Adonis: Practical and Efficient Control Flow Recovery through OS-Level Traces

XUANZHE LIU, CHENGXU YANG, DING LI, YUHAN ZHOU, SHAOFEI LI, and JIALI CHEN, Peking University, China
ZHENPENG CHEN, University College London, United Kingdom

Control flow recovery is critical to promise the software quality, especially for large-scale software in production environment. However, the efficiency of most current control flow recovery techniques is compromised due to their runtime overheads along with deployment and development costs. To tackle this problem, we propose a novel solution, *Adonis*, which harnesses OS-level traces, such as dynamic library calls and system call traces, to efficiently and safely recover control flows in practice. *Adonis* operates in two steps: it first identifies the call-sites of trace entries, then it executes a pair-wise symbolic execution to recover valid execution paths. This technique has several advantages. First, *Adonis* does not require the insertion of any probes into existing applications, thereby minimizing *runtime cost*. Second, given that OS-level traces are hardware-independent, *Adonis* can be implemented across various hardware configurations without the need for hardware-specific engineering efforts, thus reducing *deployment cost*. Third, as *Adonis* is fully automated and does not depend on manually created logs, it circumvents additional *development cost*. We conducted an evaluation of *Adonis* on representative desktop applications and real-world IoT applications. *Adonis* can faithfully recover the control flow with ***86.8%*** recall and ***81.7%*** precision. Compared to the state-of-the-art log-based approach, *Adonis* can not only cover all the execution paths recovered, but also recover ***74.9%*** of statements that cannot be covered. In addition, the runtime cost of *Adonis* is ***18.3***× lower than the instrument-based approach; the analysis time and storage cost (indicative of the deployment cost) of *Adonis* is ***50***× smaller and ***443***× smaller than the hardware-based approach, respectively. To facilitate future replication and extension of this work, we have made the code and data publicly available.

CCS Concepts: • **Software and its engineering → Error handling and recovery**; *Software maintenance tools.*

Additional Key Words and Phrases: Control Flow Recovery, OS-level Traces, Reverse Engineering

1 INTRODUCTION

In a world where operations are increasingly dictated by software, the visibility and prominence of bugs leading to system failures have escalated, especially in production environments [10]. When software failures happen, the primary concern of developers is understanding the processes the program undergoes [35, 47]. To this end, detailed control flow is of great value for the later root cause analysis or fault localization. Realizing the value of control flow, researchers have proposed many solutions for recovering the control flow from software failures [4, 14, 21, 45, 47, 54–57, 60].

However, unfortunately existing control flow recovery techniques often fall short of adequately accommodating the unique cost constraints associated with large-scale software in production environments. In particular, three types of cost constraints are relevant: Firstly, *runtime cost* is a critical consideration, as the control flow recovery

Authors' addresses: Xuanzhe Liu, liuxuanzhe@pku.edu.cn; Chengxu Yang, yangchengxu@pku.edu.cn; Ding Li, ding_li@pku.edu.cn; Yuhan Zhou, zhouyuhan@pku.edu.cn; Shaofei Li, lishaofei@pku.edu.cn; Jiali Chen, chenjiali@pku.edu.cn, Peking University, Beijing, China; Zhenpeng Chen, zp.chen@ucl.ac.uk, University College London, London, United Kingdom.

1049-331X/2023/7-ART $15.00
https://doi.org/10.1145/3607187

	Instrument-based	Hardware-based	Log-based	*Adonis*
Runtime Cost	High	Low	Low	**Low**
Deployment Cost	Low	High	Low	**Low**
Development Cost	Low	Low	High	**Low**

Table 1. Comparison between *Adonis* and existing control flow recovery techniques.

technique should not substantially slow down the monitored software [8]. This is due to the stringent latency limits imposed by production environments, where any delays could lead to substantial costs. Secondly, *deployment cost* is essential, as the technique should be easily deployable on different hardware platforms. Given the wide variety of devices that software may run on [20], the technique must be versatile enough to work across a range of systems. Finally, *development cost* is a crucial factor, which refers to the manual effort required to make the technique work when the applied software gets updated. The control flow recovery technique should require minimal manual effort to keep up with the fast-evolving production software [10]. Specifically, the technique should be fully automated, eliminating the need for manual intervention. Unfortunately, existing techniques fail to balance all three types of costs effectively. Therefore, there is an urgent need for a novel technique capable of recovering control flow information in production environments, whilst satisfying the constraints related to runtime, deployment, and development costs.

Research Challenges. It is challenging to build a control flow recovery technique that achieves moderate runtime, deployment, and development cost. To the best of our knowledge, as shown in Table 1, traditional techniques used by existing control flow recovery solutions have high cost in production environments. The first type is the *instrument-based* techniques [4, 47], which insert probes at compile time and record the branch taken by control flow statements. This technique is unsuitable for production environments, as it can slow down the monitored program by over 50% [4], exceeding the acceptable runtime cost. The second type is *hardware-based* techniques, which leverage hardware features, such as Intel-PT, to record branch outcomes [14, 21, 45, 54–56]. These techniques do not meet the requirements of deployment cost because they cannot work when Intel-PT is unavailable (e.g., in Cloud or IoT environments) and have considerable cost for storing and processing control flow data. The third type is *log-based* techniques that leverage logs manually created by developers [12, 28, 59, 61]. However, such techniques require software developers to manually improve the quality of logs that can recover control flow accurately. Moreover, the cost can be extremely high to maintain logging code along with the update of the software. Once the log misses key information of program failures, developers have to add much more logging code to try to catch the failures. In summary, new control flow recovery techniques are needed to meet the requirements of production environments.

Design Principles. This paper aims to propose a control flow recovery technique that satisfies runtime, deployment, and development cost constraints in production environments. Our approach achieves this by following three design principles: (1) Firstly, our technique is *instrumentation-free*, meaning it does not add any extra code to the monitored program, thereby minimizing the runtime cost. (2) Secondly, our technique is *hardware-independent*, meaning it does not rely on any specific hardware features. Therefore, it can be applied in different environments without any modifications, satisfying deployment cost constraints. (3) Finally, our proposed technique is *fully automated*, requiring no manual efforts. This satisfies the requirement of development cost, as software developers do not need to expend additional effort manually improving the quality of logs or maintaining logging code. By meeting all three requirements, our proposed technique offers a promising solution for recovering control flow in production environments while satisfying runtime, deployment, and development cost constraints.

Insights. Our technique is motivated by an insight that has been overlooked by existing methods: OS-level traces, such as dynamic library calls and system call traces, have a strong correlation to the control flow of a program. Applications heavily rely on the OS interface [40], including system calls and shared libraries like glibc. Therefore, we can recover the control flow of programs by linking the call sites of system calls and shared library calls. Importantly, modern operating systems can automatically collect OS-level traces with less than 5% runtime overhead [9]. Furthermore, collecting OS-level traces does not require any instrumentation or hardware features. Therefore, our approach of recovering control flow from OS-level traces meets the design requirements of being instrumentation-free, hardware-independent, and fully automated.

Technical Challenges. Leveraging OS-level traces to recover the control flow is a promising approach, but it poses two non-trivial challenges: **(1) Statement-level Ambiguity**. It is difficult to identify the call site of each trace entry inside OS-level traces. Programs invoke the same library functions or system calls at different locations, making it challenging to match items in OS-level traces to their exact call sites. Conventional log-based techniques, such as Sherlog [57], match a log message to its log printing statement through format string matching (e.g., log `"file ABC"` is matched to the format string `"file %s"`). However, this matching-based method cannot solve statement-level ambiguity because not all library functions and system calls have distinguishable arguments like the "format strings." **(2) Path-level Ambiguity**. Even though we can identify the call site of each trace entry, recovering the full path is still not straightforward. It is notable that dynamic functions or system calls are sparse in a program, and there may be code snippets that contain no library functions, as a result, simply using global symbolic execution can lead to path explosion, while previous summary-based approaches [57] or context-insensitive approaches [47] may result in inaccurate results.

Design. In this paper, we propose *Adonis*, a novel and practical approach that leverages OS-level traces to recover the control flow of a program's execution. Specifically, given a program whose *source code may or may not be accessible*, *Adonis* non-intrusively traces the program by recording its library functions or system calls and performs a two-stage analysis and outputs the control flow. To handle the challenges of statement-level and path-level ambiguity, *Adonis* uses several techniques. (1) To handle the statement-level ambiguity, *Adonis* leverages both internal information, such as the "format string" argument, and external order information between entries to identify each entry's call-site. We also propose a control flow graph simplifying approach that masks basic blocks that will not generate OS-level traces, significantly narrowing the search space. (2) To handle path-level ambiguity, *Adonis* treats each identified call-site as a checkpoint and performs a pairwise symbolic execution between pairs of checkpoints. For each pair, if certain variables are captured by the trace, *Adonis* looks back to paths between previous pairs and excludes impossible ones. Finally, *Adonis* concatenates partial paths between checkpoints as the output.

Evaluation. We have implemented the prototype of *Adonis* based on Linux. To show its effectiveness in control flow recovery, we use eleven applications, of which eight are desktop applications covered by previous work [47, 48, 56] and three are real-world IoT applications (running on a Raspberry Pi). We evaluate the accuracy, runtime cost, deployment cost, and development cost of *Adonis*. **Accuracy:** We found that *Adonis* is able to accurately recover control flow with 86.8% recall and 81.7% precision. Compared to the state-of-the-art log-based baseline, *Adonis* significantly reduces false positives by 41.7% and recovers 74.9% of statements that were missed by the baseline. **Runtime Cost:** The runtime overhead (cost for tracing) of *Adonis* is between 2.78% to 3.34%, which is lower than hardware-based techniques (6.58%). In particular, the runtime cost of *Adonis* is significantly lower (18.3× lower) than instrument-based techniques. **Deployment Cost:** *Adonis* can be deployed in desktop, IoT, and cloud environments without modifications. Compared to hardware-based techniques, *Adonis* has a much lower storage cost (443× lower) and trace processing time (50× shorter). **Development Cost:** Unlike log-based techniques, *Adonis* is fully automated and does not require manual effort when the monitored program upgrades. Specifically, *Adonis* requires 18.4× fewer log printing statements than log-based techniques.

We summarize our contributions as follows:

- We propose *Adonis*, a novel control flow recovery approach based on OS-level traces, following the design principles of instrumentation-free, hardware-independent, and fully-automated.
- We implement the prototype of *Adonis* and evaluate it on representative desktop applications and IoT applications. The results demonstrated its effectiveness in terms of low runtime cost, deployment cost, and development cost, compared to SOTA approaches.
- We make available the scripts and data used in this study[1] to the research community for other researchers to adopt Adonis or replicate and extend this work.

The rest of the paper is organized as follows: Section 2 proposes a motivating example to show the limitations of existing techniques and the insights of *Adonis*. Section 3 introduces the overview of our approach, including the input, output and the workflow of *Adonis*. Section 4 introduces the detailed design of *Adonis*. Section 5 introduces how we implemented *Adonis* and the optimization we have applied. Section 6 provides our evaluation on *Adonis* compared to existing techniques. Section 7 introduces the related work. Section 8 discusses the threats to validity and limitations of this work. Section 9 summarizes this paper.

2 A MOTIVATING EXAMPLE

In this section, we use a real-world failure example to explain the limitations of existing techniques and the insight of our approach. Figure 1 shows our example, which is from abc2mtex, a popular Linux tool to notate tunes. Here is how it works – first, it parses the `settings` and sets up the program (Lines 27 - 30). Then, it tries to load the filename of the input (Lines 32 - 42). After that, it opens the file to read the inputs (Lines 8 - 23 and Lines 44 - 45). Finally, it processes the inputs (Lines 46 - 47). Besides the functional code, developers also write some log printing statements (Lines 18, 37, and 45) to help diagnose failures.

These code snippets contain a stack overflow bug recorded as EDB-47254. The buggy code is at Line 14, where it copies `filename` to a temporary variable defined at the beginning of the function. A stack overflow bug happens when the length of `filename` exceeds the capacity of the temporary variable, and the program will crash when it returns (Line 22).

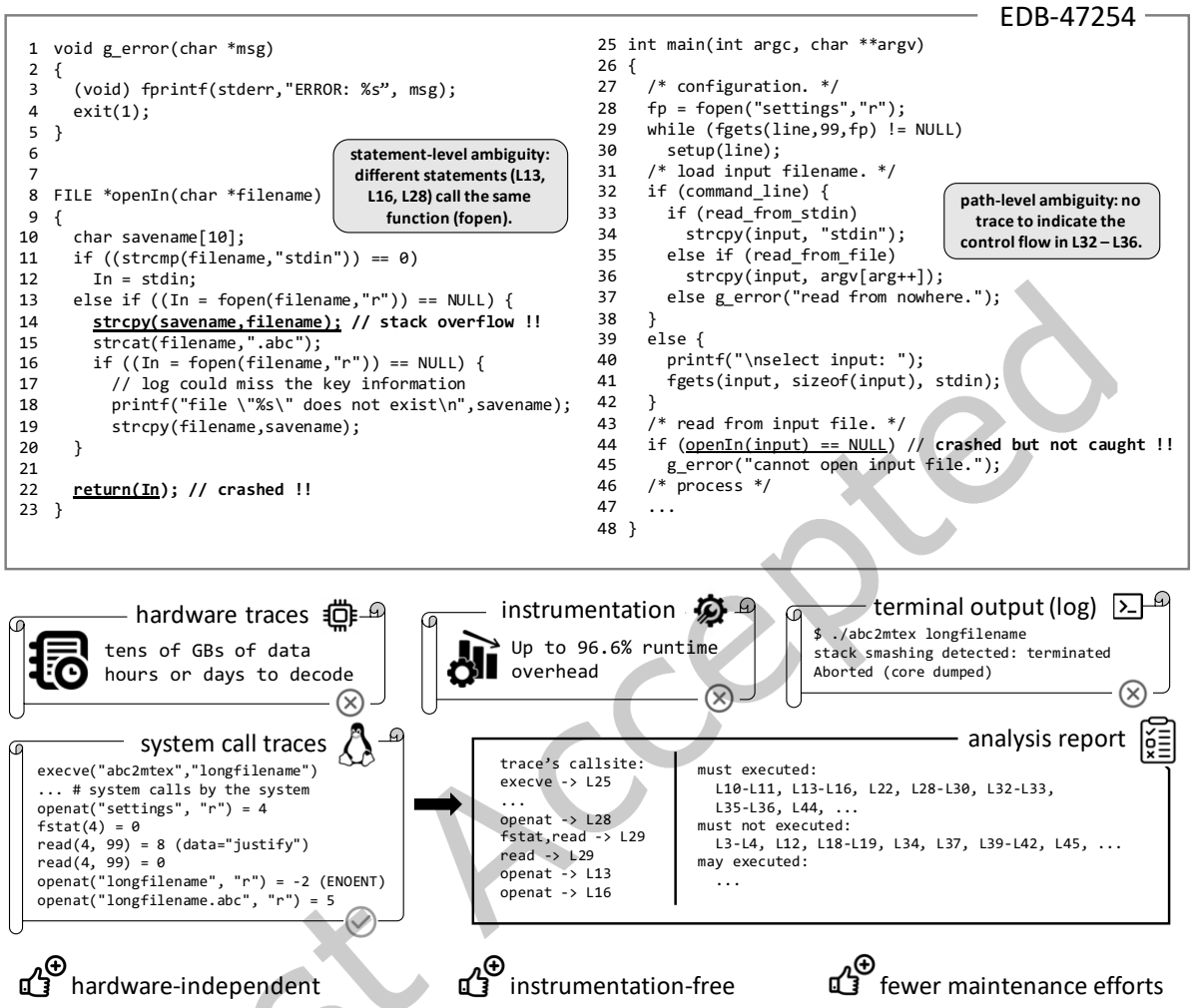
Control flow recovery in production: When the program failure happens in production, what presents to developers is usually a partial picture of the failed execution. Specifically, unlike that in a development environment where developers can use a debugger to run the program step by step and inspect variables, in production environments, most failure reports presented to developers contain only a stack trace and a few logs before failure happens [47]. It would be rather challenging and time-consuming for developers to find the bug based on these briefly described reports.

To this end, control flow information has great value in debugging failures in production environments. In our example, the control flow information can reveal the branch decisions at Lines 11, 13, and 16 and can also reveal that the program abnormally exits in function `openIn`. Developers can follow the execution paths that lead to the failure and quickly identify the buggy code.

Limitations of existing techniques: Despite the value of control flow information in debugging, recovering the control flow in production environments is non-trivial considering the runtime, deployment, and development cost. Currently, there are three categories of control flow recovery techniques, i.e., instrument-based techniques, log-based techniques, and hardware-based techniques.

(1) *Instrument-based techniques* [4, 47, 48] recover control flow by inserting probes at compile time to record branch conditions [4, 47]. However, these techniques suffer the problem of *high runtime cost*. In our example, a full instrumentation to profile executed paths would slow down the program by 47.6% on average (up to 96.6%), which is typically unaffordable in production environments [19].

[1] https://github.com/PKU-Chengxu/Adonis




```
1  void g_error(char *msg)
2  {
3    (void) fprintf(stderr,"ERROR: %s", msg);
4    exit(1);
5  }
6
7
8  FILE *openIn(char *filename)
9  {
10   char savename[10];
11   if ((strcmp(filename,"stdin")) == 0)
12     In = stdin;
13   else if ((In = fopen(filename,"r")) == NULL) {
14     strcpy(savename,filename); // stack overflow !!
15     strcat(filename,".abc");
16     if ((In = fopen(filename,"r")) == NULL) {
17       // log could miss the key information
18       printf("file \"%s\" does not exist\n",savename);
19       strcpy(filename,savename);
20   }
21
22   return(In); // crashed !!
23 }
```

```
25 int main(int argc, char **argv)
26 {
27   /* configuration. */
28   fp = fopen("settings","r");
29   while (fgets(line,99,fp) != NULL)
30     setup(line);
31   /* load input filename. */
32   if (command_line) {
33     if (read_from_stdin)
34       strcpy(input, "stdin");
35     else if (read_from_file)
36       strcpy(input, argv[arg++]);
37     else g_error("read from nowhere.");
38   }
39   else {
40     printf("\nselect input: ");
41     fgets(input, sizeof(input), stdin);
42   }
43   /* read from input file. */
44   if (openIn(input) == NULL) // crashed but not caught !!
45     g_error("cannot open input file.");
46   /* process */
47   ...
48 }
```


Fig. 1. A motivating example that shows how OS-level traces help find a real bug.

(2) *Log-based methods* [57] are limited due to the high development cost. We argue that the cost comes from two parts. The first part is the cost of maintaining logging code – with the rapid evolution of modern software, developers are required to frequently maintain existing log printing statements and add new ones for new features. According to previous studies [12, 28], maintaining logging code has proven to be error-prone and time-consuming. The second part is the cost of adding temporary logging code – previous studies on logging practices [36, 58] show that around 60% of failures do not leave any trace in logs, and developers have to spend a significant amount of effort on adding temporary logs to narrow the space in finding bugs. As shown in our example, the program bypasses two important log points (Line 18 and Line 45) and crashes without leaving any logs. For the log point at Line 18, it is triggered only when the program fails to open the input file twice. In other words, if the first `fopen` fails and the second `fopen` succeeds, the log point at Line 18 is bypassed. For the log

point at Line 45, it still fails to catch this failure because the call stack is smashed and no longer available in function `openIn`. As a result, developers have to add lots of temporary logs to narrow the search space, which significantly increases the development cost.

(3) *Hardware-based methods* [14, 54, 55] are not suitable due to the heavy deployment cost. On the one hand, hardware-based methods require specific hardware and heavy software support. For example, existing work [14, 54, 55] is typically based on Intel-PT. This hinders these methods from being widely deployed, given the popularity of nowadays VM-based cloud computing schema [7]. To date, major cloud computing platforms (including AWS, Google Cloud, Microsoft Azure, and Ali Cloud) do not support Intel-PT in their services due to the heavy implementation efforts. Besides, IoT devices also do not support hardware-based control flow recovery[2]. On the other hand, even for the platforms that support Intel-PT, hardware-based techniques are still not friendly to developers because the output traces would take massive storage (tens of GBs) and hours or days of time to decode [43].

Our insight: We argue that OS-level traces are promising for recovering crash paths in production environments. Applications need to constantly call external shared libraries and system calls to accomplish different tasks. Therefore, the call sites of shared libraries and system calls are natural probes for recovering the execution paths of an application.

Compared with existing techniques, OS-level traces also have moderate runtime, deployment, and development cost. For *runtime cost*, since system calls and shared libraries are more complex than single instructions, the relative overhead for logging OS-level traces is much lower than conventional instrument-based techniques, which log branch statements. In our example, monitoring OS-level traces slows down the program by only 2.78~3.34%, which is an order of magnitude lower than existing instrument-based techniques. For *deployment cost*, OS-level traces are independent of hardware features. Furthermore, modern OSes have provided full-fledged tools or utilities [9, 23, 42, 44] to collect OS-level traces. For *development cost*, OS-level traces can adapt to software evolution because they are inherently embedded in the program. Specifically, OS-level traces are able to cover not only the information recorded by logs (e.g., the log printing statements at Lines 3, 18, and 40) but also the information missed by logs (e.g., the file open operations at Line 13, 16, and 28).

Bug localization based on OS-level traces: We show how developers can leverage OS-level traces to locate the bug. Given the monitored system calls of the execution in Figure 1, developers can notice two "`openat`" events by reversely checking the monitored trace. These two entries represent two open operations: one failed, and another succeeded, and both of them try to open a file with a long filename. Developers can relate events to their call sites (Lines 13 and 16) by checking all open-related functions. Then, developers can quickly notice the abnormal `longfilename` recorded by the trace and find the stack overflow bug in Line 14. Moreover, based on the trace, developers can recover the path before the program crashes (shown as "`must executed`" and "`must not executed`", which is also used by existing work [57] to make the recovered control flow more readable.) and identify that the abnormal `longfilename` is passed in at Line 36 (read from user's command line).

Challenges: Despite the advantages of OS-level traces, leveraging them to recover the control flow faces two challenges: **(1) Statement-level Ambiguity** comes from the fact that different statements could invoke the same lib functions, which makes it difficult to identify the call-site of each trace entry. For example, in Figure 1, `openat` may be generated by the statement at Lines 13, 16 and 28, or other code related to the open operation. Simply testing all possibilities is exponentially time-consuming. **(2) Path-level Ambiguity** comes from the fact that there exist code snippets that would not generate any traces. For example, codes at Lines 32-36 will not generate any trace. To recover the corresponding control flow, we need information about the variable `command_line`. A naive approach is to apply traditional symbolic execution from the start and find valid paths to

[2]Although ARM's Embedded Trace Macrocell (ETM) in its Coresight architecture is similar to Intel PT, many chip designers choose not to expose it in the device tree [20, 27].

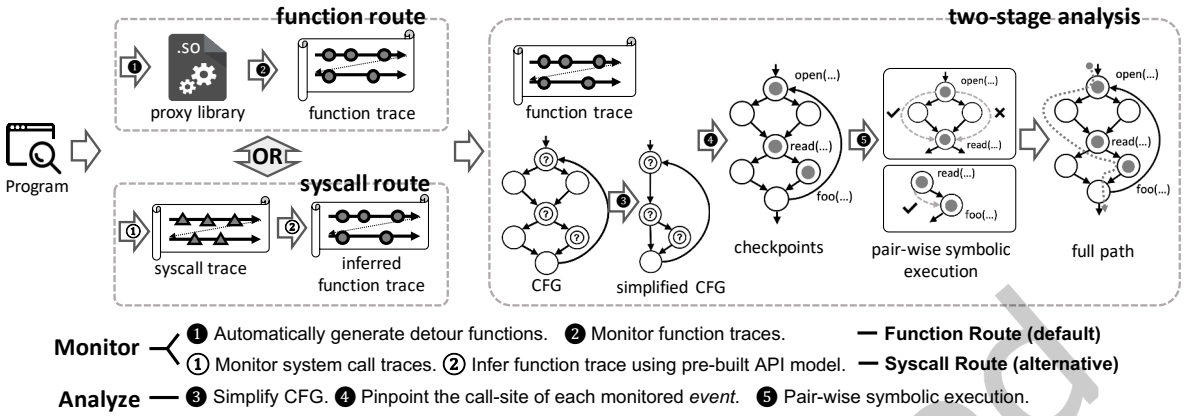


Fig. 2. Workflow of *Adonis*.

Line 38. This approach would dive into every single branch and try to find all constraints. It would easily fall into path explosion considering the complexity of current programs.

3 APPROACH OVERVIEW

We propose the design of a tool, *Adonis*, that achieves instrumentation-free, hardware-independent, and fully automated control flow recovery for production environments. The input of *Adonis* includes a target program whose source code may or may not be available and the OS-level features, such as system call and shared library call traces, before the crash. The output is the execution path of the input program that leads to the crash. The high-level workflow of *Adonis* is shown in Figure 2.

Adonis allows two types of input traces, i.e., the traces of shared library calls (*function route*) and the traces of system calls (*syscall route*). We support these two types of input to cover popular use cases of applications. By default, *Adonis* accepts the traces of shared library calls through the *function route*. However, we also notice that shared library call traces may not always be available [40]. For example, a self-contained executable that statically links all libraries does not have shared library call traces. Therefore, *Adonis* also accepts system call traces when shared library call traces are not available. Note that both types of traces can be easily collected using already-existing tools (e.g., function traces can be collected by setting "LD_PRELOAD" environment variable and system call traces can be collected by tools like strace [44] and sysdig [9], more details in §5).

Adonis contains two phases. In the first phase, it prepares the traces for analysis. In the function route, *Adonis* first scans the program and automatically generates detour functions of shared library functions (❶). Detour functions will record the internal information of library functions, such as function name, arguments, and return values. Then *Adonis* uses the function hooking techniques to monitor the function traces (❷). In the syscall route, *Adonis* first monitors the program's system call traces using system utilities (①) and then uses an offline pre-build API model to infer the corresponding function traces (②) (details in §4.1).

The second phase of *Adonis* is to analyze the control flow, which accepts the monitored or inferred library call traces and the program's control flow graph (CFG) and outputs the analyzed control flow. In this phase, *Adonis* uses a two-stage analyzing method to handle the statement-level ambiguity and the path-level ambiguity: (1) In the first stage, given library call traces, *Adonis* pinpoints each library call's call-site, which we define as a *checkpoint* (❹). Since many basic blocks may not call library functions, we propose a CFG simplifying method that removes "useless" basic blocks to improve the searching efficiency (❸) before analyzing the CFGs. (2) In the

second stage, given the checkpoints that contain the location and program state information, *Adonis* performs a pair-wise symbolic execution to recover possible paths between adjacent checkpoints (❺). *Adonis* handles the path explosion problem by limiting the search space and fully leveraging the information inside each checkpoint. Finally, *Adonis* concatenates these paths and outputs the analyzed full path.

4 DESIGN

In this section, we present the design details of *Adonis*.

4.1 System Call Trace to Function Trace

Adonis needs to infer the static library call traces when dynamic library call traces are not available, i.e., change to syscall route (Please refer to §3). To this end, we use an API model to infer the static library call traces based on system call traces.

API model introduction. The API model accepts a system call trace and outputs all possible static library call traces (i.e., the function traces). Given the generating speed of system calls (52.1/ms according to our experiments) and the exponentially increasing number of possible function traces, the model should quickly return the result. To this end, we organize the API model as an efficient information retrieval data structure, i.e., *Trie* [53], also called a *prefix tree*. Specifically, Trie is a tree-like data structure, and every node of a Trie consists of multiple branches. Each branch represents a system call, and each leaf node in the Trie contains the possible function(s) that would generate corresponding system calls from the root to the leaf.

Figure 3 shows an example of the Trie (i.e., the API model) used in our motivating example, which covers the related system calls and lib functions used in the code snippets. Starting from the "`root`" node, the `fstat`, `read` system calls may be generated by a `gets()` function (`leaf 1`), and the `write` system call may be generated by a `fprintf()` or `printf()` function (`leaf 3`).

API model construction. Building a Trie needs key-value pairs. Similarly, to build our API model, we need numerous API-to-Trace mappings (API for value and corresponding system calls for key). We collect these mappings from massive test cases provided by the glibc library. We instrument test cases by adding probes between library function calls so that we can obtain precise system call traces generated by each library function, i.e., the API-to-Trace mapping. Then given these mappings, we follow the Trie's insertion algorithm [26] to insert these mappings (key-value pairs) into the Trie pair by pair. Specifically, given a pair of API-to-Trace mapping, we start from the root node, take the branches that have the same system calls, add a new node if there are no corresponding system calls, and update the leaf node if necessary.

Figure 3 shows how our simple API model is constructed – In the beginning, the Trie is empty, i.e., there is only a `root` node. We then insert API-to-Trace mappings pair by pair. For pair `<fopen(), openat>`, there is no edge corresponding to `openat`, so we insert a new edge (`openat`) into the Trie and its leaf node (`leaf 5`) is `fopen()`. Then for pair `<fprintf(), write>`, we insert a new edge (`write`) and a new node (`leaf 3`). For pair `<printf(), write>`, we could find an edge corresponding to `write`, so we update the corresponding node (`leaf 3`) to `printf/fprintf`, which means that the `write` system call could be generated by `printf` or `fprintf`. For pair `<fgets(), fstat read>`, we insert a new node (`leaf 1`) that is two steps from the root. For pair `fstat()`, `fstat`, we insert a new node (`leaf 2`) to distinguish from `leaf 1`.

We build our API model using more than 10K API-to-Trace mappings, which ensure that the constructed API model could cover more than 99% of the lib functions and system calls in our benchmark. For new lib functions or system calls (like the ones from system update), one can just collect API-to-Trace mappings and update the Trie. Once the API model is built, there is no need to store these API-to-Trace mappings. We also collect some additional mappings from real programs (like nginx). These "realistic" mappings could make the Trie robust to the noise system calls from the system itself. For example, we have witnessed lots of `brk` and `switch` system

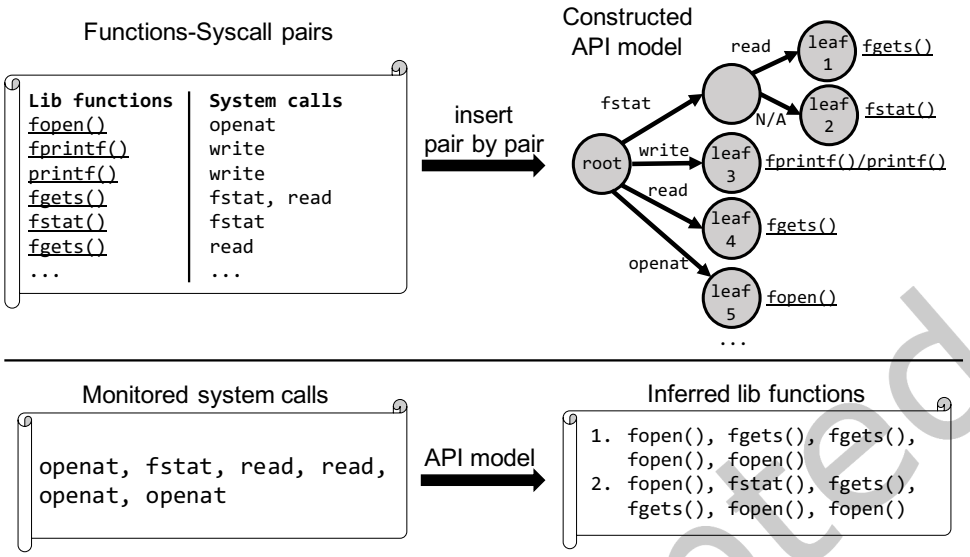


Fig. 3. An example of how API model works. Since there are extensive function-to-syscall mappings, we show only a small portion to illustrate the API model's construction.

calls that correspond to no functions. These system calls come from page fault exceptions and context switches, and the Trie could correctly ignore them once accepting these "realistic" mappings (please refer to more details about how *Adonis* handles these system calls in §8).

API model usage. Given monitored system calls, using the API model to infer the corresponding lib functions is a process of querying values (functions) by keys (system calls). Specifically, we start from the root node and take the corresponding branches until we reach the leaf node, and do the former operations recursively until all the system calls in the given trace have been inferred. Note that there could be more than one inference result, which means that all of these results could generate the given system calls.

We use the monitored system call trace in our motivating example to illustrate how the previously built API model works. As shown in Figure 3, given a trace containing 6 system calls, i.e., openat, fstat, read, read, openat, openat, the API model processes them sequentially. For openat, the Trie outputs `fopen` by following the edge of openat. For fstat, the Trie follows the edge of fstat and reaches a node that is not a leaf node (parent of leaf 1 and leaf 2). It then checks whether the succeeding system call could match the following edges (`read` and `N/A`). In our example, both the edges can be matched thus the Trie outputs two possible function traces. Similarly, The Trie processes the succeeding system calls until all of them have been inferred. Finally, our model outputs two sets of inferred lib functions.

API-model profile. In general, the API model, i.e., the Trie, used in the syscall route is a tree-like structure with a shallow depth and a relatively large width. The shallow depth indicates that each libc function will typically call only a few system calls (typically just one). The width reflects the number of system calls involved. The size of the Trie is 4.0KB. The depth of the Trie is 5. The width of the Trie is 68. Among the involved system calls, the `"write"` system call has the most functions grouped together at the leaf, which is 7. Note that more than seven functions could generate the `"write"` system call. However, only seven of them are used by our evaluated applications.

4.2 Execution Paths Recovery

Adonis recovers the execution paths of crashes from the dynamic or static library call traces. The key challenge is to handle the ambiguity of OS-level traces (i.e., statement-level and path-level ambiguity) while managing the analysis efficiency. To address this challenge, we design a two-stage approach that infers the execution paths with moderate analysis cost.

On the high level, in the first stage, *Adonis* first simplifies the program's CFG to accelerate the analysis and then performs a block-level inter-procedural analysis to pinpoint the call-site of each monitored event. We call the pinpointed call-sites the *checkpoints*. In the second stage, *Adonis* performs inter-procedural pair-wise symbolic execution to recover paths between checkpoints.

Pinpoint the call-sites. It is non-trivial to find the precise call-site for each monitored event due to the statement-level ambiguity, i.e., different statements could call the same lib functions. It can be regarded as search problem trying to find the longest match in a huge graph. Before we introduce our solution, we first formalize the task as a search problem in directed graphs.

Given a program's sCFG, we formulate it as a directed graph $G = (N, E)$, where the set of nodes N denotes the statements (or basic blocks) in sCFG, and the sets of edges E denotes the intra-procedural and inter-procedural edges in sCFG. Given a function trace (which can be monitored from the function route or be inferred from the syscall route) with M events, we formulate it as a sequence $T =< t_1, t_2, t_3, ..., t_M >$, and each event t_i as a tuple $(func_{t_i}, x_{t_i}, y_{t_i})$, where $func_{t_i}$ is the corresponding function, x_{t_i} represents the function's arguments, y_{t_i} represents the function's return value.

Now we can formulate the task as the following problem: given a simplified CFG G and a function trace T, we call π a valid pinpointing result if there exists a valid path $P =< n_0, n_1, n_2, ... >$, such that the function calls of P is equal to the functions in T. To solve the problem, a naive approach is to start from the program's entry n_0 and then traverse and test successor nodes until a valid path that matches the trace is found. However, this approach is inaccurate because it does not use the information that lies in each event t, i.e., arguments x_t and return values y_t. As a result, this approach would require more analysis time or get worse results in precision/recall. Intuitively, some arguments of library functions are usually constant values that can be easily obtained through little analysis, e.g., the format string in `printf` and flags and mode in `open`. Based on this observation, we propose the following algorithm as shown in Algorithm 1.

Algorithm 1: $pinpoint(G, T)$: pinpoint the call-sites of given trace

Data: simplified CFG G, function trace T
Result: valid pinpointing result π

$n_d \leftarrow find\ most\ definite\ node\ from\ T$
$func_d \leftarrow get\ function\ of\ n_d$
$paths \leftarrow search\ path\ in\ func_d$
$\pi = List()$
for $path \in paths$ **do**
    $i, j \leftarrow path's\ start/end\ index\ in\ T$
    $t \leftarrow (func_d, x, y)$ // treat $func_d$ as a new event.
    $traceLeft \leftarrow T[:i] + [t] + T[j:]$
    $\pi.insert(pinpoint(G, traceLeft))$
return π

We implement Algorithm 1 using the Breadth First Search (BFS) algorithm with caching optimization. The high-level idea of Algorithm 1 is to first find the *most definite node* (n_d at Line 1). A *definite node* is a node in the CFG that can be logically linked to a specific event in the monitored trace. For instance, an output statement, `printf("file %s ABC does not exist", filename)` can be considered as a *definite node*, since it can be traced back to a `"write"` event that writes the data `"file ABC does not exist"`. Conversely, given this `"write"` event, the corresponding *definite node* (i.e., the `printf` statement in the example) is the call-site that we aim to identify. To achieve this, we analyze the parameters of library functions and match the constant values (e.g., `"file %s ABC does not exist"` in `printf` statement) to the fields in events (e.g., `"file ABC does not exist"` in the write event). Currently, the events used to determine the *definite node* are related to output (like `printf`, `fprintf`, etc.) and file control (like `fopen`, `opendir`, etc).

There may be multiple *definite nodes* and *Adonis* needs to select one of them as the starting point for path recovery, preferably the one with the highest degree of certainty or the *"most definite node"*. To determine this, *Adonis* calculates the priority of each definite node. Generally, a node's priority depends on the complexity of its parameters. If a node's parameter is a complex string rather than a simple string or integer, it may be more difficult to match with an event. Therefore, if such a node is successfully matched and confirmed as a *definite node*, it should be assigned a higher priority.

If Algorithm 1 fails to identify a *definite node* on Line 1 (which is rare in our experiments), *Adonis* will attempt to locate the call site by searching from the program's entry point. This means that the first node that matches the first trace item will be treated as the *definite node*. Moreover, if the stack trace of the failed execution is available, Adonis can perform a reverse search starting from the crashed function.

Once the *definite node* is determined, we expand from it and search for a valid path in the function ($func_d$) such that the trace generated by this path can match part of the whole trace. Then, we consider the matched partial trace as a whole, whose function name is $func_d$. So we can replace this partial trace with a new event in the whole trace (Lines 6-8). Then we recursively call Algorithm 1 on the newly constructed trace (Line 9) until all the events are matched.

Pair-wise symbolic execution. After we pinpoint the call-site of each monitored event, the next step is to recover the full path. Note that we should use the original CFG instead of the sCFG (the pinpointing result on sCFG can be seamlessly ported to CFG). In this step, we encounter the challenge of *path-level ambiguity*, i.e., for some code snippets, there is no trace to indicate the control flow. A naive approach that performs symbolic execution on the whole program would easily fall into path explosion. We notice that the call-sites in previous step have divided the programs into small subroutines, on which it is affordable to perform symbolic execution. Thus we propose the pair-wise symbolic execution to fully leverage the information in traces while managing the search space. In Figure 4, we use the motivating example to show how pair-wise symbolic execution is conducted.

Inspired by the concolic execution [25, 52], the high-level idea of pair-wise symbolic execution is to (1) recover paths iteratively instead of all at once and (2) try to use concrete values recorded in traces instead of calculating symbolic constraints. Specifically, we refer to each pinpointed call-site as a checkpoint *ckpt*. We refer to the process of searching for valid paths between a pair of adjacent checkpoints $< ckpt_i, ckpt_{i+1} >$ as one *step*. In each step, there are two types of constraints used: (1) the first type is the *trace constraint* that is imposed by the concrete call-sites and corresponding (function, arguments, return value) tuples. For example, in Figure 4, for `ckpt L28`, we monitored a trace `fopen("settings","r") = 4` and the *trace constraint* for `ckpt L13` is `fp == 4` where `fp` is the variable that accepts `fopen`'s return value. (2) the second type is the *historical constraint* that is the path constraints accumulated up until the current checkpoint. We will introduce how *Adonis* maintains the historical constraint when performing pair-wise symbolic execution.

Adonis recovers the full path step by step from the starting checkpoints to the end. In each step:

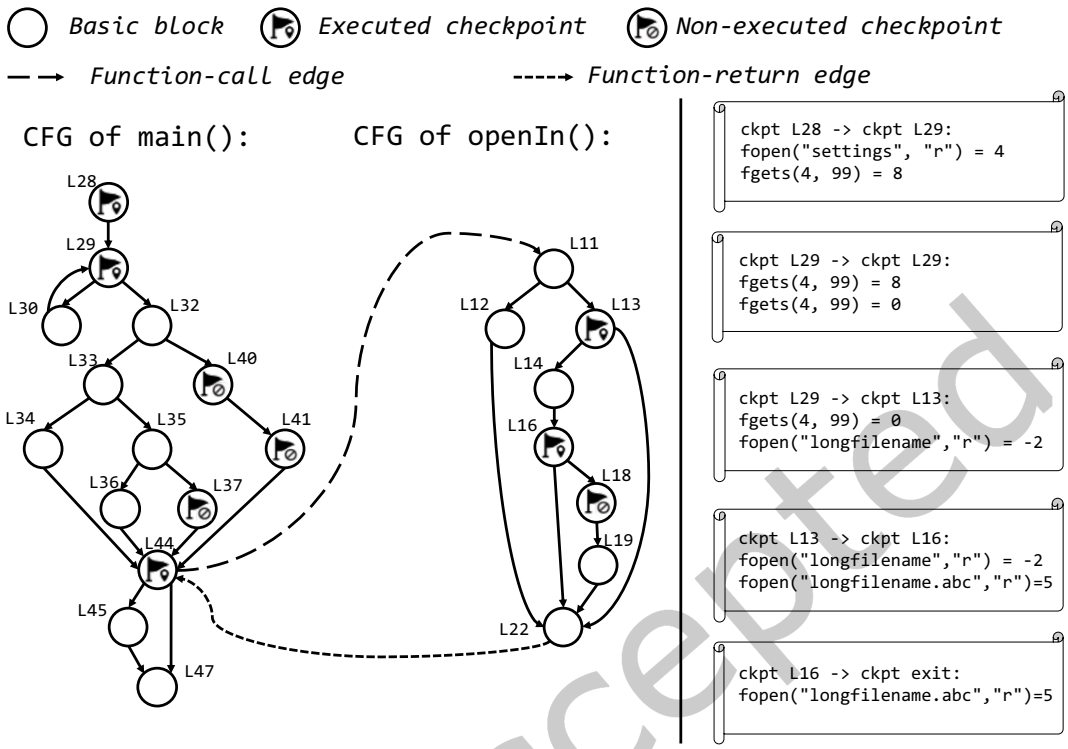


Fig. 4. An example of how *Adonis* performs pair-wise symbolic execution.

(1) *Adonis* checks if there is only one path from $ckpt_i$ to $ckpt_{i+1}$. If so, this path is the result for this step and *Adonis* symbolically executes the program from $ckpt_i$ to $ckpt_{i+1}$ and updates the *historical constraint* accordingly. For example, in Figure 4, there is only one path from `ckpt L28` to `ckpt L29`. In this case, *Adonis* regards this sub-path as the result of this step and update the constraint, i.e., add a `fp == 4` constraint.

(2) If there is more than one path, *Adonis* first uses the pinpointing information to reduce search space, i.e., possible paths. Note that there are some basic blocks that are *important* (would generate trace) but not pinpointed as checkpoints. We consider these basic blocks as non-executed and this information can help reduce the search space without performing symbolic execution. For example, for the step from `ckpt L16` to `ckpt exit`, basic block `L18` (a printing statement) is considered non-executed. So the path `L16 -> L18 -> L19` is excluded.

(3) After reducing the search space, *Adonis* tries to find valid paths according to the *historical constraint* and *trace constraint*. Specifically, for each branch, *Adonis* uses the existing constraints to try to deduce the branch condition. If it can deduce the condition, it will take the corresponding branch. If not, it will symbolically execute each branch and update the constraint. Finally, the valid paths (the ones whose constraints is solvable by SMT solver) will be considered as the results of this step. For example, there are multiple paths from `ckpt L29` to `ckpt L13`. After the symbolic execution, only the path `L29 -> L32 -> L33 -> L35 -> L36 -> L44 -> L11 -> L13` is valid. Path to `L34` is excluded (unsatisfied) because it assigns the string `"stdin"` to the variable input, but it should be `"longfilename"` according to the trace constraint at `ckpt L13`.

(4) If there are more than one valid path in this step, *Adonis* cannot determine which path is truly executed. On the one hand, it would report the uncertain basic blocks as *may-executed*. On the other hand, it would maintain the historical constraint. Specifically, it performs a live variable analysis [49] and reset the constraint to the live variable if it is used by more than one valid path. We perform this operation to avoid the case that wrong constraints are passed as historical constraints to succeeding checkpoints.

The pair-wise symbolic execution is effective and efficient. It is effective because it utilizes the locality of the program. We find that nearby checkpoints can determine most conditions, thus greatly decreasing the search space. It is efficient because it tries to use concrete values stored in traces to reduce uncertain searches.

We also take several strategies to avoid path explosion. First, when we encounter a function call, similar to previous work [57], we implement *strongest observable necessary condition* [16] for constraint conversion. It guarantees that the converted constraint is a necessary condition of the original one by keeping only the caller-observable conditions such as return values, function arguments, and global variables. Second, we limit the depths of the call stack to prevent an infinite loop. Third, we also limit the search time for each step. Once it times out, it will regard all the paths in the unsearched space as *may-executed*.

Finally, paths between pairs of checkpoints will be concatenated as possible full paths. *Adonis* cannot guarantee that there is only one feasible path left after the pair-wise symbolic execution. So the output of *Adonis* (after the pair-wise symbolic execution) is a rough but usable control flow (organized as must/must not/may executed basic blocks). As we have described in §1 and illustrated in §2, this control flow provides valuable information for failure diagnosis.

5 IMPLEMENTATION

Overall implementation. We implement the prototype of *Adonis* on Linux system based on *EOSAFE* [30], a state-of-the-art symbolic execution engine for WebAssembly. Our implementation contains ~8.2k lines of Python code. We choose WebAssembly because it is a low-level language that can be translated from other mainstream programming languages, e.g., C, C++, Go, Rust, etc. and binaries without source code can also be lifted to LLVM IR and then recompiled to WebAssembly [37]. *Adonis* does not rely on the source code to perform its analysis, even for the symbolic execution. All of *Adonis*'s analysis is performed on the WebAssembly (wasm) binary code.

In the function route, *Adonis* collects shared library call traces to recover the control flow. To collect such traces, *Adonis* automatically generates a proxy library for the given program by following the steps below: it first uses RetDec [3], an LLVM-based machine-code decompiler, to scan the program and identify used dynamic functions and their signatures. Based on the signature, *Adonis* then generates a detour function for each identified dynamic function. The detour functions are then compiled to the proxy library, which is a shared object (with the `.so` suffix) accounting for 10 - 100 KBs of storage (depending on the number of detour functions). Finally, *Adonis* modifies the LD_PRELOAD environment to direct calls to the original functions to our detour functions to collect calling traces. The process of generating the proxy library takes 1-5 minutes.

In the syscall route, *Adonis* collects system call traces to recover the control flow. To collect such traces, we use sysdig [9] to monitor the given program and implement a trace parser. What is more, *Adonis* needs the offline-built API model (i.e., the Trie in §4) to infer a system call trace to its possible function traces. Building such an API model takes around 10 minutes and the built API model accounts for less than 10 KBs of storage and can be re-used in the analysis of different programs.

CFG Simplification optimization. This is a preceding step before *Adonis* pinpoints the call sites. We design this step to improve the efficiency of the pinpointing step. Specifically, considering that OS-interface or lib functions are not called intensively, there are many *ordinary* statements that will not generate any OS-level trace lying between *important* statements that could generate traces. According to our experiments, these *ordinary* statements account for the majority of the code (>80% on average) but will not generate any OS-level traces. As a

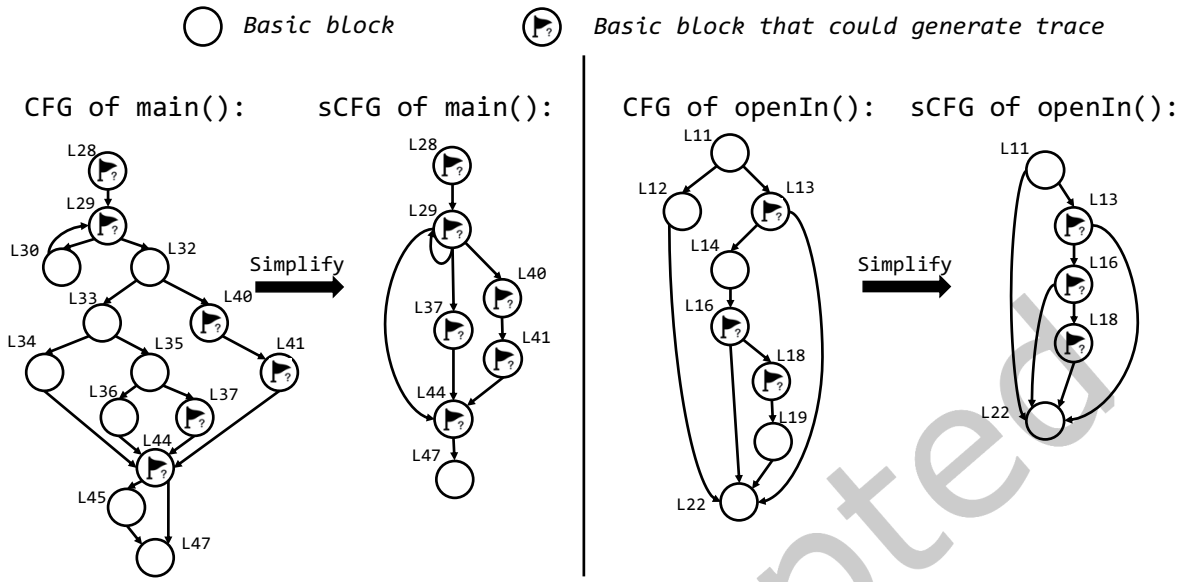


Fig. 5. CFG and sCFG of code snippets in the motivating example.

result, directly searching (i.e., pinpointing) the call-sites among the original CFG is inefficient because of the noise branch, loop, and function calls from *ordinary* statements. So, we are motivated to first perform a simplifying operation to remove unrelated but massive *ordinary* statements from the original CFG and remain a backbone (which we call an *sCFG*) that could generate the same trace as the original CFG.

As shown in Figure 5, we demonstrate the high-level idea of the simplifying operation using the CFG and sCFG of our motivating example. For any path in CFG, we can find a corresponding path in sCFG so that both paths generate the same OS-level traces. What is more, sCFG contains much fewer nodes than CFG because most *ordinary* statements or basic blocks have been removed. As a result, searching (pinpointing the callsites) based on sCFG instead of CFG will greatly improve the efficiency and will not decrease the correctness of the results.

Next we introduce how to simplify a CFG to get its sCFG. First, we define that a statement (or a basic block) is *ordinary* when *it will not generate any OS-level traces*, otherwise, it is *important*, which means that it could generate OS-level traces when executed. Note two types of statements (or basic blocks) are *important*, i.e., the ones that directly call lib functions and the ones that call functions that contain *important* statements (or basic blocks). Given a function's CFG, *Adonis* takes the following steps to simplify this CFG.

(1) Topological sort all edges in the CFG.
(2) Traverse the sorted edges. For any edge $A \rightarrow B$ that A is not the function's Entry node and B is not the function's exit node, if A or B is *ordinary* and there exists only one path from A to B, do the following *remove* operation:
   (a) Remove the edge $A \rightarrow B$ from CFG so that on of the nodes would be removed latter.
   (b) Remove one node (basic block or statement) that is *ordinary* from CFG (suppose A is *ordinary* and removed), and modify all edges pointing to or starting from A to point to or start from B.
   (c) If there are redundant edges, only one of them is retained.
(3) Repeat step 1 and 2 until there is no edge to remove.

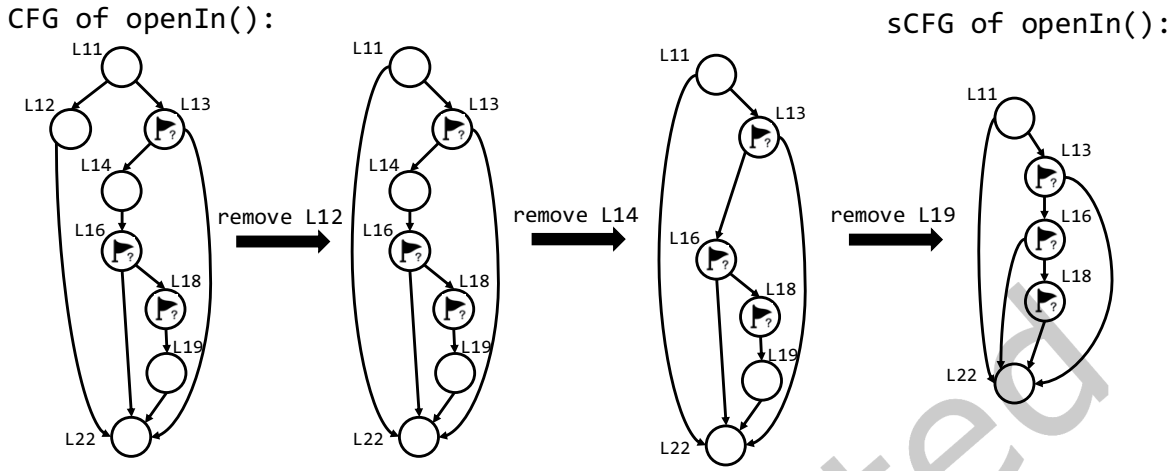


Fig. 6. An example of how CFG is simplified.

Figure 6 shows an example of how function `openIn`'s CFG is step-by-step simplified. For edge `L11` → `L12`, both `L11` and `L12` are *ordinary*, so one of them should be removed. Here `L12` is removed because `L11` is the function's entry node. For edge `L13` → `L14`, `L13` is *important* and `L14` is *ordinary*, so `L14` is removed. Similarly, `L19` is removed when processing edge `L18` → `L19`. Then the simplification finished because there is no edge to be removed.

This CFG simplification optimization could improve the efficiency of the afterward pinpointing step (the step in §4.2), especially when the analyzed program is complex and has huge functions. For example, `sqlite3VdbeExec()` is a huge function in sqlite3 with 1331 basic blocks. Our optimization reduces the number of basic blocks to 497 (reduced by 62.7%), leading to a smaller search space when pinpointing the call-site. According to our prior experiments, without the CFG simplification optimization, Adonis will take more than 12 hours to finish the pin-pointing step for a complex program like sqlite3. However, after adding the optimization, it can finish in 30 minutes.

6 EVALUATION

We focus on evaluating whether OS-level traces are sufficient to accurately recover control flow for programs in production environments with moderate runtime, deployment, and development cost. In particular, we answer following four research questions:

RQ 1: How accurate is *Adonis* in control flow recovery?
RQ 2: What is the runtime cost of *Adonis*?
RQ 3: What is the deployment cost of *Adonis*?
RQ 4: What is the development cost of *Adonis*?

6.1 Experimental Settings

Benchmarks: Our benchmark contains 11 real applications or tools, of which 8 are desktop applications used by previous work [47, 55] and the other 3 are IoT applications or libraries selected from *awesome-open-iot* [1], a curated list of open source IoT frameworks, libraries and software. The three IoT applications we select are the most popular ones in the list implemented in C. For each application, we use its default test cases as the

Applications	Description	LOC	BBs	functions
tcas	SIR application [17]	173	163	20
replace	SIR application [17]	563	437	53
tot_info	SIR application [17]	564	231	26
abc2mtex	Music notation	4,764	4,328	462
gzip	Linux utility	8,114	3,704	236
space	ADL interpreter	9,563	3,895	253
grep	Linux utility	15,460	7,121	177
microcoap	CoAP server	809	795	11
mqtt-sn-tools	MQTT for networks	1,675	1,258	37
libmodbus	Modbus protocol	5,435	1,996	64
sqlite3	Database management system	236,528	86,852	2,930

Table 2. Evaluated applications.

benchmark. If the number of test cases is too large, e.g., *space* has 13,525 test cases, we randomly choose 100 of them. We provide information about CFGs for each application in Table 2, including lines of code, numbers of functions, and basic blocks.

Baselines: We choose four baselines to compare against, representing the state-of-the-art hardware-based, instrument-based, and log-based methods that can recover the control flow. The first one is *Bunkerbuster* [56], a bug hunting framework based on Intel-PT. *Bunkerbuster* symbolically reconstructs program states leveraging the hardware traces and partial memory snapshots (**hardware-based method**). The second baseline is *gcov* [34], a commercial tool to test program coverage widely used by LLVM and GCC (**instrument-based method**). The third baseline is *Pensieve* [60], a tool for failure reproduction. *Pensieve* reconstructs failure reproduction steps based on log files (**log-based method**). Since *Pensieve* is not publicly available, we reproduce it in our platform. The fourth baseline is *s-VPA* [47], which is a control flow recovery tool based on selective instrumentation (**instrument-based method**). Note that we only compare *s-VPA* with *Adonis* in terms of accuracy, because we cannot find the exact value of the three types of cost in the paper. For example, the analysis time (measured as deployment cost) reported in the paper is a relative value.

Experimental environment: To make the experiments close to reality, we set up two environments, i.e., (1) a tracing environment where applications run on ordinary devices, and (2) an analysis environment where traces are analyzed on more powerful devices. For the tracing environment, we use two hardware platforms, including an x86-64 desktop running 8 desktop applications and an ARM raspberry pi 3B running 3 IoT applications. The desktop is equipped with an Intel i7-9750H CPU, 16GB memory, and 1TB storage and the raspberry pi is equipped with a 1.2GHz Broadcom BCM2837 CPU, 1GB memory, and 16GB storage. For the analysis environment, we perform analysis on an x86-64 server. The server is equipped with an Intel Xeon E5-4620 v2 CPU, 32GB memory, and 3TB storage. Both the desktop and server run a 64-bit Ubuntu 18.04, and the raspberry pi runs a 64-bit Raspberry Pi Bullseye OS.

6.2 RQ 1: Accuracy

We first evaluate whether OS-level traces can accurately recover control flow of programs. To answer this RQ, we compare *Adonis* to *Pensieve* [60], a log-based technique that recovers control flow information, and *s-VPA* [47], a path recovery tool based on selective instrumentation. We omit the Gcov and Bunkerbuster since they are

	Block-level Accuracy				Recall			Precision		
	s-VPA	Pensieve	Adonis (function)	Adonis (syscall)	Pensieve	Adonis (function)	Adonis (syscall)	Pensieve	Adonis (function)	Adonis (syscall)
tcas	88.1%	72.2%	95.7%	94.8%	71.5%	92.7%	92.7%	96.0%	96.2%	96.2%
replace	93.1%	71.5%	93.5%	93.0%	68.5%	89.4%	89.4%	94.4%	97.6%	97.3%
tot_info	97.9%	76.7%	98.5%	97.2%	70.3%	95.5%	94.1%	80.2%	93.3%	86.1%
abc2mtex	-	42.8%	76.7%	73.9%	44.7%	76.6%	71.7%	83.5%	94.9%	92.4%
gzip	61.9%	20.3%	67.5%	53.9%	28.8%	98.6%	99.4%	40.1%	78.4%	74.9%
space	58.6%	38.5%	70.2%	64.4%	47.3%	73.1%	75.2%	63.7%	76.5%	67.6%
grep	48.3%	22.7%	54.5%	53.6%	33.2%	92.5%	91.6%	43.2%	52.9%	50.8%
microcoap	-	55.2%	86.7%	86.0%	47.4%	79.3%	71.9%	79.2%	86.1%	86.1%
mqtt-sn-tools	-	44.3%	87.7%	84.6%	35.7%	90.6%	81.8%	70.5%	80.0%	76.3%
libmodbus	-	48.3%	85.1%	79.0%	37.5%	92.2%	88.3%	72.5%	83.0%	77.7%
sqlite3	-	43.3%	74.5%	67.2%	35.5%	74.8%	70.3%	45.5%	59.5%	57.4%
Average	74.6%	48.7%	81.0%	77.1%	47.3%	86.8%	84.2%	69.9%	81.7%	78.4%

Table 3. Block-level accuracy, recall, and precision of *Adonis* and baselines.

designed to have 100% accuracy. However, these two techniques have high runtime or deployment overhead for production environments (§6.3 and §6.4).

We measure three metrics, which are block-level accuracy, precision, and recall. We define the block-level accuracy as the ratio of basic blocks that can be definitively categorized as "executed" or "not executed". *Recall* is measured by $\frac{\#TP}{\#TP+\#FN}$ and *precision* is measured by $\frac{\#TP}{\#TP+\#FP}$, where $\#TP$ is the number of basic blocks executed in the path, $\#FN$ is for executed but not in the path, and $\#FP$ is for in the path but not executed. We select these metrics because they intuitively reflect the accuracy of the recovered path and are widely used in related work [47]. Only block-level accuracy is reported in the paper of *s-VPA*, so we only show its block-level accuracy and omit the other two metrics. We build the ground truth by running Gcov. The results are shown in Table 3, in which we used arithmetic means instead of geometric means because there are test cases that no log messages get output by the application, making *Pensieve* cannot recover any control flow. As a result, its accuracy is 0 for these test cases. Including these zero values to calculate the total geometric means would also result in a mean value of zero. We also cannot remove these test cases because it is unfair to Adonis.

Block-level accuracy: We observe that, *Adonis* (77.1% and 81.0%) substantially outperforms *Pensieve* (48.7%). The control flow recovered by *Adonis* can not only cover the results of *Pensieve* but also cover 74.9% statements missed by it. This indicates that OS-level traces are able to record more valuable information compared to logs. We will show more details when we evaluate the recall and precision. *Adonis* also outperforms *s-VPA* (74.6%). To handle the runtime cost of instrumentation, *s-VPA* selectively instrumentation the program, i.e., it only traces executed functions and paths of functions in the call stack [48]. As a result, this selective instrumentation provides limited information of the paths of functions outside the stack (i.e., functions that have been called but popped from the call stack). By contrast, the OS-level traces used by *Adonis* cover the information of these functions thus *Adonis* could get more accurate results.

Recall and precision: The results are shown in Table 3. Overall, *Adonis* outperforms *Pensieve* by 37.12% to 39.56% on recall and 8.21% to 11.56% on precision. *Pensieve* gets less effective in two aspects. First, in production environments, log levels are typically suppressed for brevity concerns [11]. For example, in `sqlite3`, the C preprocessor variable `SQLITE_DEBUG` is not set in the release version, and the DEBUG level logs would not be output. As a result, logs often record only limited paths or variables, leaving broad space not traced. Second, to

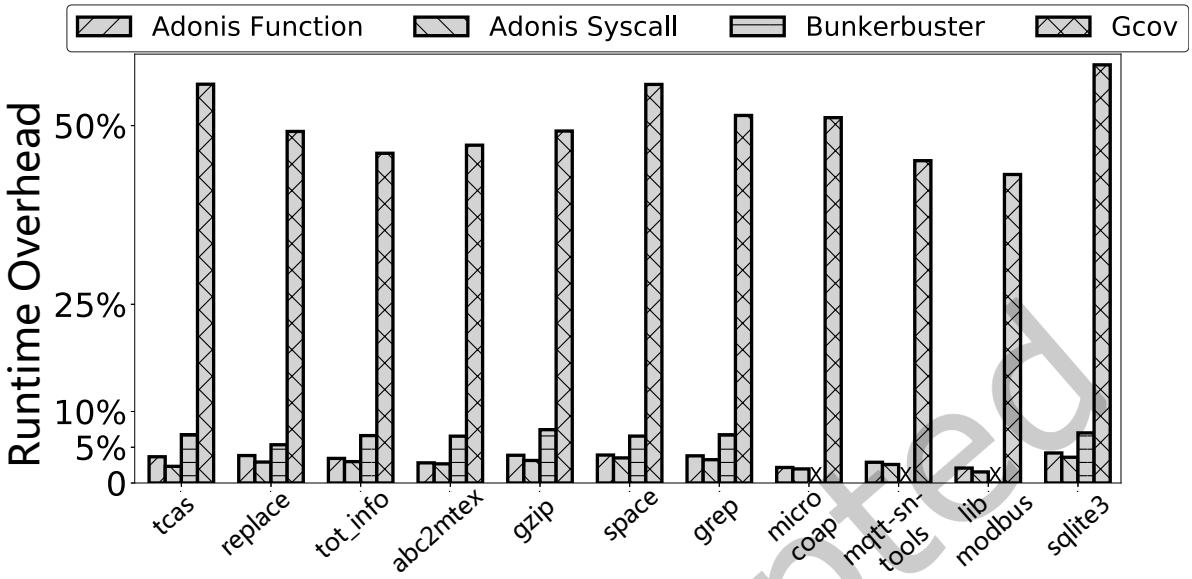


Fig. 7. Runtime overhead of *Adonis* and baselines.

improve the scalability and efficiency, *Pensieve* skips a vast majority of the code paths by focusing on instructions causally relevant to the failure. By contrast, the OS-level traces, used by *Adonis*, not only cover the logs written by developers by recording I/O events, but also contain valuable program states that could be missed by logs, e.g., `libmodbus`, as a communication protocol library, suppresses many log points in the release version (the version used in production). *Pensieve* suffers a low recall (37.5%) due to the aforementioned two aspects. Meanwhile, *Adonis* can still effectively recover the path based on the traced events and achieve a much higher recall (92.2%).

Ans. to RQ 1: *Adonis* can recover the control flow with 81.0% block-level accuracy, 86.8% recall, and 81.7% precision. *Adonis* substantially outperforms the state-of-the-art log-based control flow recovery methods in terms of the accuracy. Specifically, *Adonis* not only covers the execution paths recovered by *Pensieve* but also recovers 74.9% of statements missed by it.

6.3 RQ 2: Runtime Cost

In this RQ, we evaluate if the runtime cost of *Adonis* is low enough for production environments. We measure a tool's runtime cost by calculating the slowdown of an application when it is being traced by the tool compared to when it is not. For example, the runtime cost of "Adonis syscall" is the slowdown of an application when *Adonis* collects the application's system call traces, which does not contain the overhead of call trace inference. Similarly, the runtime cost of "Adonis function" is the slowdown of an application when it is hooked by the proxy library generated by *Adonis*. We show the results in Figure 7. We do not include *Pensieve* because its runtime overhead is zero by design.

On average, *Adonis* induces 3.34% runtime overhead when tracing dynamic library functions and induces 2.78% overhead when tracing system calls, which is lower than *Bunkerbuster* (6.58%) and *Gcov* (49.4%). Note that Intel-PT is designed to minimize the runtime overhead of control flow recovery [55]. Therefore, the runtime cost of *Adonis* is acceptable in production environments.

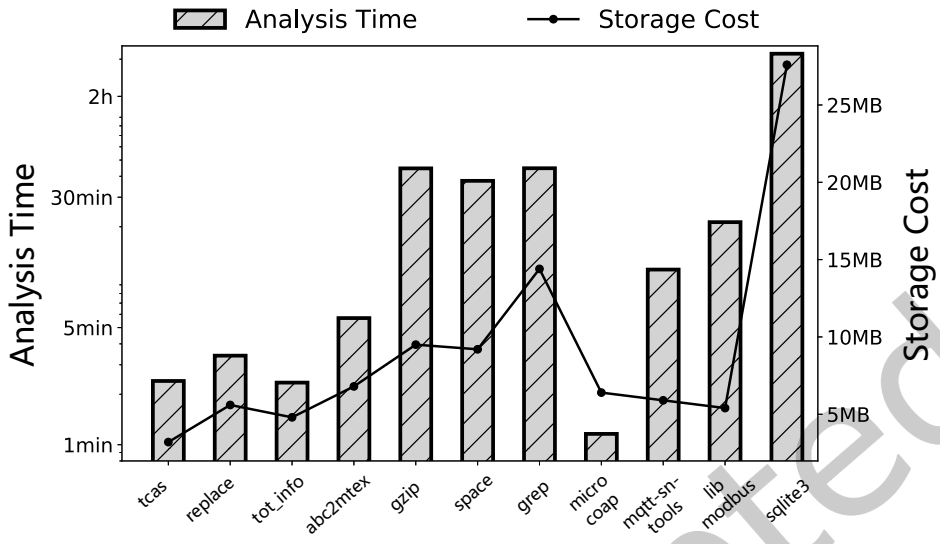


Fig. 8. Analysis time and storage cost of *Adonis*.

	Bunkerbuster	Pensieve	Gcov	Adonis (syscall)
Storage Cost	1.6 GB	2.2 MB	9.2 MB	7.8 MB
Analysis Time	35 hours	31.5 mins	-	38.6 mins
Deployment on Different Platforms	Hard	Easy	Easy	Easy
Source Code Dependency	No	Yes	Yes	No

Table 4. Summary of the deployment cost of *Adonis* and baselines.

Our evaluation also shows that *Gcov*, a full instrumentation tool, may introduce a slow down of more than 50% (18.3× higher than *Adonis*) to programs. This overhead is too high for production environments, particularly for delay sensitive applications[19]. Note that many instrument-based approaches do partial tracing at the cost of accuracy. So the comparison in this RQ cannot be stretched to all instrumentation based tool.

Ans. to RQ 2: *Adonis* has moderate runtime cost for control flow recovery in production environments. On average, *Adonis* induces 3.34% and 2.78% overhead to trace dynamic library functions and system calls respectively. This overhead is lower than *Bunkerbuster* (6.58%), the state-of-the-art hardware-based method designed for reducing the runtime cost of control flow recovery. Moreover, this overhead is 18.3× lower than *Gcov*, the commercial instrument-based method.

6.4 RQ 3: Deployment Cost

We evaluate how *Adonis* reduces deployment cost by conducting experiments from four aspects. First, we evaluate *the ease of deploying* Adonis *on different hardware*. In this experiment, we measure whether *Adonis* can run in desktop, cloud, and IoT environments, respectively, without heavy modifications. Second, we evaluate *the ease of deploying* Adonis *when source code is not available*. Third, we measure the storage overhead to keep the logs of *Adonis*. Fourth, we measure the analysis time to process the logs of *Adonis*. The analysis time evaluates whether *Adonis* requires extra computational resources to analyze generated logs. The results are shown in Figure 8 and Table 4.

Deployment across different hardware platforms: *Adonis* achieves the same block-level accuracy, precision, and recall across the desktop, IoT, and Cloud environments without any modification. In other words, *Adonis* can be directly deployed in different hardware platforms without any extra efforts. On the contrary, *Bunkerbuster* only works in the desktop environment as it depends on Intel-PT.

Dependency on source code: We further evaluate *Adonis* by measuring its block-level accuracy, precision, and recall on the executables of our test cases without debugging information (e.g. with the the `-g` configuration during compilation). Our experiment shows that *Adonis* achieves the same values. On the contrary, *Pensieve* and *Gcov* fail because they require source code.

Storage Cost: According to Figure 8, *Adonis* requires 3.2 ~ 27.6MB to keep the OS-level traces and the mean value is 7.8MBs. In contrast, *Bunkerbuster* requires 1.6 GB, *Gcov* requires 9.2 MB, and *Pensieve* requires 2.2 MB. The storage cost of *Adonis* is low enough for production environments considering that it is similar to the cost of popular logging practice, i.e., the cost of *Pensieve*. Our results also show that the storage cost of *Bunkerbuster* is three orders of magnitude higher than others. This is because Intel-PT is agnostic to upper level OS designs like process and context switch and it will simply record all instructions executed by the CPU. As a result, developers should deploy more storage when using hardware-based methods.

Analysis Time: According to Figure 8, the analysis time of *Adonis* is 1.2 ~ 215.8 mins, which is positively correlated with the length of the trace (illustrated as storage cost). And the mean value of *Adonis*'s analysis time is 38.6 mins, which is similar to *Pensieve*. *Adonis*'s analysis time is practically small considering that developers spend 49.9% of their programming time (several hours) in debugging [11]. On the contrary, *Bunkerbuster* needs 35 hours to decode and process the trace. It means that developers need to deploy around 50× number of machines to achieve a similar analysis time, which increases the deployment cost of Intel-PT based techniques.

We also notice that in terms of the deployment cost, *Pensieve* is comparable to *Adonis*. However, it is also notable that according to RQ 1 and RQ 4, Adonis has better accuracy and lower development cost.

> **Ans. to RQ 3:** *Adonis* has reasonable deployment cost for a production environment. First, It can be deployed in desktop, IoT, and cloud environment without any modifications. Second, the storage cost (7.8MB on average) and analysis time (38.6 mins on average) of *Adonis* is moderate and significantly smaller (443× smaller for storage cost and 50× smaller for analysis time) than Intel-PT based techniques, saving a large amount of storage and computational resources. Further, *Adonis* does not require source code while the log-based and instrument-based baselines are source code dependent.

6.5 RQ 4: Development Cost

To evaluate the development cost of *Adonis*, we evaluate the development efforts to adapt *Adonis* to a new version of software. As a comparison, we also measure the extra log printing statement needed for log-based techniques to achieve the same level accuracy as *Adonis*.

Development efforts to adapt to software update: In our experiment, *Adonis* can adapt to the software updates with negligible development efforts. Given a new version of the program, *Adonis* only needs to re-generate the proxy library to hook previously uncovered library functions. This process is fully automated so there are no extra development efforts for developers when using *Adonis*. On the contrary, for *Pensieve*, developers need to manually insert logging statements and its efforts depend on the complexity of the new features.

Extra log printing statements: In our experiment, *Adonis* records 393.3 log events on average. On the contrary, *Pensieve* generates only 21.3 log events and around 30% of the test cases output no logs. To achieve the accuracy similar to *Adonis*, developers need to insert 18.4× (393.3/21.3) log printing statement for log-based techniques.

Ans. to RQ 4: *Adonis* has reasonable development cost for a production environment. On the one hand, *Adonis* is fully automated while log-based methods rely on developers to manually insert log printing statements. On the other hand, a log-based technique requires developers to add 18.4× of new code to achieve the same accuracy as *Adonis*.

7 RELATED WORK

Our work is particularly related to two streams of literature: control flow recovery and OS-level traces analysis.

Control flow recovery. Control flow recovery has been widely applied in many software development tasks, i.e., software failure analysis [4, 14, 21, 45, 47, 54–57, 60], service contexts understanding [38, 39], etc. Some typical methods can be categorized into the follows. *Instrument-based methods* insert probes in the monitored programs at compile time, to record or measure executed paths [2, 4, 13, 46, 48, 51]. The main limitation of instrument-based methods is the high runtime cost. Although there are a few methods that aim to reduce the runtime cost, they are semi-automated, and thus still have high engineering cost [2, 13, 48, 63]. Specifically, developers need to manually specify targeted code snippets [2, 48] or iteratively run the programs multiple times [13, 63]. There are two pieces of instrument-based work (i.e., *s-VPA* [47] and *Trafic* [51]) that are close to *Adonis*. For *s-VPA*, its analysis is based on the trace provided by a lightweight and selective instrumentation tool [48]. What is more, according to our experiments in §6.2, the block-level accuracy of Adonis is higher than *s-VPA* by 6.4% on average. For *Trafic* [51], it is also an instrument-based method. It needs the application's source code to add its tracing logic and recompile the instrumented source code. *Log-based methods* guide the control flow recovery by analyzing logs inserted by developers. The limitation of log-based methods is that they depend on manually placed log printing statements and, thus, have high development cost [32, 33, 41, 57]. Several methods aim to automatically place log printing statements [59, 61], they are still semi-automated and require human efforts [12, 28]. *Hardware-based methods* recover control flow based on traces from hardware features [15, 24, 54, 64]. These methods depend on Intel-PT so they have high deployment cost in cloud or IoT environments. There are also several hardware tracing techniques based on ARM [15], but these techniques are limited to in-house debugging.

OS-level traces analysis. OS-level traces (e.g., system logs) are informative and valuable indicators for system behaviors. Existing work analyzes system logs mainly for anomaly detection [18, 22, 62] and system knowledge extraction [5, 6]. For example, *DeepLog* [18] uses LSTM, a popular deep neural network model, to learn system log patterns and identifies anomalies when system logs deviate from the normal patterns; *CSight* [5] mines system logs to infer a model of the system behaviors; *PerfAugur* [50] is designed to find performance problems by mining service logs using specialized features such as predicate combinations; To the best of our knowledge, to date, there has been no work that uses OS-level traces for control flow recovery.

8 DISCUSSION AND THREATS TO VALIDITY

Threats to internal validity concern confounding factors that could affect the obtained results. The threat primarily lies in our implementation of existing control flow recovery methods. To mitigate this threat, for *Bunkerbuster* and *Gcov*, we replicate them by using the code released by their authors and the default configurations suggested by them; for *Pensieve*, we implement it carefully according to the description in its paper as its authors do not make the code publicly available.

Threats to external validity concern the generalizability of our experimental results. In line with the literature [47, 55], we focus on C/C++ programs and Linux, to ensure a fair comparison with existing control flow recovery methods. However, our approach is general and can be also applied to other programming languages and OSes with engineering efforts. For example, for Java, one can collect the external library calls in JVM and recover the control flow using our approach. Another threat to external validity lies in the selection of applications for evaluation. To mitigate the threat, we use seven desktop applications that are widely-adopted in the control flow recovery literature as well as three real-world IoT applications.

Research comparison between *Adonis* and log-based methods. Firstly, as shown in our RQ 1, *Adonis* achieves a higher accuracy rate because OS-level traces are more informative than logs. OS-level traces record not only I/O events, but also other program states that may be missed by logs. For instance, in the bug diagnosis example in Figure 1, the return value of `"fopen"` at line 13 is not covered by logs, but is captured by OS-level traces. Additionally, not all program failures output logs that can be used for debugging (like the one shown in our motivating example in Figure 1). By contrast, OS-level traces are more faithful and informative, as they capture a broader range of system events beyond the I/O events that logs rely on.

Secondly, maintaining OS-level traces is fully automated, whereas maintaining logging code is time-consuming and error-prone. When a program undergoes an update, *Adonis* requires fewer developer efforts to adapt to the change than log-based techniques. For *Adonis*, developers only need to regenerate the proxy lib, which is fully automated and requires negligible human effort. In contrast, log-based techniques require developers to spend significant time maintaining the logging code by adding new log printing statements and modifying outdated ones. A previous empirical study [58] found that logging code is modified in a substantial number of committed revisions (18%), indicating that maintaining logging code is time-consuming. Furthermore, 39% of log modifications in the study were made to fix inconsistencies between logs and actual execution information intended to be recorded, suggesting that maintaining logging code is error-prone. Although there are tools to enhance logging practices (e.g., Log20 [61] and LogEnhancer [59]), these tools are only semi-automated and cannot easily adapt to software updates. Consequently, developers still rely mainly on themselves to maintain logging code.

In summary, *Adonis* outperforms *Pensieve* and other log-based methods in accuracy and efficiency by leveraging automated OS-level traces that are more informative and less error-prone than manually maintained logs.

Faithfulness of reproduced *Pensieve*. The high-level idea of *Pensieve* is based on the *Partial Trace Observation*, which is "jumping directly from an event to its prior causes (without analyzing the intermediate code path)". *Pensieve* proposes an event-chaining algorithm that uses the Partial Trace Observation to reconstruct a simplified partial trace of a failed execution. We strictly follow the design of the event chaining algorithm to reproduce the Pensieve in our experiments. We implement four types of events as Pensieve has designed, including condition, location, invocation, and output events. To implement Partial Trace Observation (i.e., jumping from one event to another), we follow *Pensieve*'s design that repeatedly explains events, i.e., reasoning the prior events that could cause the current event. For example, in Figure 1, suppose we get a log message with `"file ABC does not exist."` An output event `<e1, "O", "file ABC ...">` is generated. We will then try to explain this event `e1` by searching for the statement that would output the corresponding log message. And another location event `<e2, L, printf("file %s does not exist.", savename)>` is reasoned. Then we will try to explain `e2` by

searching for branch conditions whose basic blocks dominate `L` on the control flow graph. And a condition event `<e3, C, fopen(filename, "r") == NULL>` is reasoned. Then, we will search for the statements that assign values to the variable filename and the reason for some new location events. This process is repeated until a point where the remaining unexplained events correspond to external API calls (the user input in our case).

What is more, we also realize that it is unfair to treat the event chain as the control flow recovered from *Pensieve*. So in our experiments, we also apply pair-wise symbolic execution on pairs of events to recover detailed control flow between adjacent events (note that conditions are stored in each event by design). We cannot reproduce the experiments of the original paper, because we have difficulty in compiling distributed JAVA programs (which rely on JVM) to Webassembly bytecode. Whereas we evaluate our reproduced *Pensieve* using Magma [29], a ground-truth fuzzing benchmark suite based on real programs with real bugs. The results show that the reproduced Pensieve could successfully reproduce 38/56 real bugs on 4 applications (libpng, libtiff, libxml2, libsndfile). Our results are similar to the results of *Pensieve*'s paper (13/18 bugs are successfully reproduced).

Configuration to reduce the trace size of Intel-PT. There are several ways to reduce the trace size of Intel-PT and "Filter by CR3" [31] is the most relevant and practical configuration in our case. It reduces the size by limiting Intel-PT to trace limited number of processes. To our knowledge, Bunkerbuster supports "Filter by CR3," and in our experiment, we also make it trace the targeted application's processes.

Tracing long running processes. For long running processes (e.g., an HTTP server), *Adonis* users can choose to start or stop tracing at any time (e.g., tracing a specific HTTP request), and these actions will not affect the traced program because the tracing is at the system level. So for long running processes, there is no need to record all OS-level traces for their whole life. As for the length of the trace needed to recreate a usable control flow, it depends on the complexity of the program. In our experiments, for a simple application like `"tcas"`, Adonis can recreate a usable control flow from 14 events; and for a complex application like `"sqlite3"`, it may require 1k 6k events.

Handle system calls generated by the system itself. When using the API model to infer a system call trace's corresponding function trace, *Adonis* will try to ignore `brk` and `switch` events if it cannot find a proper function to match the current system call. These system calls come from page fault exceptions and context switches and are generated by the system instead of the traced application. In this case, these system calls should be ignored. For example, suppose the current system call trace left to match (infer) is `brk, brk, open`. In this case, *Adonis* cannot find a proper function in Trie (i.e., the API model) that matches it, so it will ignore the first `brk` and try to match again. It still fails and will ignore the second `brk` until it can match the open system call.

Applications could also use these system calls, and it does happen in our evaluated applications. There are two cases when applications may use them. The first case is that the application intentionally invokes them, which we call an *intentional call* (e.g., calling `malloc()` to allocate memory will generate the `brk` system call). In this case, *Adonis* will not consider the statement that contains an *intentional call* as a checkpoint (In the function route, we cannot use these functions for path recovery as we find that hooking these functions will interfere with the execution of the program, and in syscall route, we choose not to use these events for path recovery to protect the consistency of our analysis algorithm). In other words, these *intentional calls* are treated as normal statements that will not generate system calls. And the monitored system calls generated by these *intentional calls* will be correctly ignored by the API model. The second case is that the application uses a lib function that may generate these system calls, which we call an *unintentional call*. For example, function `opendir()` could generate a series of system calls, including `openat, fstat, brk, brk`, and the `brk` here is an `unintentional call`. In this case, our API model has already covered this mapping when we construct the model, i.e., there is a 5-depth leaf corresponding to this API-to-Trace mapping (textttopendir to the 4-length trace) in the Trie. So when the current system call trace left to match is `openat, fstat, brk, brk`, the brk here will not be ignored, and *Adonis* will correctly infer this system call trace's function trace is `opendir`.

Scalability of *Adonis*. Currently, the biggest application we have evaluated *Adonis* on is sqlite3 (with 236K LoC). One of the difficulties that prevent us from evaluating Adonis on large applications is that complex applications are usually built based on build systems (e.g., GNU Make and CMake). And using these systems to compile the source code to the WASM binary that we could analyze usually produces many compilation errors, and fixing these errors is time-consuming. In the future, we plan to evaluate *Adonis* on other complex applications, such as nginx and OpenSSL.

Handle multithread executions. Our approach inherently supports multithread programs. Note that the tracing is implemented in the OS layer, so it is easy to get the thread id (tid) of each system call event. For example, Sysdig, the tool that *Adonis* uses to trace system calls, supports extracting the thread id of each system call. And the order of these events can be determined by their timestamps. Similarly, for function trace, we can use `gettid()` API to get the thread id of each lib call.

9 CONCLUSION

We have presented *Adonis*, an instrumentation-free and hardware-independent control flow recovery tool for production environments. By leveraging the informative and easy-to-collect OS-level traces, *Adonis* is able to recover crash paths from software failures under 86.8% recall and 81.7% precision. Experiments on representative desktop and IoT applications show that *Adonis* has moderate runtime, deployment, and development cost compared with existing control flow recovery techniques. Specifically, *Adonis* slows down the program by 2.78% to 3.34%, which is 18.3× lower than the instrument-based baseline. It can be deployed in desktop, IoT, and cloud environments with negligible modification so its deployment cost is practically low. The development cost of *Adonis* is also reasonable as it is fully automated and requires no extra developer efforts.

ACKNOWLEDGMENTS

This work is supported in part by the National Key Research and Development Program of China under the grant number 2020YFB2104100 and the National Natural Science Foundation of China under the grant number 62172009. Zhenpeng Chen is supported by the ERC Advanced Grant No.741278 (EPIC: Evolutionary Program Improvement Collaborators). Ding Li and Zhenpeng Chen are corresponding authors of this paper.

REFERENCES

[1] Agile-IoT. 2022. Awesome-Open-IoT, a curated list of awesome open source IoT frameworks, libraries and software. https://github.com/Agile-IoT/awesome-open-iot. [Online; accessed 21-June-2022].

[2] Taweesup Apiwattanapong and Mary Jean Harrold. 2002. Selective path profiling. *ACM SIGSOFT Software Engineering Notes* 28, 1 (2002), 35–42.

[3] Avast. 2022. RetDec. https://github.com/avast/retdec. [Online; accessed 10-June-2022].

[4] Thomas Ball and James R Larus. 1996. Efficient path profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE, 46–57.

[5] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. 2014. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering*. 468–479.

[6] Ivan Beschastnikh, Perry Liu, Albert Xing, Patty Wang, Yuriy Brun, and Michael D Ernst. 2020. Visualizing distributed system executions. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 2 (2020), 1–38.

[7] Aditya Bhardwaj and C Rama Krishna. 2021. Virtualization in cloud computing: Moving from hypervisor to containerization—a survey. *Arabian Journal for Science and Engineering* 46, 9 (2021), 8585–8601.

[8] Marcel Böhme, Ezekiel Olamide Soremekun, Sudipta Chattopadhyay, Emamurho Juliet Ugherughe, and Andreas Zeller. 2017. How developers debug software—the dbgbench dataset. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 244–246.

[9] Gianluca Borello. 2015. System and application monitoring and troubleshooting with sysdig. (2015).

[10] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep* (2013), 1.

[11] T Britton, L Jeng, C Graham, P Cheak, and T Katyenellenbogen. 2015. Reversible Debugging Software, University of Cambridge. *Judge Business School2013* (2015).
[12] Boyuan Chen et al. 2017. Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation. *Empirical Software Engineering* 22, 1 (2017), 330–374.
[13] Trishul M Chilimbi, Ben Liblit, Krishna Mehra, Aditya V Nori, and Kapil Vaswani. 2009. Holmes: Effective statistical debugging via efficient path profiling. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 34–44.
[14] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse debugging of failures in deployed software. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 17–32.
[15] Normann Decker, Boris Dreyer, Philip Gottschling, Christian Hochberger, Alexander Lange, Martin Leucker, Torben Scheffel, Simon Wegener, and Alexander Weiss. 2018. Online analysis of debug trace data for embedded systems. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 851–856.
[16] Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 270–280. https://doi.org/10.1145/1375581.1375615
[17] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 10, 4 (2005), 405–435.
[18] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 1285–1298.
[19] Yoav Einav. 2019. Amazon Found Every 100ms of Latency Cost them 1Sales. https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales. [Online; accessed 24-May-2022].
[20] Raspberrypi Forums. 2022. ARM CoreSight. https://forums.raspberrypi.com/viewtopic.php?t=192728. [Online; accessed 28-June-2022].
[21] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. Griffin: Guarding control flows using intel processor trace. *ACM SIGPLAN Notices* 52, 4 (2017), 585–598.
[22] Siavash Ghiasvand and Florina M Ciorba. 2018. Assessing data usefulness for failure analysis in anonymized system logs. In *2018 17th International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE, 164–171.
[23] GitHub. 2006. Mac OS X Man Pages - dyld(3). https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man3/dyld.3.html. [Online; accessed 10-June-2022].
[24] Francesco Giuliari, Alberto Castellini, Riccardo Berra, Alessio Del Bue, Alessandro Farinelli, Marco Cristani, Francesco Setti, and Yiming Wang. 2021. POMP++: Pomcp-based Active Visual Search in unknown indoor environments. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 1523–1530.
[25] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 213–223.
[26] Gaston H Gonnet and Ricardo Baeza-Yates. 1991. *Handbook of algorithms and data structures: in Pascal and C*. Addison-Wesley Longman Publishing Co., Inc.
[27] Honggfuzz Google. 2022. ARM CoreSight Tracing. https://github.com/google/honggfuzz/issues/63. [Online; accessed 28-June-2022].
[28] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. 2018. Studying and detecting log-related issues. *Empirical Software Engineering* 23, 6 (2018), 3248–3280.
[29] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.
[30] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2021. EOSAFE: Security Analysis of EOSIO Smart Contracts. In *30th USENIX Security Symposium (USENIX Security 21)*. 1271–1288.
[31] Intel. 2015. Real Time Instruction Trace. https://www.intel.com/content/dam/www/public/us/en/documents/reference-guides/real-time-instruction-trace-atom-reference.pdf. [Online; accessed 3-March-2022].
[32] Tong Jia, Pengfei Chen, Lin Yang, Ying Li, Fanjing Meng, and Jingmin Xu. 2017. An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services. In *2017 IEEE international conference on web services (ICWS)*. IEEE, 25–32.
[33] Tong Jia, Lin Yang, Pengfei Chen, Ying Li, Fanjing Meng, and Jingmin Xu. 2017. Logsed: Anomaly diagnosis through mining time-weighted control flow graph in logs. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. IEEE, 447–455.
[34] Michael Kerrisk. 2021. gcov(1) - Linux manual page. https://man7.org/linux/man-pages/man1/gcov.1.html. [Online; accessed 28-June-2022].
[35] Thomas D LaToza and Brad A Myers. 2010. Developers ask reachability questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 185–194.
[36] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E Hassan. 2020. A qualitative study of the benefits and costs of logging from developers' perspectives. *IEEE Transactions on Software Engineering* (2020).
[37] lifting bits. 2022. McSema, Framework for lifting x86, amd64, aarch64, sparc32, and sparc64 program binaries to LLVM bitcode. https://github.com/lifting-bits/mcsema. [Online; accessed 21-June-2022].

[38] Xuanzhe Liu, Gang Huang, Qi Zhao, Hong Mei, and M. Brian Blake. 2014. iMashup: a mashup-based framework for service composition. *Sci. China Inf. Sci.* 57, 1 (2014), 1–20.
[39] Xuanzhe Liu, Yi Hui, Wei Sun, and Haiqi Liang. 2007. Towards Service Composition Based on Mashup. In *2007 IEEE International Conference on Services Computing - Workshops (SCW 2007), 9-13 July 2007, Salt Lake City, Utah, USA*. IEEE Computer Society, 332–339.
[40] Juan Lopez, Leonardo Babun, Hidayet Aksu, and A Selcuk Uluagac. 2017. A survey on function and system call hooking approaches. *Journal of Hardware and Systems Security* 1, 2 (2017), 114–136.
[41] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. 2018. Cloudraid: hunting concurrency bugs in the cloud via log-mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 3–14.
[42] Linux manual page. 2021. ld.so(8). https://man7.org/linux/man-pages/man8/ld.so.8.html. [Online; accessed 27-May-2022].
[43] Linux manual page. 2021. perf-intel-pt. https://man7.org/linux/man-pages/man1/perf-intel-pt.1.html. [Online; accessed 24-May-2022].
[44] Linux manual page. 2021. strace. https://man7.org/linux/man-pages/man1/strace.1.html. [Online; accessed 27-May-2022].
[45] Dongliang Mu, Yunlan Du, Jianhao Xu, Jun Xu, Xinyu Xing, Bing Mao, and Peng Liu. 2019. POMP++: Facilitating postmortem program diagnosis with value-set analysis. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1929–1942.
[46] Rashmi Mudduluru and Murali Krishna Ramanathan. 2016. Efficient flow profiling for detecting performance bugs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 413–424.
[47] Peter Ohmann, Alexander Brooks, Loris D'Antoni, and Ben Liblit. 2017. Control-flow recovery from partial failure reports. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 390–405.
[48] Peter Ohmann and Ben Liblit. 2017. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. *Automated Software Engineering* 24, 4 (2017), 865–904.
[49] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 49–61.
[50] Sudip Roy, Arnd Christian König, Igor Dvorkin, and Manish Kumar. 2015. Perfaugur: Robust diagnostics for performance anomalies in cloud services. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1167–1178.
[51] Anirban Saha, Raju Udava, Mallikarjun Bidari, Mahadeva Prasad, Venkata Raju, and Tushar Vrind. 2021. TraFic—A Systematic Low Overhead Code Coverage Tool for Embedded Systems. In *2021 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*. IEEE, 1–6.
[52] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 263–272.
[53] Wikipedia. 2022. Trie. https://en.wikipedia.org/wiki/Trie. [Online; accessed 10-June-2022].
[54] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. Postmortem Program Analysis with {Hardware-Enhanced}{Post-Crash} Artifacts. In *26th USENIX Security Symposium (USENIX Security 17)*. 17–32.
[55] Carter Yagemann, Simon P Chung, Brendan Saltaformaggio, and Wenke Lee. 2021. Automated Bug Hunting With Data-Driven Symbolic Root Cause Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 320–336.
[56] Carter Yagemann, Matthew Pruett, Simon P Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. 2021. ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. 1989–2006.
[57] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*. 143–154.
[58] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 102–112.
[59] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)* 30, 1 (2012), 1–28.
[60] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. 2017. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 19–33.
[61] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 565–581.
[62] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 683–694.
[63] Zhiqiang Zuo, Lu Fang, Siau-Cheng Khoo, Guoqing Xu, and Shan Lu. 2016. Low-overhead and fully automated statistical debugging with abstraction refinement. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 881–896.

[64] Zhiqiang Zuo, Kai Ji, Yifei Wang, Wei Tao, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. 2021. JPortal: precise and efficient control-flow tracing for JVM programs with Intel processor trace. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1080–1094.