify or trace local functions. Additionally, even when function symbols are available, the `size` field in the symbol table is optional and may be inaccurate.

Secondly, certain functions, such as those used in indirect function resolution, may be executed before the shared library is fully loaded. As most existing tracers begin monitoring only after library loading, they fail to capture these early invocations. In addition, shared libraries may define constructor (`.init`) and destructor (`.fini`) functions that are executed before the `main()` function and when the process is terminating, respectively. These functions are invoked implicitly by the dynamic linker and are not explicitly called in the application's code. Existing tracers frequently miss these lifecycle hooks, despite their importance. For example, `ltrace` failed to detect any function calls during the loading phase of the shared library in Listing 2. Alarmingly, this implicit execution makes them an ideal vector for stealthy attacks. For example, the recent XZ backdoor (CVE-2024-3094) exploited an indirect function call to overwrite a legitimate function in OpenSSH with a malicious one [14].

Thirdly, some functions do not follow standard call conventions [50]. They may begin execution at an internal offset (not `st_value` declared in the symbol table), exit without a `RET` instruction, or be invoked using non-`CALL` instructions like `JMP` or indirect branches. Tracers that assume conventional function entries and exits based on symbol information will overlook such calls.

Finally, as software systems grow in complexity, existing tracers become increasingly impractical. For instance, when attempting to trace a machine learning model training program, `ltrace` failed with an error: "too many call stacks". Other tools, such as `drltrace`, introduced extreme overheads—slowing down a 50-second workload to over 1 hours, a more than 70× degradation. At the other end, `IntelPT`, as a hardware tracing tool, is only available on about 10% of AWS instance types [51] and produces more than 200GB tracing data in the machine learning training case, making it costly and difficult to scale in practice.

As software supply chain security becomes increasingly critical, gaining visibility into the behavior of shared libraries is essential for ensuring the integrity and trustworthiness of modern software systems. This necessitates the development of a more robust and accurate tracer—one that can comprehensively capture function calls within shared libraries and overcome the limitations of existing tools.

IV. RTrace: An accurate Tracer

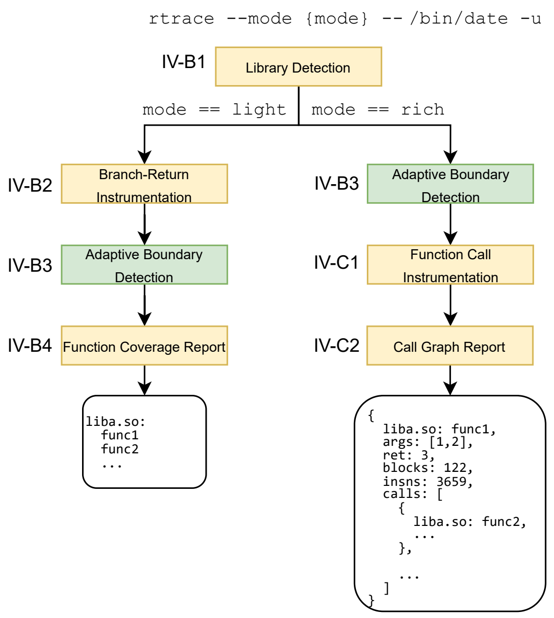
A. Overview

Figure 2 illustrates the workflow of `RTrace`, which operates in two distinct modes: *Light Mode* and *Rich Mode*. The light mode only detects whether a function has been executed, without collecting additional runtime information, with very low runtime overhead. The rich mode traces every call of a function and captures detailed runtime information, such as function arguments and return values, but with increased runtime overhead. The light mode is particularly suitable for proactive monitoring within CI/CD pipelines, while the rich mode is more appropriate for in-depth post-incident analysis when anomalies are detected.

Both modes begin by identifying shared libraries at runtime. In the light mode, `RTrace` instruments branch and return instructions only upon their first execution. It then performs function boundary detection to determine the start and end addresses of functions within each shared library. Using this information, `RTrace` generate an accurate list of functions executed by the application, achieving high accuracy with minimal performance impact.

In the rich mode, function boundary detection is performed first to identify the functions to be instrumented. Once function boundaries are identified, `RTrace` instruments each function call to capture detailed runtime data generating a call graph report for each call of a function, including its arguments, return values, number of executed basic blocks, number of executed instructions, and any other functions it calls.

We implement `RTrace` based on DynamoRIO, with over 1,500 lines of C++ for instrumentation and 1300 lines of Python for report generation. In the following sections, we provide an in-depth discussion of each component in the workflow. For each step, we first examine the pitfalls that informed our design choices, followed by a detailed explanation of the solutions to address these pitfalls.



```
rtrace --mode {mode} -- /bin/date -u
```

```
liba.so:
  func1
  func2
  ...
```

```
{
  liba.so: func1,
  args: [1,2],
  ret: 3,
  blocks: 122,
  insns: 3659,
  calls: [
    {
      liba.so: func2,
      ...
    },

    ...
  ]
}
```

Fig. 2: Workflow of the two modes of RTrace to trace executed library functions triggered by the date utility.

B. Light Mode

Light Mode is designed to detect all executed functions efficiently and accurately. It ultimately produces a function coverage report that lists the set of functions executed during program execution.

1) Library Detection: Library Detection aims to accurately identify all shared libraries used by a program during execution, including those loaded both directly and transitively at runtime.
Pitfall. Traditional methods rely on static analysis to determine the libraries in use [52], [13]. These approaches inspect program binaries by parsing the ELF `.dynamic` section to extract dependency information. However, program binaries and shared libraries can also load shared libraries dynamically at runtime (loaded by `dlopen`). Static analysis cannot detect these dynamically-loaded libraries [52].
Solution. To address this limitation, RTrace performs runtime library detection using hook functions provided by the DynamoRIO framework. This enables monitoring of shared library loading events as they occur. Whenever a new library is loaded, RTrace records its file path and its memory range, defined by its start and end addresses. A sample output from this detection process is shown in Listing 3 in the appendix.

2) Branch-Return Instrumentation: To identify which functions are executed, we must instrument the application's execution flow to detect function usage at runtime.
Pitfall. To detect used functions, traditional library call tracers utilize the rules of *Call Convention* [50]. One approach is to trace the `CALL` instruction and analyze the operand of the `CALL` instruction to get which function is called. Another approach inserts trap instructions at the beginning of a function and the `RET` instruction, assuming that the execution starts from the first instruction of a function and ends at the `RET` instruction. However, from our observation of many real-world shared libraries, we found that call convention is not always followed. Some functions are not called by the `CALL` instruction, instead, they are called by the `JMP` instruction directly, which nullifies the first approach. On the other hand, the entrance and exit of a function are not always at the beginning and the `RET` instruction, which invalidates the other approach.
Solution. Based on this observation, we assume that if a function is executed, it must eventually exit through a branch-related instruction such as `RET`, `JMP`, or `CALL`. Therefore, to detect function usage, we instrument two types of instructions: branch-related instructions and return-related instructions. We intentionally avoid instrumenting function entry points for two key reasons. First, not all functions are entered at their start addresses. For example, during a machine learning training workload, we observed that the function `__memcpy_sse2_unaligned_erms` in `libc` (at address `0xba900`) was entered from the middle rather than the defined entry point. Second, branch and return instrumentation does not rely on function boundary information. This enables instrumentation of the target application without first detecting function boundaries, thereby reducing the overhead introduced during instrumentation. The light mode takes advantage of this property to perform accurate function tracing with minimal performance impact. Additionally, to minimize overhead without sacrificing accuracy, each branch and return instruction is instrumented only once, during its first execution. Our evaluation shows that branch and return instrumentation alone is sufficient to detect function execution accurately. The output of this branch-return instrumentation is a list of executed instruction addresses, which are later matched with function boundaries to identify executed functions. Listing 4 in the appendix shows a sample output of the branch-return instrumentation.

3) Adaptive Boundary Detection: After identifying all shared libraries used by the target application and instrumenting branch and return instructions, The next step is to determine the boundaries, i.e., the start and end addresses of each function within these libraries. Function boundaries are essential for accurately identifying executed functions.
Pitfall. A common approach for boundary detection involves parsing the symbol table of each shared library to retrieve function names along with their virtual addresses and sizes. However, two major issues compromise its effectiveness. First, many shared libraries do not have the debug symbol table (`.symtab`) and include only the dynamic symbol table (`.dynsym`), which contains a limited subset of function symbols. Second, the function size field in a symbol table is optional in the ELF format and may not be set by the compiler. As a result, solely depending on symbol tables can lead to incomplete or inaccurate function boundary detection.
Solution. To overcome these limitations, RTrace employs an adaptive function boundary deduction process for each shared library, as illustrated in Figure 3. Our evaluation of existing function boundary detection algorithms reveals depending on the metadata available in each shared library,

different algorithms can be used to achieve better accuracy and efficiency. Based on this observation, we propose an adaptive boundary detection workflow that dynamically selects the most suitable boundary detection algorithm according to the metadata available in each shared library.

If the shared library includes the debug symbol table (`.symtab`), `RTrace` applies a linear detection algorithm (Algorithm 1) to deduce function boundaries. The linear detection algorithm works straightforwardly by iterating through the symbol table, identifying function symbols, and calculating their start address which is the `st_value` field of the symbol. Then the end address of a function is the start address of the next function. If the debug symbol table is absent but the library supports CET, `RTrace` uses the FunSeeker algorithm, which is specifically designed for CET-enabled binaries; If neither debug symbols nor CET are available, `RTrace` resorts to the Nucleus algorithm for boundary detection, which is designed for general binaries. The output of this process is a list of functions with their start and end offsets, as shown in Listing 5 in the appendix.

Algorithm 1: Linear Detection of Function Boundaries

```
Input  : A Shared library
Output : A list of detected function boundaries $S$
/* The index of $S$ starting from 0    */
$S \leftarrow []$;
for $s$ in .symtab or .dynsym do
    if $s.type$ in {FUNC, IFUNC} then
        Add $s$ to $S$;
    end
end
for $i = 0$ to $|S| - 1$ do
    $s_i.start \leftarrow s_i.st\_value$;
end
Sort $S$ by $s.start$ in ascending order;
for $i = 1$ to $|S| - 1$ do
    $s_{i-1}.end \leftarrow s_i.start$;
end
/* $t$ is the end offset of the .text
   section                             */
$S_{|S|-1}.end \leftarrow t$;
```

4) Function Coverage Report: The function coverage report summarizes the set of functions that are actually executed during runtime. This report is generated by combining the results from the library detection, branch-return instrumentation, and boundary detection steps. For each instruction address captured by the branch-return instrumentation, we first determine the corresponding shared library it belongs to. Then, using the previously detected function boundaries within that library, we identify the specific function to which the address belongs. Finally, this function is reported as executed.

C. Rich Mode

The rich mode provides detailed runtime information about function execution and ultimately generates a function call graph report. Unlike the light mode, the rich mode performs function boundary detection as described in §IV-B3 *before* function call instrumentation, to ensure that all functions are accurately identified and instrumented. Once function boundaries are detected, the rich mode performs function call instrumentation and generate a call graph report per thread. These two steps are discussed in the following sections.

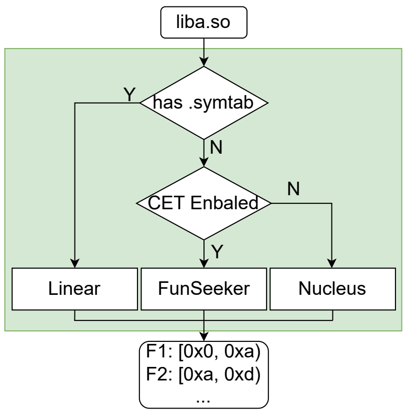


Fig. 3: Adaptive Boundary Detection Workflow.

1) Function Call Instrumentation: Function call instrumentation is designed to collect detailed runtime information about each executed function. This process has two primary challenges: (1) inferring function signatures, which describe the number and types of arguments and return types; and (2) handling calling conventions to correctly extract these values at runtime. Function signatures are not directly available in shared libraries. Existing tools such as `ltrace` and `drltrace` rely on user-provided signatures, which limits scalability. To address this, we automatically infer function signatures using static analysis. After evaluating several tools, we use Angr [38] to infer function signatures due to its better performance on large shared libraries.

Calling conventions is also essential for extracting function arguments and return values. While various calling conventions exist, the System V AMD64 ABI [49] is the most widely adopted standard on x86-64 Linux systems. Consequently, in line with existing tools such as `ltrace` and `drltrace`, the rich mode in our tool currently supports only this convention. Due to limitations of DynamoRIO, our implementation currently supports retrieving values of type `addr` (pointer type) and `int` (all integers types), but not `float`. However, this is not a fundamental limitation of our approach, and we plan to add support for `float` types in a future version.

The function instrumentation proceeds as follows; For each function identified via boundary detection, we use Angr to infer its signature to obtain the number and types of arguments and return types. We then instrument the function's entry and exit points. At entry, the specified number of arguments is read from the registers or stack according to the calling convention. At exit, the return value is similarly extracted. In addition to arguments and return values, we collect additional runtime metrics including the number of basic blocks executed, the number of instructions executed, and the functions invoked. For performance reasons, these runtime data is not assembled into a call graph during function instrumentation. Instead, two intermediate files are generated for each thread to store the collected data:

- A *basic block file* that records block start addresses and the instruction counts of each block.
- A *function trace file* that logs entry and exit events, argument values, basic block executions, and return values.

The basic block file contains a mapping of basic block start addresses to their respective instruction counts. The function trace file follows a structured format. Upon function entry, a line of the form `Entry`:$func\_addr$ is logged, where $func\_addr$ is the start address of the function being entered. This is followed by lines in the form of `Arg`:arg_value for each argument, and `BB`:bb_addr for each basic block executed, where arg_value is the value of the argument and bb_addr is the start address of the basic block. Upon return, it logs `Ret`:ret_value for the return value, and `Exit`:$func\_addr$ to indicate the function of address $func\_addr$ exits. Listing 6 in the appendix shows a sample of the function trace file. These two intermediate files are later parsed to produce the final function call graph report.

2) Call Graph Report: At this step, a dynamic function call graph is constructed by parsing two intermediate files produced by the previous function instrumentation step. The call graph report summarizes this information in a structured format, represented in Listing 1. While the listing omits details such as function names and shared libraries for brevity, the actual report includes this additional metadata.

Listing 1: Data structure used to represent each function call in the dynamic call graph.

```
struct Call {
  void* addr;          // Function address
  vector<void*> args;  // Function arguments
  void* ret_val;       // Return value
  int executed_blocks;// Number of executed basic blocks
  int executed_insts;  // Number of executed instructions
  vector<Call> calls;  // Functions called by this call
}
```

The function trace file is processed using a simple stack-based algorithm that resembles the way call stacks are managed at runtime, as shown in Algorithm 2. As each function entry `Entry`:$func\_addr$ is encountered in the function trace file, a new `Call` struct is created with the *addr* field set to $func\_addr$. This struct is then pushed onto a stack to represent the current call context. Any subsequent arguments, return values, basic blocks and function calls in the function trace file are recorded in the current call context. When a function exits, the call is popped from the stack. A representative output of the call graph report is also shown in Listing 7 in the appendix. Each thread generates a separate call graph report.

Algorithm 2: Call Graph Report

```
Input  : Basic block file $F_b$ and function trace file $F_f$
Output : A call graph starting from root
$stack \leftarrow$ [root];
/* block_info maps basic block addresses to their instruction counts */
$block\_info \leftarrow$ parse_basic_block_file($F_b$);
for $l$ in $F_f$ do
    if is_entry($l$) then
        Create a new Call struct $call$;
        $call.addr \leftarrow l.func\_addr$;
        $stack$.top().$calls$.push_back($call$);
        $stack$.push($call$);
    end
    else if is_arg($l$) then
        $stack$.top().$args$.push_back($l.arg\_value$);
    end
    else if is_bb($l$) then
        $stack$.top().$executed\_blocks \leftarrow$
          $stack$.top().$executed\_blocks + 1$;
        $stack$.top().$executed\_insts \leftarrow$
          $stack$.top().$executed\_insts +$
          $block\_info[l.bb\_addr]$;
    end
    else if is_ret($l$) then
        $stack$.top().$ret\_val \leftarrow l.ret\_value$;
    end
    else if is_exit($l$) then
        $stack$.pop();
    end
end
```

V. Experiment

In this section, we evaluate the effectiveness of existing library function tracers and `RTrace`.

A. Experiment Setup

Evaluated Metrics. Our evaluation focuses on the accuracy of detection of executed functions. We use precision, recall and F1-score as the metrics to evaluate the accuracy of each tracer. Precision measures the ratio of detected functions that are actually executed, while recall measures the ratio of executed functions that are detected. F1-score is the harmonic mean of precision and recall, providing a single metric to evaluate the overall accuracy.

TABLE I: Overview of evaluated library call tracers.

Tracer	**Type**	**Mechanism**
`ltrace`	Industry Tool	PLT-based library function calls tracing
`drltrace`	Industry Tool	DynamoRIO-based library function calls tracing
`ldaudit`	Author Implementation	Uses `la_symbind64` callback during symbol binding
`IntelPT`	Industry Tool	Hardware tracing

Benchmark Applications, We evaluate the tracers with 21 applications, 20 of which are from the Debloater-Eval benchmark [30], and one machine learning application. The machine learning application involves training a convolutional neural network (CNN) model using LibTorch [53] on the MNIST dataset. Table II summarizes the evaluated applications details. We run each application traced by each tracer and collect the function calls detected by each tracer.

TABLE II: Evaluated applications and their versions.

Category	Application	Version
Debloater-Eval Benchmark [30]	bzip2	1.0.5
	chown	8.2
	date	8.21
	grep	2.19
	gzip	1.2.4
	mkdir	5.2.1
	rm	8.4
	sort	8.16
	tar	1.14
	uniq	8.32
	bftpd	6.1
	wget	1.20.3
	objdump	2.40
	memcached	1.6.18
	lighttpd	1.4
	make	4.2
	imageMagick	7.0.1
	nmap	7.93
	nginx	7.93
	poppler	0.60
Machine Learning	train-cnn	LibTorch (2.7)

Oracle. To compute precision and recall, we implement an oracle that establishes ground truth for executed functions. The oracle operates in two main steps for each shared library used in the benchmarks: (1) obtaining correct function boundaries, and (2) determining which functions are actually executed. The oracle downloads debug symbol tables for each library used in the benchmarks, which serve as the ground truth for function boundaries. For generic shared libraries, we use the `find-dbgsym-packages` tool to retrieve their debug symbols [54]. For CUDA libraries used in the machine learning benchmark, we utilize the NVIDIA Debug Symbol Service [55]. To determine the functions executed, the oracle performs instruction-level tracing, recording the address of every instruction executed during benchmark runs. By mapping these instruction addresses to the function boundaries identified in the first step, a function is considered executed if at least one of its instructions is executed at runtime. Across all benchmarks, a total of 103 shared libraries are used, of which 92 have available debug symbols and thus serve as ground truth. All evaluation results are based on these 92 libraries.

Evaluated Tracers. We select the evaluated tracers based on the following criteria: (1) The tracer must automatically detect all executed functions, rather than relying on user-specified function lists; (2) The tracer must be application-specific, i.e., it should trace only the functions triggered by the target application, not those invoked by other processes; (3) The tracer must be compatible with Linux. Based on these criteria, we select four tracers for evaluation, as shown in Table I. Note that during the evaluation of all tracers, no additional debug symbols are downloaded using the methods as the oracle does. Debug symbol tables were used *only* if the shared libraries themselves included embedded debug symbols. Among the 92 libraries with ground truth, only 8 had embedded debug symbols within the shared libraries.

All experiments were conducted on an Ubuntu 24.04 system with Intel PT support, running inside privileged Docker containers to ensure isolation. Privileged access is necessary to enable evaluation of the hardware-assisted tracer `IntelPT`.

B. Accuracy and Performance of the Light Mode

Accuracy.Table III presents the precision, recall and F1-score of each tracer on evaluated benchmarks. While all tracers demonstrate high precision, this metric alone is insufficient for evaluating effectiveness. A tracer that identifies only a single executed function may have a precision score of 1, yet miss the majority of executed functions, resulting in low recall. This phenomenon appears in every compared tracers, which exhibit notably low recall across benchmarks. For instance, the highest recall among them is achieved by `IntelPT` on the `make` benchmark, with a value of 0.58. In contrast, `RTrace` consistently achieves both high precision and recall, exceeding 0.92 for both metrics across all benchmarks. `RTrace` achieves the highest F1-score across all benchmarks, with its lowest being 0.92 on the `ml_train` benchmark and the highest reaching 1 on the `gzip` benchmark. These results show that `RTrace` provides a significantly more accurate view of shared library execution behavior.

Performance. We also evaluate the runtime overhead introduced by each tracer on the benchmark applications. Each benchmark was executed 10 times without tracing to establish a baseline, and then 10 times under each tracer. The average execution times from both scenarios are used to compute the relative slowdown, as reported in Table IV. Among all tools, the light mode of `RTrace` 's consistently shows the second-lowest overhead, only behind `ldaudit`, which has limited utility due to its poor precision and recall (Table III).

Excluding `ldaudit`, `RTrace` light mode exhibits the most stable and moderate overhead across benchmarks, with a maximum slowdown of 30× in the worst case (`make`). This overhead is mainly due to (1) initialization of DynamoRIO framework, and (2) first-time instrumentation of branch and return instructions. For longer-running workloads such as `bftpd`, `memcached`, and `ml_train`, the overhead becomes negligible, with at most a 2× slowdown, comparable to the hardware-assisted tracer `IntelPT`. In contrast, `ltrace` and `drltrace` exhibit excessive overhead for certain cases. For example, in `nmap`, `ltrace` incurs a 11,756× slowdown and `drltrace` 113×, while `RTrace` light mode incurs only 5×. This is because this benchmark contains many repetitive function calls, and `RTrace` light mode only traces the first execution of each function, whereas `ltrace` and `drltrace` trace every function call, resulting in notably higher overhead.

TABLE III: Precision, recall and F1-score of each tracer on the benchmarks. P is precision, R is recall, and F1 is F1-score.

Benchmark	`ltrace`			`drltrace`			`ldaudit`			`IntelPT`			`RTrace (light)`		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
bftpd	0.96	0.38	0.54	0.97	0.35	0.52	0.43	0.21	0.28	0.99	0.50	0.67	0.96	0.98	0.97
bzip2	0.96	0.28	0.43	0.97	0.32	0.48	0.71	0.11	0.20	0.99	0.54	0.70	0.97	1.00	0.98
chown	0.97	0.31	0.47	0.97	0.35	0.52	0.71	0.08	0.15	0.99	0.52	0.68	0.97	0.99	0.98
date	0.94	0.27	0.41	0.95	0.32	0.48	0.69	0.09	0.15	0.99	0.53	0.69	0.99	0.99	0.99
grep	0.98	0.27	0.42	0.98	0.31	0.47	0.80	0.13	0.22	0.99	0.55	0.71	0.99	0.99	0.99
gzip	1.00	0.15	0.26	0.97	0.19	0.31	0.61	0.11	0.18	0.98	0.50	0.66	0.99	1.00	1.00
imagemagick	0.99	0.25	0.40	0.99	0.56	0.71	0.92	0.49	0.64	1.00	0.56	0.72	0.98	0.99	0.98
lighttpd	0.97	0.29	0.45	0.97	0.31	0.47	0.81	0.14	0.24	0.99	0.54	0.70	0.99	0.99	0.99
make	0.95	0.36	0.52	0.95	0.39	0.56	0.61	0.17	0.26	0.98	0.58	0.73	0.96	0.99	0.97
memcached	0.96	0.34	0.51	0.94	0.35	0.51	0.62	0.22	0.32	0.99	0.56	0.71	0.94	0.98	0.96
mkdir	0.96	0.17	0.28	0.97	0.24	0.39	0.56	0.06	0.11	0.99	0.53	0.69	0.98	0.99	0.99
nginx	0.98	0.33	0.49	0.98	0.35	0.52	0.84	0.17	0.29	0.99	0.55	0.71	0.97	0.98	0.97
nmap	0.97	0.29	0.44	0.98	0.39	0.56	0.14	0.30	0.19	0.99	0.46	0.63	0.97	0.97	0.97
objdump	0.94	0.27	0.42	0.95	0.31	0.47	0.53	0.16	0.25	0.99	0.54	0.70	0.95	0.99	0.97
poppler	0.99	0.36	0.53	0.99	0.44	0.60	0.11	0.17	0.13	1.00	0.38	0.55	0.99	0.96	0.98
rm	0.97	0.22	0.36	0.97	0.28	0.43	0.68	0.09	0.16	0.88	0.52	0.65	0.99	0.99	0.99
sort	0.93	0.27	0.41	0.94	0.31	0.47	0.75	0.13	0.22	0.99	0.46	0.62	0.99	0.99	0.99
tar	0.96	0.35	0.51	0.98	0.31	0.48	0.37	0.18	0.24	0.99	0.49	0.66	0.98	0.99	0.98
uniq	0.95	0.26	0.41	0.96	0.32	0.48	0.72	0.09	0.17	0.99	0.53	0.69	0.99	0.99	0.99
wget	N/A[1]	N/A[1]	N/A[1]	1.00	0.41	0.58	0.54	0.15	0.24	1.00	0.26	0.41	0.99	0.98	0.98
ml_train	N/A[2]	N/A[2]	N/A[2]	0.99	0.03	0.06	0.88	0.03	0.05	1.00	0.24	0.38	0.93	0.92	0.92

[1] Due to the significant slowdown introduced by `ltrace`, the `wget` benchmark repeatedly timed out and failed run correctly.
[2] The `ml_train` benchmark failed to run with `ltrace`.

TABLE IV: Relative slowdown of `RTrace` compared to other tracers. Results are averaged over 10 runs; standard deviations are omitted as they remain below 10% across all benchmarks.

Benchmark	**Original(ms)**	`ltrace`	`drltrace`	`ldaudit`	`IntelPT`	`RTrace (light)`	`RTrace (rich)`
bftpd	2,082	4×	1×	1×	1×	1×	1×
bzip2	13	47×	7×	1×	96×	8×	188×
chown	4	170×	25×	2×	329×	23×	128×
date	4	99×	23×	2×	330×	22×	116×
grep	5	304×	23×	1×	263×	21×	123×
gzip	4	42×	18×	2×	329×	19×	96×
imagemagick	26	1,048×	37×	1×	47×	12×	343×
lighttpd	4	204×	24×	2×	330×	25×	137×
make	8	3,158×	40×	2×	164×	30×	230×
memcached	2,011	9×	1×	1×	2×	1×	1×
mkdir	4	19×	20×	2×	328×	20×	102×
nginx	1,036	14×	1×	1×	2×	1×	2×
nmap	51	11,756×	113×	1×	25×	5×	831×
objdump	13	757×	21×	1×	93×	13×	215×
poppler	27	6,504×	102×	1×	45×	14×	613×
rm	4	125×	22×	2×	330×	22×	114×
sort	5	155×	21×	1×	264×	19×	105×
tar	7	409×	23×	2×	188×	20×	139×
uniq	5	76×	17×	1×	264×	17×	90×
wget	85	N/A	165×	1×	15×	9×	856×
ml_train	53,218	N/A	70×	1×	1×	2×	345×

C. Accuracy and Performance of the Rich Mode

For the rich mode of `RTrace`, we evaluate both the accuracy of capturing correct function arguments and return values, as well as the runtime overhead it introduces. Obtaining ground truth for runtime arguments and return values is challenging, as no existing tools can automatically capture these values for all functions, to the best of our knowledge. As a result, establishing ground truth requires significant manual effort. Therefore, instead of evaluating all 21 benchmarks, we focus on the standard C library (`libc`) and utilize the `libc-test` project [56]. We selected 117 functions from the functional tests that do not involve `float` types, as our rich mode currently does not support tracing `float` arguments or return values yet. We manually modified the source code of the test cases to log the arguments and return values of the tested functions. Then, we ran these tests with `RTrace` in rich mode and compared the traced values of the function with the logged ground truth. Note that we do not compare against the other tracers. Because `ltrace` or `drltrace` require user-specified function signatures, making automated large-scale evaluation infeasible, while `ldaudit` and `IntelPT` do not support capturing argument or return values.

TABLE V: Error analysis of argument and return value tracing of the rich mode (out of 117 functions).

Category	**Count**	**Root Cause**
Wrong Argument	12	8: Incorrect function signature 4 Incorrect function boundary
Wrong Return	5	1: Incorrect function signature 4: Incorrect function boundary
Total Failure	12/117	

Accuracy. Table V summarizes the accuracy of rich mode tracing. Out of 117 function calls, 12 are traced with incorrect argument values or return values. For argument values, eight of them are caused by inaccurate function signatures (e.g., the actual number of arguments exceeds what `RTrace` inferred), and four due to incorrect function boundary detection. For instance, in the `fscanf` test case, which uses variadic arguments, the rich mode only detected two arguments, causing the remaining ones to be missed. Regarding return values, five out of 117 function calls are traced incorrectly. Among them, one is due to incorrect function signature and four due to incorrect function boundary detection. For example, `RTrace` failed to detect boundaries of some functions like `memset`, resulting in no return value being recorded. To address these issues, we modified `RTrace` to allow users to override the automatically inferred function signatures and boundaries with user-specified ones. This design helps improve tracing accuracy for functions of particular interest.

Performance. Regarding performance overhead, we evaluate the rich mode using the benchmarks in Table II, with results shown in Table IV. As expected, the rich mode incurs higher overhead than the light mode. This increase is due to three factors: Firstly, unlike light mode, which only traces the first call to each function, rich mode traces every function call, and for each call it inspects registers and walks the call stack to extract function arguments and return values, which is a major source of overhead. Secondly, it also performs function boundary detection before instrumentation and must load the resulting boundary information into memory, adding setup overhead. Finally, boundary detection also increases the number of functions to be instrumented, which further increases the overall overhead. Despite this overhead, we argue that the rich mode remains practical, as it is primarily intended for post-incident or offline analysis where completeness and detail are prioritized over runtime performance. In addition, it is comparable in performance to `ltrace`.

D. Impacts of Boundary Detection Algorithm

In this section, we evaluate how the choice of boundary detection algorithm affects the accuracy of the light mode of `RTrace`. As outlined in the workflow in Figure 3, we categorize shared libraries into three groups based on their available metadata: (1) libraries containing a debug symbol table (`symtab`), (2) libraries with CET enabled (`CET`), and (3) libraries that lack both a debug symbol table and CET support (`Neither`). Out of a total of 92 shared libraries, 8 contain debug symbol tables, 71 have CET enabled, and 13 fall into the third category with neither feature. We apply the three boundary detection algorithms respectively to *all* groups and evaluate their performance.

Table VI summarizes the results. As shown, the linear algorithm performs best on libraries with debug symbol tables, achieving the highest precision, recall, and F1-score of 1. This also indicates that branch-return instrumentation is effective in tracing function execution. For CET-enabled libraries, the FunSeeker algorithm consistently yields the most accurate results, with precision, recall, and F1-score all of 0.98. Meanwhile, the Nucleus algorithm provides the best recall and F1-score for libraries without debug symbol tables or CET, where linear and FunSeeker algorithms shows lower recall and F1-score. These results further validate the boundary detection workflow illustrated in Figure 3: Rather than relying on a single boundary detection algorithm for all libraries, selecting appropriate algorithm based on the available metadata within each library results in the most accurate tracing results.

TABLE VI: Accuracy of executed function tracing by `RTrace` light mode using different boundary detection algorithms. P is precision, R is recall, and F1 is F1-score.

Category	Metric	Linear	FunSeeker	Nucleus
`symtab`	P	1.00	1.00	0.98
	R	1.00	0.98	0.85
	F1	**1.00**	0.99	0.91
`CET`	P	0.85	0.98	0.94
	R	0.51	0.98	0.91
	F1	0.64	**0.98**	0.93
`Neither`	P	0.99	0.83	0.83
	R	0.37	0.55	0.83
	F1	0.54	0.66	**0.83**

E. Ablation Study

To better understand the contribution of each component within `RTrace`, we perform an ablation study by selectively disabling individual components and measuring their impact on function detection accuracy. We disabled the following components: (1) boundary detection, as shown in §IV-B3; (2) branch instrumentation and (3) return instrumentation. Table VII reports the corresponding changes in precision, recall, and F1-score when each component is removed.

We observe that disabling the boundary detection component significantly reduces the effectiveness of `RTrace`, resulting in substantial drops in precision, recall, and F1-score across all benchmarks. For instance, in the `gzip` benchmark, removing this component leads to a 0.23 drop in precision, a 0.64 drop in recall, and an overall F1-score reduction of 0.50. Boundary detection impacts both precision and recall because failing to detect a correct function boundary can simultaneously introduce a false positive and a false negative. An example illustrating this effect is provided in §VIII-C in the appendix.

Branch instrumentation also has a pronounced effect, particularly on recall. Disabling it results in a 0.43 recall reduction for the ml_train benchmark, indicating that many functions exit using branch instructions instead of standard return instructions. However, disabling branch instrumentation has a positive effect on precision, with many benchmarks showing a slight increase in precision. This is because disabling branch instrumentation results in fewer detected functions, which also reduces false positives, thus marginally increase precision.

Return instrumentation exhibits a similar pattern: its absence leads to a notable decline in recall but has little effect on precision. For example, in the wget benchmark, disabling return instrumentation decreases recall by 0.15, while precision remains almost unchanged. Overall, all the components of RTrace contribute to its effectiveness, with boundary detection being the most critical. We present specific examples illustrating how the disabling of individual components in RTrace leads to missed function detections in §VIII-C in the appendix.

TABLE VII: Performance change when disabling each component of RTrace. Negative values indicate degradation; Positive values indicate improvement. P is precision, R is recall, and F1 is F1-score.

Benchmark	**Boundary Detection**			**Branch**			**Return**		
	P	R	F1	P	R	F1	P	R	F1
bftpd	-0.12	-0.51	-0.37	0.01	-0.11	-0.05	-0.00	-0.04	-0.02
bzip2	-0.16	-0.55	-0.41	0.02	-0.09	-0.04	-0.00	-0.04	-0.02
chown	-0.19	-0.55	-0.42	0.02	-0.10	-0.05	-0.00	-0.04	-0.02
date	-0.21	-0.57	-0.44	0.00	-0.10	-0.05	-0.00	-0.04	-0.02
grep	-0.21	-0.59	-0.46	0.00	-0.08	-0.04	-0.00	-0.04	-0.02
gzip	-0.23	-0.64	-0.50	0.00	-0.07	-0.03	-0.00	-0.05	-0.03
imagemagick	-0.05	-0.28	-0.18	0.01	-0.22	-0.12	-0.00	-0.05	-0.02
lighttpd	-0.18	-0.53	-0.40	0.00	-0.08	-0.04	-0.00	-0.06	-0.03
make	-0.16	-0.51	-0.38	0.02	-0.11	-0.05	-0.00	-0.05	-0.03
memcached	-0.13	-0.45	-0.32	0.03	-0.11	-0.05	-0.00	-0.06	-0.03
mkdir	-0.26	-0.63	-0.50	0.00	-0.09	-0.04	-0.00	-0.05	-0.02
nginx	-0.15	-0.48	-0.35	0.02	-0.11	-0.04	-0.00	-0.05	-0.03
nmap	-0.10	-0.36	-0.25	0.01	-0.13	-0.07	-0.00	-0.09	-0.05
objdump	-0.18	-0.56	-0.42	0.02	-0.11	-0.05	-0.00	-0.04	-0.02
poppler	-0.11	-0.32	-0.23	0.01	-0.18	-0.10	-0.00	-0.10	-0.05
rm	-0.24	-0.60	-0.48	0.00	-0.09	-0.04	-0.00	-0.05	-0.02
sort	-0.15	-0.47	-0.35	0.00	-0.10	-0.05	-0.00	-0.05	-0.02
tar	-0.15	-0.55	-0.41	0.01	-0.10	-0.05	-0.00	-0.04	-0.02
uniq	-0.21	-0.56	-0.44	0.00	-0.09	-0.05	-0.00	-0.04	-0.02
wget	N/A	N/A	N/A	0.01	-0.20	-0.11	-0.00	-0.15	-0.08
ml_train	N/A	N/A	N/A	0.06	-0.43	-0.27	-0.01	-0.11	-0.06

F. Analysis of Missed Functions

In this section, we analyze the reasons why different tracers fail to detect certain function calls. To facilitate this analysis, we categorize missed functions based on three key attributes: symbol type, symbol binding, and whether the function is part of a shared library's initialization routine. We consider two primary symbol types: (1) FUNC, representing standard, directly callable functions; (2) IFUNC, representing indirect functions that are dynamically resolved during shared library loading. For binding types, we distinguish three categories: (1) GLOBAL, functions visible across all shared libraries; (2) LOCAL, functions with visibility limited to the defining library; (3) WEAK, functions that can be overridden by other definitions. Finally, we also categorize functions based on whether they are initialization functions, which are called after a shared library is loaded but before the application's main() function is executed.

Table VIII presents the ratio of missed functions across these categories for each tracer. All the four compared tracers exhibit significantly higher miss rates for IFUNC functions compared to standard FUNC functions. They also show a higher miss rate for LOCAL functions. This is primarily due to the fact that most LOCAL symbols are stripped from shared libraries. Without symbol information or function boundary detection, these tracers cannot trace such functions. ltrace, drltrace and ldaudit cannot detect shared library initialization functions. One reason is that they begin tracing only after the main program starts—missing early initialization functions, as with ltrace. Another reason is that symbol information of these initialization functions are often stripped from shared libraries, making them undetectable to tools like drltrace and ldaudit. While IntelPT captures some of these initialization functions, which is likely because they are called later by some other functions, it still fails to detect many of them. RTrace shows a significantly lower miss rate across all categories and all the missed functions are caused by incorrect boundary detection.

TABLE VIII: Percentage of missed functions for different functions categories. wget and ml_train are excluded from this analysis as ltrace failed to trace them.

Category	**Value**	ltrace	drltrace	ldaudit	IntelPT	RTrace
Type	FUNC	55%	49%	74%	48%	1%
	IFUNC	82%	69%	74%	89%	0
Bind	GLOBAL	25%	11%	57%	47%	0
	LOCAL	75%	70%	84%	57%	2%
	WEAK	40%	30%	53%	45%	0
Init.	No	58%	51%	74%	53%	1%
	Yes	100%	100%	100%	94%	0

G. Application of RTrace in Security Domain

In this section, we demonstrate two security applications of RTrace: the light mode for detecting malicious Python packages and the rich mode for analyzing compromised shared libraries.

1) Malicious Python Package Detection: Although RTrace is not specifically designed for malicious package detection, it can be effectively applied in this domain. We use the dataset from [26], which comprises 1,500 benign and 500 malicious Python packages, to perform a statistical analysis of the installation behaviors of these packages. Our focus is on install-time attacks, i.e., attacks that occur during the package installation phase. Some packages fail to install due to incompatible Python versions or operating systems, thus we only focus on the packages that are successfully installed. Among the 1500 benign packages and 500 malicious packages, 898 and 289 were successfully installed, respectively. Using the light mode of RTrace, we trace the installation process of these successfully installed packages.

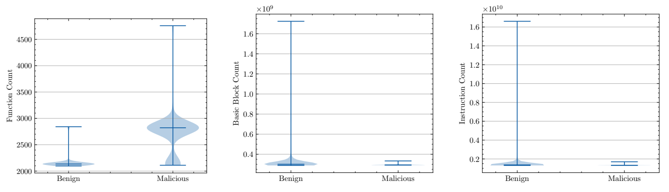
Figure 4a shows a violin plot of the number of functions executed during installation for each package. The distribution for benign packages is narrow and centered around 2,100 functions. In contrast, malicious packages exhibit a broader distribution.

While some have function counts similar to benign packages, the majority invoke significantly more functions—around 2,800. We found that 77% of the malicious packages executed more than 2,800 functions, while only 4% of benign packages do so. This suggests that function-level behavior during installation is a strong indicator of malicious activity, making it a promising dimension for detecting malicious packages.

Figures 4b and 4c present the distributions of basic block and instruction counts during installation. These metrics do not show a distribution as distinct as the function count. Interestingly, benign packages have longer-tailed distributions, suggesting more variability and complexity in their installation processes. In comparison, malicious packages display a more compact distribution. Both benign and malicious packages has a median around 3×10^8 basic blocks and 1.3×10^9 instructions, making it harder to differentiate between benign and malicious behavior using only these metrics.

We further investigate the reason behind the distinct distribution patterns observed in malicious versus benign packages—specifically, why malicious packages exhibit a long tail in function count but a compact distribution in basic block and instruction counts, while benign packages show the opposite trend. A recurring pattern among malicious packages is the use of custom installation scripts that override standard routines. These install-time attack scripts typically follow a fixed structure: they replace the default installation logic and frequently use the Python `requests` library to download a payload from a remote URL, followed by conditional checks to decide whether to execute the downloaded file based on the runtime environment. At the time of our experiments, many of the remote resources used by malicious packages were no longer accessible, preventing the payload from being downloaded or executed. As a result, the installation process of malicious packages was straightforward, involving fewer instructions and basic blocks. However, the initial stages of these attacks still invoke functions from uncommon shared libraries, such as `libnss_dns`, `libresolv`, and `libnss_files`, which are rarely used by benign packages. This leads to a broader variety of function call. In contrast, some benign installations involve building from C code or setup processes that include repetitive, same function calls. These activities result in more executed basic blocks and instructions, even if the number of distinct functions invoked remains lower. This difference highlights the potential of using function-level execution patterns as a lightweight and effective signal for detecting suspicious installation behaviors.

2) XZ Backdoor Case Study: We use `RTrace` to analyze a case in which the shared library itself is compromised, demonstrating how the two modes of `RTrace` can detect and analyze abnormal behaviors introduced through malicious modifications. We use the recently disclosed supply chain attack, the XZ backdoor vulnerability (CVE-2024-3094), as a case study. This vulnerability affects the widely used XZ compression library, which is pre-installed in many major Linux distributions. To avoid the risk of running a compromised library, we conduct this case study in a sandboxed AWS



(a) #Function. (b) #Basic block. (c) #Instruction.

Fig. 4: Distribution of executed function counts(detected by the light mode of `RTrace`), basic block counts and instruction count during installation of benign and malicious packages.

EC2 instance. As the instance lacked hardware support, the `IntelPT` tracer was excluded from this experiment.

Attack Mechanism. The attack was carried out by a malicious actor who gained control of the XZ project [57] and injected backdoor code into the shared library `liblzma.so`, specifically in versions 5.6.0 and 5.6.1. This library is a dependency of the `sshd` server binary. When the SSH server is launched, it loads the malicious version of `liblzma.so`. The library exploits the indirect function call mechanism to replace an internal OpenSSH function with a malicious version. As a result, the attacker is able to authenticate using a private key and gain unauthorized access to the system through the SSH server.

Analysis Setup. To evaluate the ability of various tracers to detect abnormal function behavior, we launch an SSH server with several versions of `liblzma.so` respectively, including nine older benign versions and the compromised versions (5.6.0*). We then use the tracers to trace the function calls made within the `liblzma.so` library during the server's launch.

Figure 5 shows the number of function calls detected by compared tracers and the light mode of `RTrace` in each version of the library. `ltrace`, `ldaudit` and `drltrace` fail to detect any function calls within the library across all versions, including the compromised version. `RTrace` consistently detects more function calls than all other tracers for every version of the library. Notably, `RTrace` constantly detect 7 function calls in the benign versions of the library. However, the compromised version exhibited a sharp increase, with 90 detected function calls. Given the fact that `liblzma.so` is a mature library with stable APIs, this significant increase in the number of function calls should be considered an anomaly signal and a potential red flag during security audits.

We further leverage the rich mode of `RTrace` to perform a detailed analysis of function call behavior. Figures 6a and 6b illustrate the dynamic function call graphs generated by the rich mode `RTrace` for the benign (5.6.0) and compromised (5.6.0*) versions of `liblzma.so`, respectively. In the benign version, the graph shows only standard function calls associated with shared library loading and unloading, such as `_init` and `_fini` functions. These are expected and represent normal, benign initialization behavior. In contrast, the function call graph for the compromised version reveals a stark different pattern. To improve readability, we present a simplified view

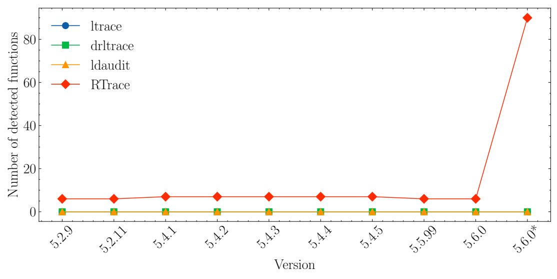
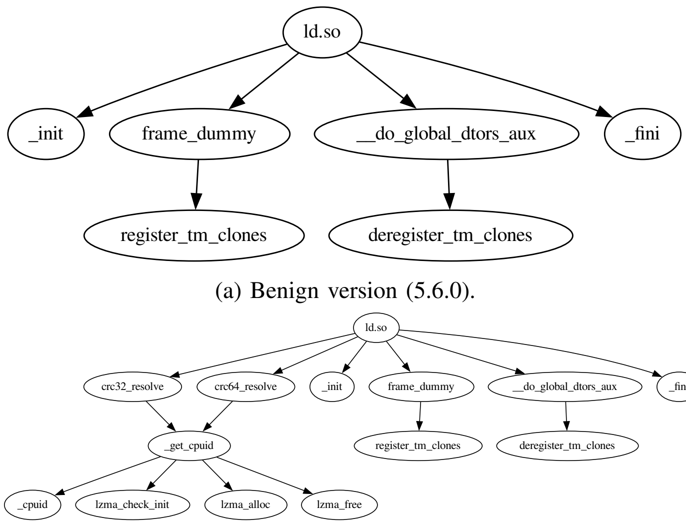


Fig. 5: Number of function calls detected by different tracers in the `liblzma.so` across different versions. 5.6.0* is the compromised version.

of the graph due to its significantly larger size. A key anomaly is observed: two indirect functions, `crc32_resolve` and `crc64_resolve`, are invoked before the standard `_init` function. These two functions serve as hidden entry points for the backdoor code, initiating setup and injection routines. All subsequent function calls in the graph are associated with the injected malicious payload. The full version of the compromised call graph is substantially more complex and contains 90 distinct function invocations in the library and differ significantly from all the previous benign versions.



(a) Benign version (5.6.0).

(b) Compromised version (5.6.0.*).

Fig. 6: Function call graph generated by the rich mode of `RTrace` for benign and compromised versions of `liblzma.so`.

VI. Discussion and Future Work

Modern software systems have become increasingly difficult to trace at runtime. This is primarily due to two factors: (1) they rely on large and complex dependencies, and (2) the implicit features that are employed to enhance performance and developer convenience, such as indirect function calls, shared library constructors and destructors, compiler optimizations, and so on. These mechanisms, while improving performance and developer productivity, often introduce implicit code executions that do not only occur outside the visibility of application developers but also negatively impact the effectiveness of existing library call tracing tools. As a result, these implicit behaviors can serve as stealthy vectors for malicious activity, making them attractive targets for attackers seeking to conceal malicious behavior. Therefore, an accurate and comprehensive library function call tracer is of great importance for software supply chain security. In this paper, we identify the limitations of existing library call tracers and propose a new tracer, `RTrace`, that addresses these limitations and traces shared library function calls more accurately and comprehensively.

Limitations. The light mode of `RTrace` depends on accurate function boundary detection to identify executed functions. While existing techniques have demonstrated good accuracy, they still face limitations. Some tools require substantial time to analyze large shared libraries, and others may fail to detect function boundaries reliably for certain binaries. Similarly, the rich mode of `RTrace` attempts to infer function signatures automatically using static binary analysis frameworks. These frameworks can be time-consuming, particularly for large libraries, and may produce incorrect function signatures. To mitigate this issue, we have designed `RTrace` to be extensible, allowing users to provide function signature manually if accessible. The rich mode currently supports only System V AMD64 ABI, a limitation shared with existing tools such as `ltrace` and `drltrace`. Overall, most limitations stem from the reliance on static analysis.

Future Work. Given that `RTrace` operates at runtime and has access to dynamic execution information, it is promising to explore how this runtime data can be leveraged to enhance the accuracy of both function boundary detection and signature inference. Integrating such dynamic insights into `RTrace` could potentially improve its robustness and precision. For example, we observed that FunSeeker can produce false positives in function boundary detection due to indirect returns; however, these returns can be captured at runtime and used to refine the detected boundaries. Additionally, `RTrace` does not yet support tracing `float` types, which we plan to address in future work.

VII. Conclusion

In this paper, we presented `RTrace`, a tracer designed to accurately capture function calls within shared libraries. Through a systematic evaluation of existing library function tracers, we found that their effectiveness and scalability is limited. Many shared library function calls are missed and these tools do not work for complex workloads, such as machine learning workloads. We analyzed the root causes of these limitations and identified key pitfalls in existing tracers, including reliance on incomplete symbol information, inability to detect early or non-standard function calls, and poor support for complex runtime behaviors. Building on this analysis, we designed `RTrace` to address these shortcomings through comprehensive runtime monitoring, function boundary detection, and support for unconventional function calls. Our evaluation on 21 real-world applications shows that RTrace significantly improves the accuracy of shared library function call tracing, achieving an F1-score of at least 92% across all benchmarks. We also show how RTrace can be used to assist in software security analysis using real-world examples.

Acknowledgment

We would like to thank Prof. Philipp Leitner for his valuable suggestions that inspired this work. We are also grateful to the anonymous reviewers for their valuable comments, which greatly helped improve the paper. This project is supported by the Knut and Alice Wallenberg Foundation via a Wallenberg AI, Autonomous Systems and Software Program PhD grant and an SSF future research leaders grant.

References

[1] Gartner, "Application development will shift to application assembly and integration." https://www.gartner.com/en/newsroom/press-releases/2021-11-10-gartner-says-cloud-will-be-the-centerpiece-of-new-digital-experiences, 2021, [Accessed 08-04-2025].

[2] Y. Shen, X. Gao, H. Sun, and Y. Guo, "Understanding vulnerabilities in software supply chains," *Empirical Software Engineering*, vol. 30, no. 1, pp. 1–38, 2025.

[3] W. Enck and L. Williams, "Top five challenges in software supply chain security: Observations from 30 industry and government organizations," *IEEE Security & Privacy*, vol. 20, no. 2, pp. 96–100, 2022.

[4] L. Gao, M. Lu, L. Li, and C. Pan, "A survey of software runtime monitoring," in *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. IEEE, 2017, pp. 308–313.

[5] G. Rousseau, R. Di Cosmo, and S. Zacchiroli, "Software provenance tracking at the scale of public source code," *Empirical Software Engineering*, vol. 25, pp. 2930–2959, 2020.

[6] S. Grayson, F. Aguilar, R. Milewicz, D. S. Katz, and D. Marinov, "A benchmark suite and performance analysis of user-space provenance collectors," in *Proceedings of the 2nd ACM Conference on Reproducibility and Replicability*, 2024, pp. 85–95.

[7] F. Wang, Y. Kwon, S. Ma, X. Zhang, and D. Xu, "Lprov: Practical library-aware provenance tracing," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 605–617.

[8] L. Documentation, "Shared Libraries — tldp.org," https://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html, [Accessed 10-04-2025].

[9] H. Zhang and A. Ali-Eldin, "The hidden bloat in machine learning systems," 2025. [Online]. Available: https://arxiv.org/abs/2503.14226

[10] "ltrace — ltrace.org," https://ltrace.org/, [Accessed 08-10-2024].

[11] "GitHub - mxmssh/drltrace: Drltrace is a library calls tracer for Windows and Linux applications. — github.com," https://github.com/mxmssh/drltrace, [Accessed 28-08-2024].

[12] "rtld-audit(7) - Linux manual page — man7.org," https://man7.org/linux/man-pages/man7/rtld-audit.7.html, [Accessed 08-04-2025].

[13] A. Ziegler, J. Geus, B. Heinloth, T. Hönig, and D. Lohmann, "Honey, i shrunk the elfs: Lightweight binary tailoring of shared libraries," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–23, 2019.

[14] "NVD - CVE-2024-3094 — nvd.nist.gov," https://nvd.nist.gov/vuln/detail/CVE-2024-3094, [Accessed 26-09-2024].

[15] D. Bruening and S. Amarasinghe, "Efficient, transparent, and comprehensive runtime code manipulation," 2004.

[16] L. Williams, G. Benedetti, S. Hamer, R. Paramitha, I. Rahman, M. Tamanna, G. Tystahl, N. Zahan, P. Morrison, Y. Acar *et al.*, "Research directions in software supply chain security," *ACM Transactions on Software Engineering and Methodology*, 2024.

[17] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*. Springer, 2020, pp. 23–43.

[18] B. Hammi and S. Zeadally, "Software supply-chain security: Issues and countermeasures," *Computer*, vol. 56, no. 7, pp. 54–66, 2023.

[19] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Sok: Taxonomy of attacks on open-source software supply chains," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1509–1526.

[20] R. Alkhadra, J. Abuzaid, M. AlShammari, and N. Mohammad, "Solar winds hack: In-depth analysis and countermeasures," in *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. IEEE, 2021, pp. 1–7.

[21] "CVE-2014-0160." Available from NVD, CVE-ID CVE-2021-44228., December 2021, [Accessed 10-04-2025]. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160

[22] W. Liang, X. Ling, J. Wu, T. Luo, and Y. Wu, "A needle is an outlier in a haystack: hunting malicious pypi packages with code clustering," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 307–318.

[23] C. Huang, N. Wang, Z. Wang, S. Sun, L. Li, J. Chen, Q. Zhao, J. Han, Z. Yang, and L. Shi, "{DONAPI}: Malicious {NPM} packages detector using behavior sequence knowledge mapping," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 3765–3782.

[24] J. Zhang, K. Huang, Y. Huang, B. Chen, R. Wang, C. Wang, and X. Peng, "Killing two birds with one stone: Malicious package detection in npm and pypi using a single model of malicious behavior sequence," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 4, pp. 1–28, 2025.

[25] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *Network and Distributed System Security (NDSS) Symposium*, 2021.

[26] X. Zheng, C. Wei, S. Wang, Y. Zhao, P. Gao, Y. Zhang, K. Wang, and H. Wang, "Towards robust detection of open source software supply chain poisoning attacks in industry environments," in *Proceedings of the 39th IEEE/ACM international conference on automated software engineering*, 2024, pp. 1990–2001.

[27] "elf(5) - Linux manual page — man7.org," https://man7.org/linux/man-pages/man5/elf.5.html, [Accessed 20-04-2025].

[28] J. Engblom, "A review of reverse debugging," in *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*. IEEE, 2012, pp. 1–6.

[29] J. Wang, N. Liu, A. Casadevall-Saiz, Y. Liu, D. Behrens, M. Fu, N. Jia, H. Härtig, and H. Chen, "Enabling efficient mobile tracing with btrace," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 325–338.

[30] M. D. Brown, A. Meily, B. Fairservice, A. Sood, J. Dorn, E. Kilmer, and R. Eytchison, "A broad comparative evaluation of software debloating tools," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 3927–3943.

[31] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, "{RAZOR}: A framework for post-deployment software debloating," in *28th USENIX security symposium (USENIX Security 19)*, 2019, pp. 1733–1750.

[32] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[33] "Tiny tracer: A pin tool for tracing api calls," https://github.com/hasherezade/tiny_tracer, [Accessed 16-09-2024].

[34] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3: System Programming Guide*, vol. 2, no. 11, 2016.

[35] J. Alves-Foss and J. Song, "Function boundary detection in stripped binaries," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 84–96.

[36] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 177–189.

[37] A. Di Federico, M. Payer, and G. Agosta, "rev. ng: a unified binary analysis framework to recover cfgs and function boundaries," in *Proceedings of the 26th International Conference on Compiler Construction*, 2017, pp. 131–141.

[38] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," 2016.

[39] X. Meng and B. P. Miller, "Binary code is not easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 24–35.

[40] "GitHub - NationalSecurityAgency/ghidra: Ghidra is a software reverse engineering (SRE) framework — github.com," https://github.com/NationalSecurityAgency/ghidra, [Accessed 24-07-2025].

[41] H. Kim, J. Lee, S. Kim, S. Jung, and S. K. Cha, "How'd security benefit reverse engineers?: The implication of intel cet on function identification," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2022, pp. 559–566.

[42] K. Pei, J. Guan, D. Williams-King, J. Yang, and S. Jana, "Xda: Accurate, robust disassembly with transfer learning," *arXiv preprint arXiv:2010.00770*, 2020.

[43] S. Yu, Y. Qu, X. Hu, and H. Yin, "{DeepDi}: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2709–2725.

[44] S. Kim, H. Kim, and S. K. Cha, "Funprobe: Probing functions from binary code through probabilistic analysis," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1419–1430.

[45] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[46] C. Ye, Y. Cai, A. Zhou, H. Huang, H. Ling, and C. Zhang, "Manta: Hybrid-sensitive type inference toward type-assisted bug detection for stripped binaries," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, 2024, pp. 170–187.

[47] F. de Goër, R. Groz, and L. Mounier, "Lightweight heuristics to retrieve parameter associations from binaries," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, 2015, pp. 1–12.

[48] Z. Lin, J. Li, B. Li, H. Ma, D. Gao, and J. Ma, "Typesqueezer: When static recovery of function signatures for binary executables meets dynamic analysis," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2725–2739.

[49] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell, "System v application binary interface," *AMD64 Architecture Processor Supplement, Draft v0*, vol. 99, no. 2013, p. 57, 2013.

[50] "Guide: Function Calling Conventions — delorie.com," https://www.delorie.com/djgpp/doc/ug/asm/calling.html, [Accessed 29-08-2024].

[51] "Introducing eight new Amazon EC2 bare metal instances - AWS — aws.amazon.com," https://aws.amazon.com/about-aws/whats-new/2023/10/new-amazon-ec2-bare-metal-instances/, [Accessed 26-07-2025].

[52] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: debloating binary shared libraries," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 70–83.

[53] "PyTorch — pytorch.org," https://pytorch.org/, [Accessed 22-04-2025].

[54] "find-dbgsym-packages: Debian Manpages — manpages.debian.org," https://manpages.debian.org/testing/debian-goodies/find-dbgsym-packages.1.en.html, [Accessed 26-07-2025].

[55] "NVIDIA CUDA Toolkit Symbol Server — NVIDIA Technical Blog — developer.nvidia.com," https://developer.nvidia.com/blog/cuda-toolkit-symbol-server/, [Accessed 26-07-2025].

[56] "musl libc - libc-test — wiki.musl-libc.org," https://wiki.musl-libc.org/libc-test, [Accessed 20-07-2025].

[57] tukaani, "XZ Compression Library," https://github.com/tukaani-project/xz, [Accessed 14-04-2025].

VIII. Appendix

A. Sample Code

Listing 2: A single call to function `indirect_func` will trigger all the functions in the library.

```c
#include <stdio.h>
void export_func(){
  printf("export_func");
}
static void constructor() __attribute__((
   constructor));
void constructor(){
  export_func();
  printf("constructor");
}
int called_by_resolver(){
  printf("called_by_resolver");
}
void indirect_func() __attribute__((ifunc("
   resolve_indirect_func")));
static void indirect_func_impl(){
  printf("indirect_func_impl");
}
static void *resolve_indirect_func(void){
  called_by_resolver();
  return (void *)indirect_func_impl;
}
```

Listing 3: A sample output of library detection.

```
/usr/lib/ld.so: [0x717234fc0000, 0
   x717234ffa000)
/usr/lib/libc.so: [0x71723449c000, 0
   x7172346ae000)
/usr/lib/libdl.so: [0x717234e8b000, 0
   x717234e90000)
```

Listing 4: A sample output branch-return instrumentation.

```
0x721537f4225b
0x721537f42275
...
```

Listing 5: A sample output of adaptive boundary detection of shared libray `ld.so`.

```
_dl_call_init: [0x1000, 0x1090)
_dl_call_fini: [0x1090, 0x1150)
...
```

Listing 6: A sample output of function instrumentation.

```
Entry: 129645018424432
Arg_0: 1
BB: 129645018424432
Entry: 129645017074208
BB: 129645017074208
Exit: 129645017074208
BB: 129645018424467
Ret: 0
Exit: 129645018424432
...
```

Listing 7: A sample output call graph report.

```
{
 "name": "root",
 "start_addr": "0x0",
 "so_path": "root",
 "num_args": 0,
 "args": [],
 "ret_val": null,
 "executed_blocks": 0,
 "executed_insts": 0,
 "calls": [
   {
     "name": "boundary_detected_0x201d0",
     "start_addr": "0x201d0",
      "so_path": "/usr/lib/x86_64-linux-gnu/ld-linux-x86
    -64.so.2",
      "num_args": 0,
      "args": [],
      "ret_val": 0,
      "executed_blocks": 160,
      "executed_insts": 636,
      "calls": [
        {
          "name": "boundary_detected_0x1d1f0",
          "start_addr": "0x1d1f0",
          "so_path": "/usr/lib/x86_64-linux-gnu/ld-linux-
    x86-64.so.2",
          "num_args": 1,
          "args": [129644961628184],
          "ret_val": 0,
          "executed_blocks": 1,
          "executed_insts": 10,
          "calls": []
        }
        ...
      ]
    },
    ...
  ]
}
```

B. Commands to Run Evaluated Tracers

Listing 8: Command lines used to run evaluated tracers.

```
# ltrace
ltrace  -f -o output.log -s 1024 -x "*" {cmd}

# drltrace
drltrace  -logdir output {cmd}

# ldaudit
LD_AUDIT=/path/to/audit.so {cmd}

# IntelPT
perf record -e intel_pt//u  -o {log_dir}/perf.
    data -- {cmd} # record
perf script -i {log_dir}/perf.data -F  ip,sym,
    dso  --no-demangle  > {log_dir}/output.txt
     # parse
```

C. Examples of Ablation Study

Boundary Detection. In the imagemagick benchmark, the function decompress_onepass located at address 0x21630 in the libjpeg library was executed during runtime. This function's symbol is absent from the symbol table as it is stripped from the library. However, with boundary detection enabled, RTrace successfully identified the function boundary and correctly reported it as executed. When boundary detection is disabled, the function boundary for decompress_onepass is not identified, and the function is consequently not reported, resulting in a false negative. Moreover, due to the presence of executed instructions, RTrace incorrectly attributes this activity to the closest known function before 0x21630, namely jpeg_write_coefficients at 0x214c0. Since this function was not actually executed, this also leads to a false positive. This example highlights a critical insight: missing a single function boundary can simultaneously lead to both false positives and false negatives. Thus, boundary detection directly influences both precision and recall.

Branch Instrumentation. Certain functions do not exit with a RET instruction but instead exiting by branching directly to another function using a jump instruction. In the imagemagick benchmark, the function AnnotateComponentTerminus at address 0x62220 in the libMagickCore shared library is such a case. This function executes without exiting via a standard RET; instead, it performs a jump to another function. When branch instrumentation is disabled, such control flow transitions are not captured, and AnnotateComponentTerminus is not reported as executed, resulting in a false negative. Listing 9 shows the corresponding assembly code for this function, where no return instruction in the function.

Listing 9: Assemble code of function AnnotateComponentTerminus.

```
62220:  endbr64
62224:  cmpq
6222b:
6222c:  je      62240
6222e:  lea     0x41d653(%rip),%rdi
62235:  jmp     58de0
6223a:  nopw    0x0(%rax,%rax,1)
62240:  lea     0x41d641(%rip),%rdi
62247:  sub     $0x8,%rsp
6224b:  call    58ce0
62250:  lea     0x41d631(%rip),%rdi
62257:  add     $0x8,%rsp
6225b:  jmp     58de0
```

Return Instrumentation. A normal function typically exits with a RET instruction. Therefore, return instrumentation plays an important role in accurately identifying function execution. A function that does not have any branch instruction, but only RET instructions, can be only detected by return instrumentation. As the example shown in Listing 10, in the imagemagick benchmark, the function omp_get_max_threads, located at address 0x12290 in the shared library libgomp.so, is executed during runtime. However, this function contains no branch instructions—only a RET. As a result, when return instrumentation is disabled, RTrace fails to detect this function, resulting in a false negative.

Listing 10: Assemble code of function omp_get_max_threads.

```
12290:  endbr64
12294:  mov     0x42ce5(%rip),%rax
1229b:  mov     %fs:0x58(%rax),%rdx
122a0:  test    %rdx,%rdx
122a3:  lea     0xa0(%rdx),%rax
122aa:  lea     0x4318f(%rip),%rdx
122b1:  cmove   %rdx,%rax
122b5:  mov     (%rax),%eax
122b7:  ret
```