A Survey of Program Analysis for Distributed Software Systems

HAIPENG CAI, University at Buffalo, USA

Distributed software systems are pervasive today and they are increasingly developed/deployed to meet the growing needs for scalable computing. Given their critical roles in modern information infrastructures, assuring the quality of distributed software is crucial. As a fundamental methodology for software quality assurance in general, program analysis underlies a range of techniques and tools for constructing and assuring distributed systems. Yet to this date there remains a lack of systematical understandings of what have been done and how far we are in the field of program analysis for distributed systems. To gain a comprehensive and coherent view of this area hence inform relevant future research, this paper provides a systematic literature review of the (1) technical *approaches*, including analysis methodology, modality, underlying representation, algorithmic design, data utilized, and scope, (2) *applications*, with respect to the quality aspects served, and (3) *evaluation*, including the datasets and metrics considered, of various program analyses in the domain of distributed software in the past 30 years (1995–2024). In addition to knowledge systematization, we also extend our insights into the *limitations* of and *challenges* faced by current technique and evaluation designs, which shed light on potentially promising *future research directions*.

CCS Concepts: • **Theory of computation → Program analysis**; • **Computer systems organization → Distributed systems**.

Additional Key Words and Phrases: program analysis, distributed system, program representation, analysis algorithm, software quality, application technique, benchmarks, evaluation metrics, distributed program

ACM Reference Format:
Haipeng Cai. 2025. A Survey of Program Analysis for Distributed Software Systems. 1, 1 (May 2025), 41 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

A distributed software system[1] consists of loosely coupled or entirely decoupled components that are located at physically separate sites [55]. It helps users share different resources and capabilities in an integrated coherent network [121, 203]. In recent years, with continuously increasing performance and scalability demands for a growing variety of computation tasks, a rising number of distributed software systems have been designed, developed, and deployed to leverage high-performance computing infrastructure and resources that are typically decentralized, including aircraft control systems, airline reservation systems, banking/financial systems, industrial control systems, medical networks, web search, and so on. Given their paramount roles and applications in our daily lives and the business/industrial world, their quality (e.g., maintainability, performance efficiency, portability, reliability, security, usability [88, 106, 180, 209]) and corresponding quality assurance approaches are significantly important [82, 83, 85, 148, 156].

[1] Without loss of generality but for brevity, we use “distributed software system", “distributed software", and “distributed system" exchangeably throughout the paper—while a *distributed system* generally refers to a networked computing environment, this paper focuses on the software perspective.

Author’s Contact Information: Haipeng Cai, haipengc@buffalo.edu, University at Buffalo, Buffalo, New York, USA.

Manuscript submitted to ACM

As a primary form of software analysis, program analysis is an automatic technique or process of analyzing the behaviors hence properties (e.g., correctness, robustness, safety) of software systems based on their programs [166]. There are generally three classes of program analysis approaches, categorized according to the kinds of data utilized (i.e., actually analyzed) by a program analysis technique/algorithm: *static analysis* reasoning about program behaviors based on the program code with respect to any possible program executions, (purely) *dynamic analysis* computing run-time properties of specific program executions with respect to particular program inputs, and *hybrid analysis* combining static and dynamic analysis, which may also be considered a kind of dynamic analysis that utilizes static analysis results. Historically, program analysis of any of these classes has been the enabling technique for many fully- or semi-automated application solutions to various software quality assurance issues and challenges.

Particularly for distributed software systems, program analysis remains a fundamental methodology for software assurance. As for other software domains, this methodology has empowered typical software engineering and systems tasks, such as fault diagnosis, code optimization, performance tuning, security defenses, and various maintenance/evolution activities [17, 115, 120, 224]. In fact, program analysis for distributed software systems as a research area/topic can be dated back to 1980s [65, 66, 157], when the need for testing and debugging distributed systems was recognized in distributed/parallel computing environments. While much of the work in the broad area of parallel and distributed computing has been done from coarse, system-level perspectives such as those of architecture, networking, and resource management, studies on code-level quality of distributed software via program analysis also went through a long journey. These studies include both fine-grained, deeper program modeling and reasoning and those at coarse levels. For instance, techniques have been devised to resolve dependencies in distributed systems for enhancing parallelization [181], system configuration [119], and high-level system modeling [1, 23]. And the underlying program analyses include static [91, 161, 179], dynamic [38, 39, 79, 83], and hybrid [30, 82] ones.

Yet despite its long history and great significance, the area of program analysis for distributed software (i.e., analysis of *distributed programs*[2]) has not been systematically surveyed. As demonstrated in such surveys of program analysis for non-distributed systems (i.e., software running on one single computer, regardless in a single process or multiple processes) [53, 223] as well as many other areas in computing, a literature review can serve many **beneficial purposes**. First, for researchers in software engineering, programming languages, and systems, especially those who just get into respective areas, a comprehensive survey offers an overview of what progresses have been made and what approaches are available. For practitioners and distributed system researchers who do not have core expertise in program analysis, this overview can be very valuable in helping them navigate through the literature for technique/tool selections for immediate use or developing domain-specific solutions. Second, the survey can serve as systematization of knowledge on program analysis for distributed software, summarizing challenges and limitations faced by existing techniques, hence guiding both researchers and practitioners in discerning the strengths and weakness of different choices. Third, the survey is essential for distilling insights into this topic regarding how far it is now and what gaps remain, hence informing future research to take the most pressing problems in the right directions.

A few prior works summarized different program analysis techniques for *specialized* distributed systems. For example, Alqahtani et al. [10] summarized of cloud system data security via information flow control. It first discussed how different techniques were used with the CloudMonitor tool that guarantees the data protection of cloud systems, followed by an overview of the operations of some information flow control systems, and the advantages and disadvantages of those approaches. Also, Wei et al. [221] provided a survey on data-flow management, concerning sensing, control,

[2] We refer to the code components of a distributed system together as a *distributed program*, where a *component* is the code running in a separate *process*.

and security, in Internet of Things (IoT) systems. It started with the key challenges facing IoT data-flow management, and then summarized relevant techniques in this area, with illustrations via representative management tools and/or platforms as well as example application scenarios (e.g., smart city/transportation/manufacture). However, there has been no systematic literature review on program analysis *for distributed software* in general (i.e., both common-purpose and specialized systems), despite the aforementioned benefits of such a survey and the importance of this topic.

```
//NioEcho client component/process
public class NioClient implements Runnable {
  public void send(...) ... {
    SocketChannel socket = this.initiateConnection();
    this.rspHandlers.put(socket, handler);
    synchronized (this.pendingData) {
    List queue = (List) this.pendingData.get(socket); ...
    this.pendingData.put(socket, queue);
    queue.add(ByteBuffer.wrap(data)); }
    this.selector.wakeup();     } ... }

  public static void main(String[] args) {
    String host = "localhost";
    int port = 9090;
    ...
    NioClient client = new NioClient(InetAddress.getByName(host),
    port);
    Thread t = new Thread(client);
    t.setDaemon(true);
    t.start(); ... }
......
//NioEcho server component/process
public class NioServer implements Runnable {
  public void run() {  ...
    synchronized (this.pendingChanges) {
    Iterator changes = this.pendingChanges.iterator();
    while (changes.hasNext()) { ...
      SelectionKey key = change.socket.keyFor(this.selector);
      key.interestOps(change.ops);       ... }}

  public static void main(String[] args) {
    String host = "localhost";
    int port = 9090;
    if (args.length > 0) {
      port = Integer.parseInt(args[0]);    }
    if (args.length > 1) {
      host = args[1];    }
    InetAddress serveraddr = InetAddress.getByName(host); ... }
```

Fig. 1. An example distributed program NioEcho [197] for illustration.

To fill this gap, we conduct *a comprehensive review of program analysis for distributed software systems* across 30 years (1995–2024), addressing the long missing knowledge systematization on what is unique with the program analysis for this particular software domain. To offer a systematic understanding of the status quo, we examined three high-level, closely connected aspects of distributed-program analysis: (1) the technical approach (i.e., analysis method), (2) the application problem addressed by and used for evaluating the technical approach (i.e., specific software quality problem the program analysis technique is applied to), and (3) the evaluation approach (i.e., experiment design) followed for assessing the technique in respective application contexts. In examining technical approaches, we take an extensive list of angles relevant to program analysis in general, including the underlying program representation, analysis modality and scope, algorithmic design, and data utilized by the analysis.

Paper organization. Specifically, we start with necessary background on program analysis and distributed systems (§2), followed by the survey guided by a principled methodology (§3,§4,§5). This methodology covers all the common major steps of a systematic literature review, including (1) the literature search process (§4) which resulted in the collection of papers that serve as the basis of survey, (2) the derivation of taxonomy (§5) of program analysis for distributed systems which resulted in a paper-attribution framework, and (3) the paper attribution process that follows the framework to produce our survey results (§6). Based on these results, we discuss the limitations and challenges with existing distributed-program analysis techniques/tools surveyed (§7.1), from which we further identify future research directions for overcoming those limitations/challenges in the field of program analysis for distributed software (§7.2). Finally, we discuss the limitations of our survey itself (§8) before making brief concluding remarks (§9).

2 Background

This section provides essential background on program representations. Additional information on program analysis and distributed systems can be found in Appendix A.

An analysis of a program is often performed on a representation of the program. Then, reasoning about the program behaviors is based on that representation. Among others, *graph* representations are the most commonly used by various

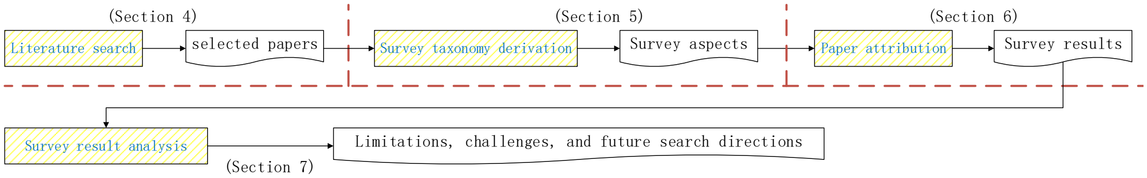

Fig. 3. Overview of our survey methodology and workflow.

analysis techniques. As examples, we briefly introduce three basic graph representations as follows: *control flow graph*, *call graph*, and *program dependence graph*, while mentioning a few other related ones. We use code excerpts from (the version r69 of) NioEcho (shown in Figure 1) for illustration purposes. This is a simple distributed program consisting of one server component and a number of client components, where the server echoes any messages sent by a client [197].

A control flow graph (CFG) [7] is a graph representing the control flow relationships among code entities in a program, often used in static analyses, such as those in a compiler. For example, the CFG of the method main of the NioEcho server (class NioServer, Lines 31–37) is depicted in Figure 2. Moreover, in this case, Line 31 is the method *entry*, Line 37 is the *exit*, and Lines 35 and 37 are branch statements each including one conditional referred to as *predicate* (e.g., args.length>0).

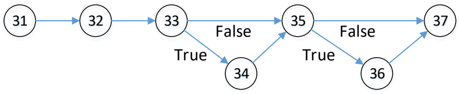

Fig. 2. An example control flow graph (CFG)—of the main method in the example distributed program NioEcho's NioServer class (Figure 1)

A call graph (CG) is a directed graph that represents calling relationships among program subroutines (e.g., functions/methods) [187]. Each CG node represents a subroutine, and each CG edge <*n1,n2*> connecting two nodes *n*1 and *n*2 represents that *n*1 calls *n*2. In the program NioEcho of Figure 1, for example, NioServer::run calls SelectionKey::interestOps; thus, an edge between the two nodes representing the respective methods is included in the call graph of the server component of this distributed program.

A fundamental way of modeling a program's behavior is by modeling the dependencies among code entities of the program, and one basic form of such models is program dependence graph (PDG). Each node of this graph represents a code entity (e.g., statement) while an edge between two nodes represents either a data or control dependence between the two respective entities that the two nodes represent. For example, in the main method of NioEcho's class NioClient, Line 17 is *data dependent* on Line 16 as the former defines (writes to) the variable client while the latter uses (reads) the same variable without any intermediate redefinition (overwrite) of it. A control dependence exists between two statements when the evaluation result of one statement (e.g., Line 35) determines whether the other (e.g., Line 26) executes or should be bypassed [9].

Per the original definition [74], PDG is *intraprocedural*—it consists of dependencies *within* a function/method. Horwitz et al. [102] extended the PDG to an *interprocedural* representation of a program that consists of more than one function/method, called a *system dependence graph* (SDG), which captures the calling context of data/control dependencies within a PDG [140]. Given such a program, its SDG can be constructed from its per-method/function PDGs while referring to its *interprocedural control flow graph* (or ICFG)—by default, a CFG is also intraprocedural. The ICFG itself further relies on the call graph to be constructed from the per-method/function CFGs of the program.

Another important representation of programs is information flow graph (IFG) [60], a graphical representation used in program analysis to model how information flows through a software system. In an IFG, nodes represent program variables, memory locations, or other entities where information can be stored (while edges represent the flow of information between nodes), versus nodes in a CFG/ICFG/PDG/SDG representing statements/instructions.

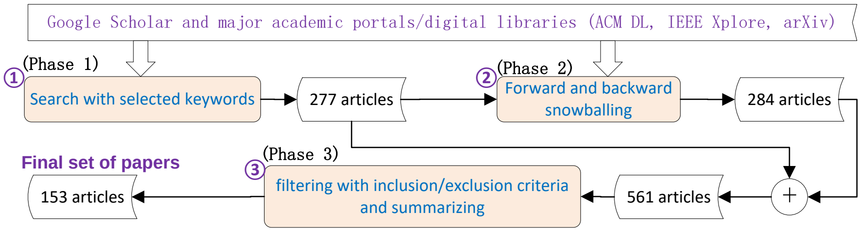
Google Scholar and major academic portals/digital libraries (ACM DL, IEEE Xplore, arXiv)
1
(Phase 1)
Search with selected keywords
277 articles
2
(Phase 2)
Forward and backward snowballing
284 articles
Final set of papers
153 articles
3
(Phase 3)
filtering with inclusion/exclusion criteria and summarizing
561 articles
+

Fig. 4. Overview of our literature search process, including its three phases and respective inputs and outputs.

3 Overview of Survey Methodology

To gain systematization and insights about program analysis for distributed systems, we conducted a comprehensive survey on this topic following a principled methodology, as shown in Figure 3 and summarized below.

Starting with a systematic *literature search* (§4), we first searched and selected papers (using chosen, relevant keywords). From these selected papers, we proceeded with the *survey taxonomy derivation*, resulting in the survey aspects/attributes/items (§5) that form an attribution framework. Then, following the *paper attribution* process [53] we attributed the surveyed papers within the taxonomy, hence producing the survey results (§6). Lastly, through the *survey result analysis*, we identified the limitations/challenges in the field of program analysis for distributed software systems, while offering our views on relevant future research directions for addressing those limitations and challenges (§7).

4 Literature Search

We searched, identified, and filtered studies related to program analysis for distributed software in three phases, as shown in Figure 4. Next, we describe each of these phases in detail.

4.1 Phase 1

In this phase, we searched broadly for all potentially relevant research papers on Google Scholar and well-known academic publication portals/digital libraries such as *ACM digital library (DL)*, *IEEE Xplore*, and *arXiv*. We used the following topic-relevant **keywords** and their combinations as queries: *distributed software, distributed system, distributed program, distributed/parallel computing, cloud system, program/code analysis, static/dynamic analysis, dependence analysis/slicing, taint analysis, information flow, quality/security/reliability/performance/dependability*, and *internet of things*. We chose these keywords because they are indicative of the software domain (i.e., distributed system) and technical topic (i.e., program analysis) of our survey, or common types of program analysis (e.g., dynamic analysis) and their typical applications (e.g., quality). And then we scooped a conservative, comprehensive list of publications related to any of these queries, resulting in the initial set of 277 papers. *We aimed to cover any papers that are relevant to our survey domain and topic, regardless of where they were published and when.*

4.2 Phase 2

To ensure comprehensive inclusion of relevant papers, we went further to scoop additional related papers that are not in our initial list (i.e., the result of *Phase 1*) through manual forward and backward snowballing processes (i.e., including papers that cite, and are cited by, any of the 277 papers, respectively). These processes led us to 284 additional papers that are potentially relevant according to the inclusion of any of the keywords (same as used in *Phase 1*) in the paper title or body. As a result, our initial paper pool size grows to 561 (=277+284).

4.3 Phase 3

This phase is our manual inspection and filtering step to select the relevant papers, for which we used the following inclusion and exclusion criteria. In particular, we defined two inclusion criteria as follows:

- Studies about any kind of program analysis approach or its application, either technical or empirical.
- Studies that address any kind of distributed software systems, including common and specialized ones (e.g., event-based, cloud, and Internet of Things (IoT) systems, micro-services), as the subject program under analysis.

Meanwhile, to speed up the filtering process, we also defined the following exclusion criteria:

- Studies that do not involve development, use, or evaluation of any form of program analysis.
- Studies about single-process programs, either single-threaded or multi-threaded ones.
- Materials that are not official academic/research publications (e.g., abstracts, posters, and tutorials)
- Papers that are not written in English.

For each of the 561 papers in our pool resulting from *Phase 2*, we carefully looked into the paper, inspecting its title, keywords, abstract, and the main text. We immediately dismissed the paper if it met any of the exclusion criteria; otherwise, we used the two inclusion criteria to decide on inclusion or not and only kept the paper if it met both. In the end of this phase, we selected **153** papers as the basis of our next survey steps, for which we manually summarized and recorded about the following information for each included paper: the problem addressed, motivation, approach, application, evaluation subjects, evaluation metrics, evaluation results, strengths and limitations, and future work—if explicitly discussed or implicitly retrievable in the paper.

5 Survey Taxonomy Derivation

In this section, we describe the angles in which we examine the existing literature on program analysis for distributed software. We start with the process of deriving all of these angles, which collectively form our survey taxonomy (§5.1, §5.2). Then, we present the derivation results—i.e., the taxonomy itself (as outlined in Table 1), which covers all the three high-level aspects of a program analysis: the technical approach (§5.3.1), the application of the analysis (§5.3.2), and the evaluation of the analysis technique—either directly or through the application (§5.3.3).

Table 1. The Derived Taxonomy of Our Survey on (i.e., Examined Aspects of) Program Analysis for Distributed Software

Aspect	Attribute	Item	Description
Approach	Analysis Methodology	Code-based	analyzing the given program based solely on its code
		Learning-based	using machine/deep-learning technique(s)
	Analysis Modality	Static	analyzing the given program without executing it
		Dynamic	using the program's execution/run-time information
	Program Representation	CFG/ICFG	representing control flows
		CG	representing calling relationships among functions
		PDG/SDG	representing data and control dependencies
		IFG	representing information flows
	Algorithmic Parameter	Analysis Sensitivity	if the analysis is context-/flow-/field-/object-/path-sensitive
		Data Granularity	the granularity of the data used
	Analysis Data	Run-time Event	run-time/execution events of the program
		System Log	log generated by the system during its operation
		Artifact	(code/non-code) items produced during software development
	Analysis Scope		targeted distributed-system types (common or specialized)
Application	Functional Testing		verifying/validating software functionalities
	Fault Localization		identifying faulty program entities/locations
	Security Support		aiming to assure security requirements
	Performance Diagnosis		diagnosing performance efficiency
	Maintenance/Evolution Support		enabling/facilitating software maintenance/evolution tasks
Evaluation	Dataset	Benchmark Suite	systems used purposely for comparison/measurement
		Real-world System	distributed systems used by real end-users
	Metric	Effectiveness	how effective (e.g., accurate/precise) the analysis is
		Efficiency	how (e.g., time/space) efficient the analysis is

5.1 Attribute/Item Identification

Our attribute identification step aims to refine each of the three aspects (*Approach*, *Application*, *Evaluation*) into specific attributes. First, we studied all the articles and wrote down relevant words as attributes for particular aspects, such as "Analysis Methodology" or "Analysis Modality" for the "Approach" aspect. Then, we recorded words of interest as items that could be related to particular attributes, such as "static" or "dynamic" for the "Analysis Modality" attribute.

5.2 Attribute/Item Generalization

After the initial attributes and items have been identified, we generalize them to render their numbers manageable and improve their reusability, through a few iterations. For example, regarding the target aspect "Application", the attributes "system testing" and "unit testing" can intuitively be generalized to "Functional Testing". Similarly, regarding the target attribute "Algorithmic parameter“ (of the "Approach" aspect), the items "context-sensitivity", "flow-sensitivity", "field-sensitivity", and "object-sensitivity" can intuitively be generalized to "Analysis Sensitivity". After this generalization step, the resulting attributes and items are documented.

5.3 Resulting Taxonomy

Following the attribute/item identification and generalization steps, we derived our taxonomy (as outlined in Table 1), as elaborated below for each of the three high-level aspects separately.

5.3.1 Approach. In this aspect, we identified six attributes. In terms of **Analysis Methodology**, the existing approaches fall in two major categories: Code-based, where the analysis directly reasons about the given program code in a deterministic manner, and Learning-based, where the analysis leverages machine/deep learning methods.

The approach aspect also differentiates two general types of **Analysis Modality**: *static* and *dynamic*. A static approach performs the analysis using program information with respect to all possible executions of the program (i.e., without executing it). While consuming fewer resources (when considering those needed for program executions), static analysis cannot *precisely* capture the behaviors of a program during its concrete execution. In contrast, a dynamic approach utilizes run-time program information obtained from the program's concrete execution, including *(purely) dynamic analysis* which only uses the run-time information and *hybrid dynamic analysis* which additionally utilizes static program information. Dynamic analysis enables reasoning about the run-time behaviors hence validating program properties or just understanding the system better [53]. While generally incurring higher overhead [34], it often offers greater precision [31, 36]. In particular, hybrid dynamic analysis helps gain better balance between the precision and overhead of the analysis [28, 32, 37], although it tends to be more complex and need more resources. Since hybrid (dynamic) analysis is essentially a kind of dynamic analysis, we do not treat it as a separate analysis modality.

As for other domains of (e.g., centralized) software systems, analyses for distributed systems commonly work on the basis of a certain kind of **Program Representations**. In particular, this attribute differentiates four types of representations: *CFG/ICFG*, *CG*, *PDG/SDG*, and *IFG*, as defined/described earlier (§2). Control flow graph (CFG) represents the control flow of a single procedure (i.e., function) [7], while interprocedural control flow graph (ICFG) connects all of the CFGs of a program hence representing the control flow of the entire program. Call graph (CG) represents calling relationships among functions in a program [187]. Program dependence graph (PDG) models the data and control dependences among program entities within a function, while system dependence graph (SDG) captures those dependencies across functions. Information flow graph (IFG) represents the information flows of a program, where each flow is often represented by a chain of data/control dependencies between an information source and an information sink. Each of these program representations may be constructed at different granularity (e.g., instruction, statement, method) levels.

The **Algorithmic Parameter** attribute of distributed-program analysis distinguishes two major items (sub-attributes): the sensitivity of the analysis and the granularity of data used by the analysis. More specifically, Analysis Sensitivity concerns whether the analysis is context-, flow-, field-, object-, or path-sensitive. These sensitivity properties commonly concern the differentiation between different execution scenarios when all possible executions need to be considered—thus, they are usually only relevant to a *static* analysis. Enabling (disabling) a sensitivity property often implies a greater (lower) level of precision at a higher (lower) level of cost; therefore, setting this algorithmic parameter immediately affects the cost-effectiveness of the analysis [186]. Similarly, Data Granularity also has a clear effect on the balance between analysis cost and effectiveness—usually, finer (coarser)-grained data bring higher (lower) precision while incurring greater (lower) overhead. In our survey, this algorithmic parameter is only relevant to dynamic analysis, which uses run-time data such as method-level execution events or statement-level coverage records.

Another key attribute of a program analysis approach for distributed programs (as for program analysis in general) is the **Analysis Data**. This attribute further distinguishes three types of data used by the analysis as follows. Run-time Event refers to execution events that externalize program run-time behaviors, such as method entry/exit [32], message receiving/sending [39], and statement coverage [30]. These events are usually captured through code instrumentation. System Log is another major kind of analysis data, which is produced through logging facilities provided by the developers in the original program, rather than being inserted to the program via instrumentation. Analysis can often be enriched by using non-code Artifact as well, such as README and manifest files, configuration/installation documents, and commit logs and code comments, etc.

Finally, given the presence of different types of distributed systems, a program analysis for distributed programs often has a targeted **Analysis Scope**. The relevant literature has addressed mainly two analysis scopes. The first is *Common Distributed System*, which uses standard networking facilities (e.g., Socket-based message passing) for the interaction between distributed components of the system, as defined in a commonly-used textbook on distributed systems [55] (§A.2). The second subsumes all sorts of specialized distributed systems, such as distributed event-based (DEB) [160], cloud systems [152], and Internet of Things (IoT) systems [218] (§2).

5.3.2 Application. In this aspect, we examine the concrete application of a program analysis for distributed programs to solving a specific problem. From the current relevant literature, we derived five types of applications, according to the common aspects of software quality. In particular, **Functional Testing** addresses any program analysis being applied to verifying/validating functional correctness of distributed software, while **Fault Localization** addresses applications on identifying faulty code locations (i.e., debugging). **Security Support** focuses on any technique/tool support for securing distributed systems, concerning their *confidentiality*, *integrity*, *accountability*, *non-repudiation*, and *authenticity*. According to the ISO/IEC 25010 [106] software quality model, these are the five sub-characteristics of the *security* quality characteristic. Many software systems are designed as distributed ones due to higher-level performance goals. Thus, intuitively **Performance Diagnosis** of distributed systems themselves is an important application of distributed-program analysis, which aims at accurate identification of actual/desired performance levels and/or specification of interventions to improve performance [211]. **Maintenance/Evolution Support** is another major application area of program analysis of distributed programs, which addresses enabling/facilitating activities for maintaining [212] the distributed system and those for continual development that changes the system's functionality or properties experienced by the system customer(s) [46].

5.3.3 *Evaluation.*

To systematically understand the existing works on program analysis for distributed systems, it is essential to also look at how the technical approach has been evaluated. As for evaluation of a technique, through the evaluation aspect we examine two common attributes: the *Dataset* used in the evaluation studies and the *Metric* in which evaluation results are measured.

The **Dataset** attribute distinguishes the following two main types of evaluation subjects (and the accompanying test inputs if the analysis follows a dynamic approach). To evaluate in terms of metrics that rely on ground truth, standard or commonly-adopted Benchmark Suite is often used, which is a collection of benchmarks that are test cases or sets of test cases purposely aimed to enable rigorous assessment of the proposed technique and/or comparisons between alternative techniques. To assess generalizability and practicality, however, it is desirable to use Real-world System (and their run-time inputs in real operations) as study subjects, which gives more confidence to users regarding how the proposed technique may perform in practice.

Regarding **Metric**, the literature we studied typically seek to measure the Effectiveness and/or Efficiency of analysis techniques/tools in the evaluation against distributed software. Here effectiveness concerns efficacy of the technical solution (to the targeted application problem), commonly quantified in terms of specific measures like precision and recall (or false/true positive rates). Efficiency concerns how much (e.g., CPU, GPU) time and/or how many resources (e.g., peak memory, network bandwidth, external storage) are consumed by the proposed approach. For dynamic analysis in particular, run-time overhead (e.g., measured via slowdown ratio) is also usually considered in efficiency evaluation.

6 Paper Attribution

The next phase of our survey process is to apply the attribution framework derived (in §5) to the relevant literature identified (in §4). In this section, we focus on summarizing the surveyed papers as per (i.e., mapping them to) our taxonomy of program analysis for distributed software. We start with statistics on the collected papers based on their venues (*S*6.1) and survey aspects/attributes (§6.2), followed by the summary of the research body in each of the three aspects: Approach (§6.3), Application (§6.4), and Evaluation (§6.5).

6.1 Paper Distribution by Venues

The 153 relevant papers were published between October 1995 and July 2024 across over 90 venues, including 15 conferences and 9 journals each covering 2 papers or more surveyed, institutions of 7 PhD dissertations, as shown in Table 2. The 71 miscellaneous other venues each covering only one paper, thus not enumerated.

Overall, related to program analysis techniques/tools for distributed software, these 153 papers are unevenly distributed across four main research areas: (1) *programming languages and software engineering*, including 7 venues: ICSE, FSE, ASE, PACMPL, TOSEM, TSE, and JSS, which contributed a total of 32 papers; (2) *operating systems and parallel/distributed computing*, including 9 venues: EuroSys, SoCC, ATC, SOSE, OSDI, SOSP, DEBS, JCC, and TPDS, which contributed 25 papers; (3) *computational science*, including 5 venues: CSCI, ICCCI, JPCS, Access, and IASC, which contributed 11 papers; and (4) *computer security*, including 3 venues: DSN, CCS, and USENIX Security, which contributed 7 papers. This venue distribution can be explained by several reasons:

- Program analysis is one of the most important and popular topics in the general area of *programming languages and software engineering*; thus, intuitively a good number of papers in our survey scope have been published in this area.
- Program analysis is also a main type of technical approach in the area of *computer security*, widely utilized to address security problems; this leads to a noticeable presence of security venues among the surveyed papers.

Table 2. Distribution of Publication Venues of Surveyed Papers

Type	Acronym	Description	# Papers
Conference	ICSE	IEEE/ACM International Conference on Software Engineering	11
	FSE	ACM Symposium on the Foundations of Software Engineering	7
	EuroSys	European Conference on Computer Systems	6
	SoCC	ACM Symposium on Cloud Computing	4
	USENIX ATC	USENIX Annual Technical Conference	3
	CSCI	Computational Science and Computational Intelligence	3
	DSN	IEEE/IFIP International Conference on Dependable Systems and Networks	3
	CCS	ACM SIGSAC Conference on Computer and Communications Security	2
	SOSE	International Conference on Service-Oriented System Engineering	2
	OSDI	USENIX Symposium on Operating Systems Design and Implementation	2
	SOSP	ACM Symposium on Operating Systems Principles	2
	ASE	ACM/IEEE International Conference on Automated Software Engineering	2
	DEBS	ACM International Conference on Distributed Event-Based Systems	2
	ICCCI	International Conference on Computer Communication and the Internet	2
	USENIX Security	USENIX Security Symposium	2
Journal	PACMPL	Proceedings of the ACM on Programming Languages	5
	TOSEM	ACM Transactions on Software Engineering and Methodology	3
	JPCS	Journal of Physics: Conference Series	2
	IASC	Intelligent Automation & Soft Computing	2
	JCC	Journal of Cloud Computing	2
	JSS	Journal of Systems and Software	2
	Access	IEEE Access	2
	TPDS	IEEE Transactions on Parallel and Distributed Systems	2
	TSE	IEEE Transactions on Software Engineering	2
Thesis	PhD Dissertation	Various institutions (each having only one thesis among the surveyed)	7
Various other venues		Various venues (each having only one paper among the surveyed)	71
All venues		Total	153

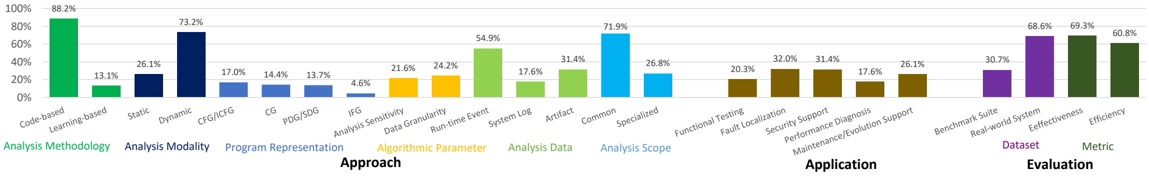

Fig. 5. Distribution of the surveyed papers over the three aspects (approach, application, and evaluation) and attributes/items.

- The others, *OS and parallel and distributed computing* and *computational science*, are primary areas addressing the subject of distributed software systems; hence, multiple relevant venues contributed to the studied literature.

Meanwhile, the fact that these 153 papers were spread quite thin (over 90 venues) reveals the wide attention and interest of broad research communities to/in the surveyed topic. On the other hand, ICSE and FSE dominates the current literature (contributing 18 papers), followed by EuroSys and SoCC (accounting for 10 of the papers). A plausible reason is that ICSE and FSE have long been a primary software-engineering (SE) venue publishing significant advances in program analysis, while EuroSys and SoCC are flagship venues on *cloud computing* which is a key application domain of distributed systems. Also, the vast majority of the papers are presented in conferences rather than journals, a contrast common in computer science in general. The 71 venues not listed are mainly in SE, computer networks, and high-performance computing.

6.2 Paper Categorization by Taxonomy

Figure 5 shows the distribution of the target literature over the derived survey aspects, attributes within each aspect, and items (when available) within each attribute. The results reveal dominating subcategories within respective categories among the 153 papers. For instance, the vast majority of these papers present code-based program analysis approaches in terms of their analysis methodology—machine learning has yet to be widely exploited for program analysis for distributed systems, and most of these approaches are dynamic analysis using various run-time events far more often than using system logs while mainly addressing common distributed systems as opposed to specialized ones. For another example, fault localization has been the primary application area of distributed-program analysis, followed by security support and maintenance/evolution support. In terms of evaluating these approaches, real-world systems have been used much more often than benchmark suites, while effectiveness is more attended than efficiency when it comes to evaluation metrics. Note that for some (sub)categories (e.g., *Application*), the percentages sum up over 100% because a paper may fit more than one (e.g., it addresses multiple application tasks); while for some (e.g., *Program Representation*) the total does not add up to 100% because some papers cannot be attributed to any of them (e.g., papers on purely dynamic analysis do not use any of the program presentations—which are all static).

In the following three subsections, we look into each of these categories (along with their subcategories), examining the actual research presented in the surveyed papers, with one subsection dedicated to each of the three high-level survey aspects. To offer an overview of our results, Table 3 (years of 2022-2024), Table 4 (years of 2020-2021), Table 5 (2015-2019), and Table 6 (1995-2014) in Appendix B show the mapping of these papers (*as linked in 1st column*) to the survey taxonomy (*all the other columns to the right*) we derived.[3] These tables serve as useful references for seeking relevant articles on particular topics addressing program analysis techniques/tools for distributed software. For example, when a user looks for dynamic analysis techniques/tools supporting the security of distributed software, he or she only needs to identify the articles with the item "Dynamic" of the attribute "Analysis Modality" within the aspect "Approach" and the item "Security Support" within the aspect "Application", and so on.

6.3 Survey Result: Approach

We now present our summary of existing distributed-program analysis approaches along the six attributes identified in our survey taxonomy, as separately examined as follows.

6.3.1 Analysis Methodology. Existing program analysis approaches for distributed systems have adopted two analysis methodologies, predominantly *Code-based* and relatively a few *Learning-based*,

Code-based. As depicted in Figure 5, by far most (88%) of the current analysis techniques/tools for distributed systems are based on the system's code. In particular, the majority of these code-based studies [2, 21, 24, 47, 50, 89, 90, 98, 107, 119, 143, 146, 150, 151, 159, 162, 163, 165, 175, 178, 183, 188, 193–195, 199, 202, 206–208, 213, 220, 232] were based on distributed program *source code*. For example, Acay et al. [2] proposed `Viaduct`, a system that allowed users to specify security policies by annotating distributed programs with information flow labels and to enforce these policies by compiling high-level source code in order to secure the programs.

Some studies [70, 93, 145, 147, 211, 231, 237] worked on the *bytecode* of distributed programs. For instance, Zhao et al. [237] presented a non-intrusive request flow profiler named `lprof`, performing static analysis on distributed systems' bytecode to stitch together and interpret the log messages of requests.

In addition, a number of analysis approaches [30, 38, 39, 76, 77, 79–84, 86, 91, 96, 130, 135–137, 200] used *intermediate representation (IR) code*, including Soot (a Java analysis framework) [214] IR [30, 38, 39, 76, 77, 79–84, 91, 96] , WALA (a Java/JavaScript analysis framework) [205] IR [130, 135–137, 200], and LLVM (a multilingual compiler framework) [126] IR [96, 206, 207]. For example, Gu et al. [96] proposed a static analysis engine, named `BigSpa`, taking the LLVM IR and Soot IR as its inputs for generating intra-component/process graphs of distributed programs written in C/C++ and

[3] Our survey covers papers published and searchable online up to July 2024; thus, results for 2024 do not represent all relevant works in that entire year.

Java, respectively. As another example, Suminto et al. [200] presented their cascading outage bug elimination (COBE) technique, which leverages WALA for initial program parsing.

Learning-based. A few analysis approaches (for distributed software) are based on machine learning techniques, including *deep learning* and *traditional machine learning*. For instance, both [236] and [108] utilized *deep learning* techniques. In particular, Zhang et al. characterized the suitability of a new parallel deep-learning model selection/execution method called Model Hopper Parallelism (MOP) for data systems and performed a comparative analysis of various distributed deep learning approaches for that purpose [236]. Similarly, Jia et al. delivered insights and guidelines on how to fully exploit distributed deep learning clusters for deploying deep learning applications [108]. Yu and Zhang [230] combines graph convolutional network (GCN) and LSTM to learn the spatio-temporal dynamics in cloud systems for anomaly detection. LSTM has also been used in [57] to facilitate the performance modeling of cloud and edge systems. Recently, He et al. [100] uses Transformer with a novel *anomaly attention* mechanism and a graph neural network (GNN) to learn spatio-temporal features to enable effectively detecting anomalies in distributed systems. With their generalizability merits, large language models (LLMs) have emerged as the key enabler of state-of-the-art learning-based techniques. For example, RCACopilot [48] integrates LLMs with automated incident handlers to streamline cloud incident Root Cause Analysis (RCA). The system first collects focused diagnostic information through customized workflows, then leverages GPT-3.5/4 to analyze this data, predict root cause categories, and provide explanations. As another example, FaultProfIT [103] employs hierarchical textual classification and hierarchy-guided contrastive learning to automatically profile fault patterns in incident tickets for cloud systems. The approach learns meaningful representations with limited training data by focusing on similarities and differences between samples, using an optimized BERT model to encode the raw text of incidents.

Other studies [4, 15, 83, 84, 87, 138] exploited *traditional machine learning* techniques. In [83, 84], Q-leaning (a type of reinforcement learning) algorithms were utilized to adjust analysis configurations [61] to achieve scalable and cost-effective dynamic dependence analysis for distributed software systems. Meng et al. [156] also leverages Q-learning but for guiding a greybox fuzzer to select which fault to inject in the system state based on current observations of distributed system execution. Astekin et al. [15] proposed a case study for evaluating distributed machine learning algorithms that could detect the logs of large-scale systems. Fukuda et al. [87] also presented a computational framework for distributed data analysis, applying traditional machine learning techniques (k-means and k-nearest neighbors (KNN)) to clustering and classification tasks. Prism [139] proposes a coarse-to-fine clustering based approach to identifying hidden functional clusters in large-scale cloud systems. DistFax [85] trains unsupervised (e.g., KNN-based) and supervised (random-forest-based) models to predict quality anomalies in common distributed system executions.

6.3.2 *Analysis Modality.*

Both *static* and *dynamic* analyses are present in the current literature of program analysis for distributed systems, with dynamic approaches clearly dominating over static ones (Figure 5).

Static. For distributed software, static approaches have been applied to various, well-known types of program analyses, including *data flow analysis* [41, 91, 96, 154, 200], *taint analysis* [189, 206, 207, 219], and *pointer analysis* [96]. For instance, Gu et al. [96] proposed a static analysis engine, `BigSpa`, performing data flow analysis and pointer analysis. Wang et al. [219] scaled static taint analysis to industrial service-oriented architecture (SOA) applications. MPCChecker [142] uses a static data flow analysis to detect missing-permission-check (MPC) vulnerabilities in cloud systems.

Dynamic. Most (73%) of the surveyed analysis techniques/tools (for distributed systems) are dynamic. In particular, many studies [2, 12, 13, 19, 24, 30, 38, 39, 76, 77, 79–84, 98, 107, 112, 130, 135–137, 146, 147, 158, 159, 165, 175, 179, 195, 199, 211, 213, 215, 237] adopted a *hybrid* approach to the dynamic analysis, rather than being purely dynamic. For example, Mohapatra et al. [159] presented a hybrid slicing technique, `DSDJ`, for distributed Java programs; it statically constructs the distributed program dependence graph (DPDG) prior to its run-time analysis—it is ultimately a dynamic analysis. Maruf et al. [5] combines system traces, quality metrics, and log data to enable (hybrid) dynamic analysis of micro-services for detecting quality anti-patterns in them. Another example is seen in Kim et al. [117], where concolic execution is used for detecting/localizing XSS vulnerabilities in web-based IoT services based on dynamic taint analysis.

In other studies [6, 11, 15, 21, 22, 44, 45, 47, 50, 62, 70, 89, 93, 97, 99, 104, 108, 111, 114, 116, 119, 125, 127, 143, 145, 151, 155, 162–164, 172, 174, 177, 178, 185, 188, 189, 193, 194, 202, 208, 212, 220, 227, 229, 231–234, 236], *purely dynamic analysis* (i.e., without any assistance of static analysis) was employed. For instance, Esteves et al. [70] presented a non-intrusive framework `CaT`, using kernel-level tracing to dynamically capture the content and context of network/storage, and then dynamically analyzing the exchanges and interactions among the distributed system components. In fuzz testing based approaches [72, 148], typically instrumentation is necessary to monitor code coverage, which itself does not always need any substantive static analysis; thus, they are considered purely (rather than hybrid) dynamic analysis. In fact, the instance of purely dynamic analysis of distributed programs can be traced back to even earlier time [226], where a distributed program's execution data is used to build a program activity graph which models performance-relevant activities in the program.

6.3.3 Program Representation.

There are several types of program representations used in existing program analysis techniques/tools for distributed software, including *CFG/ICFG*, *CG*, *PDG/SDG*, and *IFG*.

CFG/ICFG. The surveyed papers constructed and used CFGs at different levels, including rarely *instruction level* [129] and commonly *statement level* [76, 77, 79–82, 86, 141, 206, 207]. For example, Li et al. [129] proposed `DFix`, a distributed timing-bug fixing tool that first identifies individual instructions in the given program and then statically analyzes every program path on the instruction-level CFG. In addition, Fu and Cai [82] proposed FlowDist, a multi-staged refinement-based information flow analysis for common distributed software, which constructs the statement-level interprocedural control flow graph (ICFG) of the distributed system under analysis [82] to guide the instrumentation needed for its dynamic analysis. Note that some studies [72, 148, 156, 235] use control flow analysis by default as an underlying step, although the papers may not always explicitly describe CFG construction.

CG. A number of prior studies [41, 90, 91, 96, 147, 154, 179, 200, 219, 237] used CGs as the main program representation, at *application level* or *whole-program* level. An application-level CG only includes calling relationships in the application code, ignoring those in the libraries or between application code and library code. Modern distributed programs often involve extensive library calls (e.g., to the Java SDK APIs), for which application-level CGs are incomplete. A whole-program CG includes calling relationships in both application and library code [8]. For example, Wang et al. [219] developed `ANTaint`, a static taint analysis tool that builds the CG only for the application code, expands the CG to cover other code selectively, and then propagates application-level taints through the CG. Instead, Garbervetsky et al. [90] proposed a whole-program CG analysis framework that can scale reasonably well with the input code size. In CFTaint [239], the CG of a given micro-service is constructed to enable static interprocedural taint flow analysis.

PDG/SDG. *PDGs* are used in [83, 84, 130, 135, 136, 159] while *SDGs* are utilized in [19, 159, 195, 206, 207], as the representations of the analyzed distributed programs. For example, Li et al. [130] proposed a tool, `PCatch`, that could automatically analyze system executions to predict performance cascading bugs, using WALA to build the PDG. And Sirjan et al. [195] introduced a new state distribution policy based on the call dependency graph (CDG), which may be considered a special variant of SDG. Fu [78] developed a series of hybrid dynamic analysis of distributed software using PDG/SDG as the base program representation on which the application-specific analysis facts are derived.

IFG. A few distributed-program analyses are based on IFGs. For instance, in [188], an asymmetric bandwidth allocation strategy was derived based on the *IFG* of the given distributed system. Maturana and Rashmi [155] modeled a code conversion problem, also using *IFGs*. As another example, in [45], the *IFG* (i.e., IFD) was used to resolve node repair problems in distributed storage systems.

Note that there are a few studies that do not directly utilize any of the above program representations, but instead looking at abstract representations (e.g., formal model such as a state transition system). This is common in techniques based on model checking [92, 113, 134, 204, 216] or other formal verification methods [3, 64, 149, 201].

6.3.4 Algorithmic Parameter. Distributed-program analysis algorithms are typically designed with parameters regarding *Analysis Sensitivity* and *Data Granularity*, which immediately affect analysis behavior and cost-effectiveness.

Analysis Sensitivity. Some of the studies [22, 41, 83, 84, 90, 96, 200] presented *context-sensitive* analyses for distributed software. In particular, two studies [83, 84] proposed scalable and cost-effective analysis techniques whose respective configuration parameter can be enabled to perform context-sensitive analysis (1-CFA) or be disabled for context-insensitive analysis (i.e., 0-CFA).

The analyses in some of the surveyed works [76, 79–84] are *flow-sensitive* or flow-insensitive. For example, Fu et al. [83, 84] presented cost-effective dynamic dependence analysis framework, Seads and Dads, both supporting flow-sensitive analyses when the configuration parameter "flow sensitivity" is enabled, while the analyses are flow-insensitive when that configuration parameter is turned off.

Regarding field sensitivity, Gu et al. [96] developed a static analysis engine, `BigSpa`, which performs *field-sensitive* pointer/alias analysis and dataflow analysis. Garbervetsky et al. [90] presented a framework whose intra-procedural analysis is *field-sensitive* also. In contrast, the static analyses in [79, 82–84] are all *field-insensitive*.

Wang et al. [219] proposed a static taint analysis tool, `ANTaint`, using a 1-*object sensitive* analysis to only record the types of variables being accessible along taint paths. Several other studies [82–84] use object-insensitive static analysis to compute static dependencies in order to achieve better scalability overall.

Data Granularity. In several studies [13, 70, 83, 84, 104, 164, 211, 213, 215, 220, 229], the distributed-program analysis worked at *method instance level*, i.e., using all of the execution instances of the exercised methods. Other studies [30, 38, 39, 76, 79–84, 185] worked at *method level*, with only the first entry (i.e., program control entering a method) and last returned-into (i.e., program control returning from a callee back into the caller) events of each executed method being recorded and used. Depending on adjustable analysis configurations, two studies [83, 84] took both *method instance level* and *method level* data. In addition, [30, 76, 79, 80, 82] utilized both dynamic data at *method level* and that at *statement level* (i.e., mixed data granularity). In [20], the authors extended a real-time tracing framework to support the analysis and visualization of the flow of messages; thus, the data is at the *message level*, even coarser than method level.

6.3.5 Analysis Data. There are three types of analysis data utilized by the surveyed program analysis techniques/tools for distributed software, including *Run-time Event*, *System Log*, and (code or non-code) *Artifact*.

Run-time Event. Many studies (dynamic analyses) [12, 20, 24, 30, 38, 39, 41, 44, 70, 76, 77, 79–84, 111, 112, 125, 130, 135–137, 143, 145, 146, 150, 151, 159, 164, 165, 172, 178, 179, 193, 195, 199, 202, 211, 212, 227, 229, 237] used *message-passing events* (e.g., the event of receiving/sending a message from one process to another) as a major form of run-time events in distributed system executions. Other studies [12, 24, 30, 38, 39, 41, 50, 76, 77, 79–84, 86, 104, 114, 125, 127, 130, 135–137, 145, 146, 150, 151, 164, 172, 174, 178, 185, 188, 195, 199, 202, 211–213, 229, 231, 237] utilized *method-execution events* (e.g., the event of entering or exiting a method). Meanwhile, some other studies [50, 50, 114, 114, 188, 188, 236, 236] exploited *system events* for (dynamically) analyzing distributed programs.

For example, Cai and Thain [39] proposed DistIa, a dynamic analysis for distributed systems that exploits the semantics of *message-passing* and the happens-before relationships between *method-execution events*, to compute the change impacts of one method on others of a given distributed system. Instead, Kelbert and Pretschner [114] modeled and implemented data-flow tracking across distributed systems via 1) monitoring system events, 2) tracking the flow of data, 3) deciding whether data usage related system events should be allowed, delayed, inhibited, or modified, or whether compensating actions need to be taken, and 4) enforcing the decisions.

System Log. In real-world distributed system executions, it is common that *system-generated* logs were produced—i.e., the logs are generated by the original system without post-deployment modifications (e.g., instrumentation), as opposed to run-time events that result from instrumenting the code to capture relevant actions or occurrences during the executions. Such logs were used in many prior studies [15, 16, 24, 70, 125, 143, 147, 151, 162, 163, 188, 208, 231, 236, 237]. For instance, Luo [147] presents `approf`, a non-intrusive dynamic-analysis tool that can reconstruct an approximate run-time method-call hierarchy from the *system-generated* logs. In [109], the system logs are collected with adaptive logging levels to enable anomaly detection. Cotroneo et al. [54] also provides the ability to utilize failure logs generated by systems themselves to assist with run-time verification and failure detection, targeting cloud systems.

In contrast, some of the other studies [16, 24, 125, 150, 151, 193, 194, 208, 211] utilized *instrumentation-based probing* logs—the logs produced by the probes inserted into the system via instrumentation. For example, Beschastnikh et al. [24] introduced an approach to visualizing distributed system executions, which consists of (1) `XVector` that instruments a distributed system to capture partial ordering information from the happens-before relations among system events and (2) `ShiViz` that processes the resulting instrumentation-based logs and constructs time-space diagrams.

Artifact. Commonly, program analysis utilizes the code as the main form of artifact as the analysis data, regardless of its being source code, binary code, or intermediate representations. Non-code artifacts have also been utilized by some of the surveyed works. For instance, Chaturvedi et al. [47] proposed a service change classifier algorithm that mines change information from two versions of an evolving distributed software system. In addition to the code that implements the Web service itself, the analysis technique also utilized a specification that describes the Web service in a non-programming language. Revelio [68] utilizes historical bug reports and debugging logs to train a deep neural network to answer developers' queries when debugging distributed systems. Likewise, COLA [122] takes a large set of cloud system failure alerts as input to aggregate them hence helping developers sift out root causes using LLMs.

6.3.6 Analysis Scope. Some of our examined studies [91, 107, 179, 193, 212, 215] presented analysis techniques and tools for *distributed event-based* systems (DEBS), which use events to organize the communications among their components that typically run on different nodes [56]. For example, Popescu et al. [179] proposed `Helios`, an impact analysis technique for DEBS, which combines component-level control flow analysis, system-level state-based dependency analysis, and structural analysis that generates complete message dependence graphs. Since extracting information relies on the specialized messaging interface in DEBS, the analysis only works with such a particular kind of systems.

In studies [11, 21, 95, 116, 175, 177, 200, 231, 233], the presented analyses are aimed at *cloud* systems, which are generally considered an application variant of distributed systems [218]. For instance, Suminto et al. [200] presented the cascading outage bug elimination (COBE) project to detect/eliminate cascading outage bugs to improve cloud system availability. Likewise, a few studies [42, 71, 171, 190, 238, 239] are particularly focused on micro-services as a special kind of distributed systems. In [184], the authors target distributed systems of a specific component-based reconfiguration model, while the deadlock detection algorithm developed in [105] addresses MPI programs in particular. A specialized study [20] targets distributed robot systems when developing a run-time tracing method.

Some of the studies [6, 41, 116, 154, 206, 207, 228] presented analysis techniques/tools for *Internet of Things (IoT)* systems, which typically use embedded technologies to define environments where various physical objects interact and cooperate with other ones [69]. For example, Alashjaee et al. [6] presented `IoT-Taint`, an IoT malware detection framework based on *dynamic* taint analysis. Yavuz and Brant [228] proposed IFLOW, a *static* taint analysis for IoT frameworks, to detect bugs and localize relevant components in those frameworks.

The analysis approaches proposed in all of the other surveyed studies generally work with *common* distributed systems (§A.2), rather than only with specialized systems like DEBS [91, 107]. Meanwhile, distributed systems can be categorized in other ways [203], with each category presenting unique characteristics and program analysis challenges—and accordingly various analysis methods are potentially favorable. For instance, data-intensive systems process and manage large volumes of data across distributed nodes, emphasizing scalability and parallel processing, while microservices structure an application as a collection of small, independently deployable services each focusing on specific business functionalities [43], thereby enhancing modularity and scalability. These characteristics introduce unique challenges not typically encountered in traditional single-machine systems [30, 39, 84]. For example, due to the emphasis on concurrency and synchronization, the need for managing simultaneous operations across distributed nodes introduces complexities such as race conditions and deadlocks, necessitating specialized analysis techniques. Also, unlike centralized systems, distributed systems must handle scenarios where individual components may fail independently, requiring robust fault tolerance mechanisms. In response, specific program analysis methods are particularly effective [96]. As an example, for data-intensive systems, dataflow analysis can optimize data processing pipelines and ensure efficient data movement across nodes. For another example, dynamic analysis that monitors inter-service communications and detects issues such as improper API usage or latency problems are desirable for systems of microservice architectures.

6.4 Survey Result: Application

In the context of distributed systems, program analysis has enabled a range of concrete applications, including *Functional Testing*, *Fault Localization*, *Security Support*, *Performance Diagnosis*, and *Maintenance/Evolution Support*.

6.4.1 Functional Testing. In several works [95, 104, 111, 127, 178, 213, 216, 232], the proposed techniques were aimed to support *system testing* of distributed software. For instance, Lee and Levchenko [127] introduced a framework for system testing, following black-box testing strategies; the framework provides a pattern-recognition-based deterministic approach to replaying sequences of system events that may have caused defects and managing network messages by proxying all connections through orchestrators. As another example, Pereira [178] presented `Spider`, an automated approach that detects potential data races in distributed systems via SMT solving. `Spider` first performs a trace analysis to eliminate useless events and then builds a causality model by encoding the happens-before relationships from the rest of events. `Spider` utilizes an SMT solver to compute conflicting event pairs, so as to identify data races.

In addition, for *unit testing* of distributed software, Newsham et al. [165] proposed a tool chain that enables users to develop assertions on interaction history written in regular expressions incorporating inter-process and inter-thread dynamics in distributed systems.

6.4.2 Fault Localization. Some studies [77, 83, 84, 163, 213] were used to locate *method-level* faults. For example, Fu et al. [84] presented SEADS, a dynamic dependence analysis framework that works as an online dynamic analysis to solve the fault localization problem (via dynamic slicing) against unbounded execution traces. SEADS features with the ability to automatically adjust itself to better cost-effectiveness tradeoffs (than otherwise) with user-specified time budgets.

This ability is realized by changing varied analysis parameters according to analysis time costs and precision. As a result, the framework continuously provides *method-level* dependencies [35] to immediately empower fault localization.

In contrast, other studies [19, 30, 76, 79, 81, 82, 112, 219] work at a finer-grained level, reporting *statement-level* faults. For instance, Wang et al. [219] proposed `ANTaint`, a static taint analysis tool for industrial service-oriented architecture (SOA) applications. It improves the tool scalability by building a sound CG and provides a precise taint model by pruning unrealizable paths, hence pinpointing *statement-level* faults with the correctness for 95% of production-benchmark cases (the corresponding precision and recall was 95% and 98%, respectively).

Note that we only have a *comparative* qualification of fault localization regarding its granularity. In general, fine-grained fault localization refers to identifying the precise elements within a system responsible for a fault, where the localization granularity can vary depending on the system architecture and the methodologies employed. For example, for fault localization focusing on the software's codebase, statement-level localization is considered finer-grained than method-level localization. In contrast, microservice architectures consist of numerous loosely coupled services, each with its own set of quality metrics. In such systems, causal inference-based approaches have been developed to achieve fine-grained root cause analysis at the quality-metric level (e.g., CPU usage, memory consumption, or response time).

6.4.3 Security Support. Program analysis has been a fundamental methodology for offering security support, which we found is also the case with distributed-program analysis—in fact, it has served all of the different security objectives.

The analyses proposed in [2, 11, 13, 93, 114, 171, 177, 195, 220, 233] focused on distributed software *confidentiality*. For example, Zavou et al. [233] presented `Cloudopsy`, a framework that remediates cloud users' security concerns through visualization and automated analysis based on the graphs produced by a visualization tool for data safety.

In a number of prior studies [2, 11, 13, 63, 116, 175, 177, 220], the program analysis techniques primarily concerned the *integrity* of distributed systems. For instance, Acay et al. [2] proposed `Viaduct`, a tool that allows developers to annotate a distributed system with information flow labels for specifying security policies regarding system integrity. Pilla et al. [63] proposed a technique for system auditing to manage the complexity of distributed systems (blockchains) hence defending the integrity of the systems. Kucab et al. [123] utilizes hardware capabilities to develop a new attestation process for defending both static and runtime integrity of cloud deployments.

Some studies [99, 175, 177, 193] addressed the *non-repudiation* objective of distributed system security. For example, Pappas et al. [175] proposed `CloudFence`, a framework aimed at preventing extensive security breaches so as to achieve non-repudiation of cloud environments. `CloudFence` allows users to independently audit their data treatment through third-party services, enables service providers to confine the use of sensitive information in well-defined domains, and offers additional protection against inadvertent data leakage and unauthorized access.

The *accountability* objective of distributed software security also has been addressed as a main application goal [21, 99, 174, 193]. For instance, Hauser et al. [99] proposed an intrusion detection framework for validating the accountability of distributed systems. Based on taint marking and implemented in the Linux kernel as a security module, the framework uses tokens as security labels on network packets to carry taint information between multiple hosts.

Quite a few other studies [6, 30, 41, 76, 77, 79–82, 84, 86, 116, 154, 200, 206, 207, 219, 220] were aimed to address *authenticity* problems with distributed systems and/or their executions. For example, `Secure-CamFlow` [116] focused on securing the entire process of the data migration from devices to a cloud system, where the system data flow is monitored through user-specified information flow control policies. It achieves user authenticity between the cloud service provider (CSP) and users, via storing relevant security keys in a CSP directory with the signatures of a key distribution center.

6.4.4 Performance Diagnosis. Most of the earlier works using distributed-program analysis to diagnose system performance [16, 20, 24, 62, 87, 104, 108, 145, 147, 162, 189, 194, 208, 211, 234, 237] concerned the *time cost* of distributed systems, including performance anomaly identification in cloud/IoT systems [16, 104, 211]. For example, Huang et al. [104] presented an analysis tool named `tprof`, a performance profiler that aggregates the traces of a distributed system to help diagnose time-wise performance inefficiencies (e.g., to identify the relatively slow regions of the system).

Neves et al. [164] proposed a black-box monitoring approach that gathers the detailed information about the processes in a distributed system for helping the user diagnose performance problems but focusing on the *memory* that is available to interim data and original cache. Performal [235] aims to verify whether the time latency property of a distributed system is observed, proving rigorous upper bounds for the latency, using formal verification techniques.

Several other studies [45, 97, 151, 155, 164, 183, 208] considered *network throughput* as the focus of performance issues. For instance, Rakotondravony et al. [183] presented an interactive and cost-aware visualization architecture for monitoring data (e.g., network traffic) of distributed systems, benefiting developers to verify the system conformance to network throughput requirements.

6.4.5 Maintenance/Evolution Support. Earlier works that provide this support often present analysis techniques that may serve various maintenance/evolution (e.g., change-management) tasks. In particular, the analyses in [30, 38, 39, 47, 79, 81–84, 91, 135–137, 179, 212, 238] can support *change impact analysis* [29] of distributed programs. For example, Chaturvedi et al. [47] proposed an intelligent tool, `AWSCM` (short for Automatic Web Service Change Management), based on an interface slicing algorithm that analyzes the change impact between two versions of an evolving distributed program. It works by comparing the old and new versions through three classification labels "inserted", "deleted", and "modified". Along with a service evolution analytic model, `AWSCM` supports change mining of evolving distributed systems.

Other studies [19, 21, 42, 80, 107, 112, 119] targeted the *program comprehension* task hence supported the maintenance and evolution of distributed software systems. For instance, Fu et al. [80, 86] presented multiple interprocess communications (IPC) metrics for common distributed programs computed via dynamic dependence analysis, which help understand various quality factors/aspects of the programs. Cerny et al. [42] developed a static-analysis-based approach to micro-service-specific architecture reconstruction to support system understanding via visualizations, similar to the work in [26] achieving the same using static code analysis of the microservice mesh. The micro-service coupling metric proposed by Zhong et al. [238] can also support the understanding of the system itself and its quality.

6.5 Survey Result: Evaluation

We examined two attributes in the evaluations of program analysis for distributed programs: *Dataset* and *Metric*.

6.5.1 Dataset. The dataset used in the evaluation of a distributed-program analysis at least includes the subject distributed systems, and optionally (as required by dynamic analysis) their run-time inputs. Regarding the subjects alone, existing analyses have been evaluated against *Benchmark Suite*s and *Real-world System*s.

Benchmark Suite. A large number of prior studies [24, 104, 108, 114, 129, 135–137, 162, 163, 172, 177–179, 206, 207, 211, 215, 219] utilized *open/standard* benchmark suites (usually created by others) in their evaluations. For example, Beschastnikh et al. [24] used the `Yahoo! Cloud Serving Benchmark (YCSB-B)` [51] to evaluate their tool that creates visualizations of distributed system executions. Liu [135] took benchmarks from the `TaxDC` suite [128], all triggered by the communications across nodes of a distributed system, for evaluating `DCatch`—a technique to model and detect message timing bugs in the system. Jia et al. [108] evaluated the training performance of deep neural network (DNN) models by using image classification benchmarks from the `CIFAR-10` [144] open dataset which contains 60k images.

Other studies [2, 16, 90, 107, 127, 146, 165, 188, 193, 199] used benchmark suites that were *self-curated* (by the authors themselves). For instance, to evaluate the software model checker FlyMC for testing distributed systems, Lukman et al. [146] used self-curated bug benchmarks that included the numbers of bug-triggering events (measured as the bug "depths"). Also, Newsham et al. [165] evaluated a framework that they proposed for developing assertions on threads' interaction history in a distributed program using two self-curated benchmarks: (1) a message-passing benchmark designed to measure overheads and (2) a verification benchmark designed to verify the time scalability according to the historical sizes of threads. Nunez [170] only considered a simulated environment (using a simulator to model different distributed systems architectures) in their evaluation. In [118], the API extension to a graph-based distributed-memory runtime system (to track dependencies on opaque node-local objects and transfer runtime-managed data) was only evaluated on synthetic benchmarks created by the authors.

Real-world System. Most of the surveyed papers utilized *large-scale* real-world distributed systems as their evaluation subjects [12, 15, 22, 30, 38, 39, 41, 47, 62, 70, 76, 79–84, 87, 89, 91, 96–99, 111, 116, 125, 127, 129, 130, 135–137, 143, 145, 147, 150, 151, 154, 162, 164, 172, 175, 179, 185, 194, 195, 199, 200, 202, 207, 208, 211–213, 215, 227, 229, 231, 232, 234, 236]. For example, Stuardo et al. [199] used stable versions of four large real-world distributed systems (i.e., Cassandra, HDFS, Riak, and Voldemort) to evaluate ScaleCheck—an approach for finding scalability bugs in large-scale distributed systems. Li et al. [129] evaluated DFix, a timing-bug-fixing tool for distributed programs, against 22 harmful bugs from four widely used real-world distributed systems: Cassandra, Hadoop, HBase, and ZooKeeper. Yuan et al. [232] studied a sophisticated error [110] in the transaction protocol of a real-world distributed system Cassandra [40] by modeling the error in Erlang and using Concuerror [49], a systematic testing tool for Erlang, to check against such errors using randomized and systematic strategies, each of which performed 100,000 trials in the evaluation.

There were also some studies [19, 50, 159, 183, 189] using *small-scale* distributed systems as study subjects in their evaluations. For instance, Mohapatra et al. [159] evaluated their slicing tool DSDJ, using seven small-scale distributed Java programs. The largest of those subject programs only had 894 lines of code.

In a preliminary study on detecting technical debt based on anti-patterns [71], the authors use both an open-source project as a benchmark and a real-world (train ticketing) system for evaluation. This is similar to the evaluation setup in CFTaint [239], MirrorTaint [171], and DisTA [217]. Yet, overall it is not common that both benchmark suites and real-world subjects are considered in the evaluation among our survey works. On other hand, there are more-theoretical works (e.g., [210]) that do not have any empirical evaluations and used neither benchmark suites nor real-world systems.

6.5.2 *Metric.*

Existing evaluations of distributed-program analyses typically seek to measure the *effectiveness* and/or *efficiency* of the analysis technique/tool [53].

Effectiveness. Many studies [15, 38, 39, 79, 81–84, 91, 150, 211, 229, 237] used *precision* as a key metric in evaluating the effectiveness of the proposed analysis. For example, Astekin et al. [15] gauged the analysis precision when evaluating their technique for anomaly detection based on large distributed system logs.

Some of the surveyed studies [15, 38, 39, 79, 81–84, 91, 229] utilized *recall* as an effectiveness metric, typically in addition to precision. For instance, Garcia et al. [91] proposed a static analysis technique, Eos, whose analysis results were compared with the "ground truth" to compute the technique's recall.

False positive rates (FPRs) were reported in some studies [91, 219, 229, 232], while *False negative rates (FNRs)* were presented in others [91, 130, 135–137, 232], also as part of the effectiveness metrics. In two studies [91, 200], *True positive rates (TPRs)* were also measured in their evaluations. For example, the evaluation of Eos [91] considered spurious (false-positive, FP), missing (False negative (FN)), and matching (true positive (TP)) results (i.e., message types and intra-flow dependencies) as indicators of effectiveness. Suminto et al. [91, 200] also evaluated their cascading outage bug elimination (COBE) technique in terms of the number of true positive (TP) cases.

In the evaluation of their analysis, Astekin et al. [15] reported the *F1* score as a common accuracy metric, according to the precision/recall of the evaluation results—i.e., F1 = 2 * (precision * recall) / (precision + recall).

Efficiency. As the most frequently used efficiency metric, *time* (e.g., analysis time cost in absolute terms, run-time overhead as a ratio) has been widely used in the prior studies we surveyed [2, 12, 15, 19, 22, 24, 30, 38, 39, 62, 70, 79, 82–84, 87, 90, 91, 96, 98, 104, 107, 108, 114, 116, 125, 127, 129, 130, 135–137, 143, 145–147, 150, 151, 154, 159, 162–165, 175, 177, 178, 183, 188, 189, 193, 194, 199, 200, 202, 206, 208, 211–213, 215, 219, 227, 231, 232, 234, 236, 237]. For example, Pereira et al. [178] assessed the efficiency of `Spider`, a tool for identifying data races from distributed system traces, by measuring its time and space overhead. Another example is Halfmoon [182], which optimizes log placement in serverless-computing runtime system in order to reduce latency and run-time overhead caused by logging.

Several studies [22, 127, 147, 190] also measured the *memory usage* of their analysis techniques/tools for distributed systems in the evaluations. For instance, Benavides et al. [22] showed the memory usage of capturing system events using `DProf`, a dynamic analysis tool used to construct distributed performance profiles.

The analyses in [96, 147, 193, 208] used *CPU usage* data as efficiency metrics. As a specific example, Thereska et al. [208] measured the CPU time consumed in the client and storage-nodes and found that the CPU utilization had a direct relationship with the data encoding scheme selected in the client.

Some studies [107, 114, 147, 151, 208] measured *network usage* when the corresponding analysis techniques/tools were evaluated. For instance, Kelbert et al. [114] measured the impact of their data usage control infrastructure in terms of network throughput in their evaluation.

In addition, a number of our surveyed studies [2, 24, 79, 81–84, 90, 96, 146, 147, 151, 162–164, 188, 189, 199, 211, 212, 215] took into account the *scalability* of their analysis techniques/tools for distributed systems when evaluating the efficiency. For example, Lukman et al. [146] evaluated the efficiency of their testing approach for distributed systems in terms of the scalability of the tool named `FlyMC`.

Some of the studies [43, 71, 210] do not consider any of the above metrics as they are preliminary and/or exploratory in nature. Thus, they did not come with official evaluation experiments/results. For instance, Cerny and Taibi [43] discuss how static analysis may help support the development/management of micro-services as emerging ideas.

7 Analysis and Discussion of Survey Results

In this section, we systematize the state of knowledge about program analysis for distributed software, discussing the limitations of existing works and challenges facing them. Based upon these discussions, we shed light on several future research directions for overcoming/mitigating those limitations and challenges.

7.1 Limitations and Challenges of Existing Works

By examining the surveyed analyses of distributed software, we observed a number of common limitations suffered by existing relevant works, signifying challenges concerning their technical approaches, applications, and evaluation.

7.1.1 Scalability and Cost-effectiveness. Program analysis for distributed software often faces scalability challenges. The reason is that when the analysis works at a fine granularity—for desirably precise results, the typically large sizes and great complexity of distributed systems imply high analysis costs [82], which makes it difficult for the analysis to scale to large-scale, complex systems.

Achieving highly cost-effective program analysis for distributed software is also challenging because the two factors here (cost and effectiveness) often counteract and compete for a given analysis. Specifically, an efficient and scalable analysis is usually coarse and imprecise, while a precise analysis is typically expensive and unscalable. For example, DistIa [39], a lightweight dynamic (impact) analysis for distributed programs, monitors method events and their timestamps during the program execution hence approximating dynamic dependencies among relevant methods. DistIa is highly efficient and scalable, but its results (dependency sets) are very coarse (imprecise), leading to overall low cost-effectiveness. For instance, Cai and Fu [30] developed D^2Abs, a dynamic dependence analysis approach for distributed programs. Its most-precise version, referred to as Doda, when applied to Voldemort [14]—an industry-scale distributed key-value storage system, could not even finish the analysis in over 12 hours on a high-performance server (i.e., Ubuntu 16.04.3 LTS workstation with four 2.67 GHz processors and 512 GB DRAM). Apparently, with this level of efficiency, Doda is neither practically scalable nor cost-effective for industry-scale distributed systems [84].

Moreover, what aggravates the cost-effectiveness balancing challenge is that different systems, as well as the different executions of the same system, often exhibit varying characteristics that necessitate various trade-offs between the cost and effectiveness of the same analysis. Thus, one tradeoff rarely fits all scenarios universally, and different tradeoffs are needed [30, 82]. D^2Abs [30] attempts to address this need by mixing the use of various kinds of program information hence offering various cost-effectiveness tradeoff options. Yet the technique only offers a few options, and the particular option must be manually set up specifically for each system and execution. Seads [84] made a further advance in this regard by automatically tuning the cost-effectiveness of a dynamic dependence analysis for distributed systems through reinforcement learning. Nevertheless, Seads still faces the challenge of meeting the diverse cost-effectiveness balancing needs of different systems with varying characteristics.

7.1.2 *Test Availability and Quality.*

Recall that among all the surveyed program analyses for distributed software, the vast majority are dynamic analysis (Figure 5), which is commonly known to rely on the test inputs that trigger the program executions underlying the analysis. Accordingly, most of the surveyed analyses are limited by the availability and quality of those run-time tests. For example, Fu and Cai [82] proposed FlowDist, a multi-staged dynamic information flow analysis approach for distributed software systems, which aims to identify vulnerabilities exercised at runtime during the system executions. Yet FlowDist's actual potential and capabilities for vulnerability discovery depend on the underlying test inputs' coverage (of any vulnerabilities). If some vulnerabilities are not covered during the executions being analyzed, those vulnerabilities would not be found. As another example, Yuan and Yang [232] presented `Morpheus`, an effective concurrency testing technique for real-world distributed systems in Erlang, which relies on given test cases to drive the tested systems hence high-level invariants to check against fail-stop errors and infinite loops. Any such defects not covered by those test cases would be missed by the technique.

Intuitively, high-quality test inputs are essential for dynamic analysis of distributed software to be practically useful. However, in practice such test inputs are not always available. In fact, many software packages for distributed systems come with quite limited test cases [82, 83], which is a major challenge facing dynamic analysis for distributed systems.

7.1.3 *Applicability and Accommodations of Various Architectures/Platforms.*

Program analysis techniques/tools for distributed systems are overall different from those targeting single-process programs. The former are generally much fewer than the latter in terms of the overall presence in the current literature, while the latter are mostly not immediately/effectively applicable to distributed systems. This forms a sharp contrast with, and a standing gap between, the increasing dominance of distributed systems among any software application domains and total availability of techniques/tools for those systems. Thus, a critical challenge here lies in how to adapt existing program analysis techniques/tools to distributed systems—as one way of mitigating the gap.

Another applicability issue is that even among the relatively few techniques/tools that do work for distributed systems, some only fit specific system architectures, hence suffering from limited applicability scopes. For instance, several static analyses [91, 107, 179, 212] of distributed programs were designed for and only work with event-based distributed systems [56, 160], neither immediately working with nor readily adaptable for common distributed systems. Achieving this adaptation or developing techniques that work well with a good variety of distributed-system architectures remains a standing challenge.

The third type of applicability issues has to do with the platform on which the distributed systems are supposed to run. For example, Sapountzis et al. [189] implemented a dynamic information flow tracking (DIFT) framework, `MITOS`, on top of `FAROS`, which is an existing open-source DIFT system running on Windows 7; this apparently limits the operating system choice of users who want to use `MITOS`. As another example, Yuan and Yang [232] presented a concurrency testing tool, `Morpheus`, with limits ways to support interacting with non-Erlang code. And Chaturvedi et al. [47] proposed an interface slicing algorithm that only works with distributed programs written in the web services description language (WSDL). Essentially, the platform (language runtime here) restrictions lead to applicability barriers.

Finally, it is not uncommon that the applicability of an analysis is limited by how it is designed or its inner workings. One prevalent evidence is that a great portion of the surveyed dynamic analyses for distributed programs rely on static instrumentation, hence requiring modifications of (i.e., inserting probes into) the original distributed systems. For instance, most of the state-of-the-art tools implementing such analyses (e.g., [30, 32, 37–39, 77, 81–84]) must statically probe for relevant run-time information to enable their analysis. These restrictions make them inapplicable to real-world scenarios where the subject (e.g., production) systems cannot be modified. Getting over such applicability limitations is certainly not trivial—the analyses would need to be largely redesigned.

7.1.4 *Evaluations.*

A major challenge to distributed-program analysis among the surveyed papers lies in the weaknesses/limitations with their evaluations. Among others, only a very few subject systems were considered and/or the scale of the subjects evaluated against was often too small; and in many cases, the evaluation subjects were not real-world distributed systems. For example, Barpanda and Mohapatra [19] presented a dynamic slicing algorithm for distributed object-oriented systems where they defined various kinds of dependencies induced by interprocess communications. However, the algorithm was not evaluated against enterprise-scale distributed systems in the real world. And its largest evaluation subject had only 918 source code lines. Likewise, Mohapatra et al. [159] proposed a dynamic slicing technique, `DSDJ`, which was not evaluated against any large real-world systems as evaluation subjects. The challenge was that, without being evaluated against a diverse set of real-world systems, the analysis techniques/tools have not been concinvingly assessed against their practical effectiveness and scalability.

In addition, some studies did not present reasonably sufficient evaluation results. For example, in the evaluation of [154], Mandal et al. only reported information flow descriptions and cross-program taint analysis time on the edge software and mobile applications, but there was not any result on accuracy, such as precision, recall, or F1, being reported in the paper. Another example is [47], where only the reduction ratios of test cases were provided, but no accuracy and/or performance metrics such as analysis time costs, overheads, etc., are reported. These limitations increase threats to the validity of results reported in respective studies that cannot be easily justified/mitigated.

Notably, as recently found in the area of software vulnerability analysis [169], open science is not well practiced in the field of program analysis for distributed systems according to our survey. The vast majority of the surveyed papers did not provide accessible artifacts such as code and datasets developed/used in their evaluation. This significantly hinders the advancement of the field with respect to the critical merits of reproducible and replicable research.

7.1.5 Practical Effectiveness. Among those surveyed prior works that were relatively more sufficiently evaluated, we also identified, from the evaluation results, a lack of efficiency and/or effectiveness for wider practical adoptions as a common limitation/challenge.

First, some of the existing analyses exhibited lower effectiveness in certain use scenarios than in others when compared to peer approaches. In [193], Sigelman et al. introduced `Dapper`, Google's tracing infrastructure for distributed systems, targeting on-line systems rather than off-line data-intensive workloads, such as those that MapReduce fits [59]. `Dapper` is effective in determining system parts experiencing slowdowns but insufficient for finding the root causes. In [87], Fukuda et al. developed an agent-based data-discovery approach, MASS (short for multi-agent spatial simulation), for distributed data analysis. While meritorious, MASS cannot handle non-numeric data as effectively as MapReduce and Apache Spark [173] do. Thereska et al. [208] proposed an infrastructure, `Stardust`, for collecting and tracking traces in distributed systems. Yet `Stardust` does not help more than known software profiling tools for finding/fixing algorithmic problems—e.g., a poor networking layer implementation may mean that requests handling spends most of the time using network resources. `Stardust` identifies the network as the source of latency but does not have better suggestions for fixing the problems compared to `gprof` under Linux [73, 94].

Second, some of the current analyses are not fully automated, requiring human intervention during the analysis or much additional (e.g., post-analysis) effort. Toslali et al. [211] developed a variance-driven automated instrumentation framework, `VAIF`, for distributed applications. The framework aims at automatically searching for possible instrumentation space choices for diagnosing performance problems. However, `VAIF` requires additional assistance for asynchronous design patterns if any latency is not reflected in the response time of critical paths. Moreover, `VAIF` cannot identify whether observed variances are caused by the application code itself or low-level code (e.g., kernel code); nor can it identify transient/infrequent problems that disappear before `VAIF` is used.

Third, some of the extant analysis approaches, especially those based on purely static analysis, face practicality challenges due to great imprecision or low recall. Wang et al. [219] proposed `ANTaint`, a static taint analysis approach for Java applications. An effectiveness challenge with this approach is that it suffers from great imprecision when analyzing taint flow paths that involve multiple systems (because its scanner cannot recognize all possible data types in the given application that includes such paths). Similarly, Mandal et al. [154] presented a cross-program taint analysis approach for Internet of Things (IoT) systems, only considering communication channels as follows: a method is conservatively assumed to be tainted if it reads data from a channel and any tainted data also reached the channel. The approach suffers excessive imprecision against large industrial IoT systems using one channel for multiple different communications, because the taint analysis is overly conservative for those systems. Concerning recall, Lu et al. [143] developed an automatic tool, `CloudRaid`, to find distributed concurrency bugs by automatically analyzing the message orderings that may expose errors. However, it cannot guarantee that the pruned message ordering will not trigger any error. Also, it cannot detect bugs triggered when instrumenting delays in the middle of bug-message handlers.

7.2 Future Research Directions

Following the limitations and challenges with existing distributed-program analysis techniques/tools surveyed, we now discuss important future research problems accordingly, while offering insights into the directions that may be fruitful in addressing those problems.

7.2.1 Balancing Cost-Effectiveness and Achieving Scalability. As we revealed as discussed above, balancing cost and effectiveness while achieving practical scalability is a fundamental challenge facing distributed-program analysis. From some preliminary effort being invested towards dealing with this challenge, we identified several promising directions in developing future scalable, cost-effective program analysis for distributed systems.

Variable cost-effectiveness. To address the challenge of balancing the analysis effectiveness and cost, several earlier approaches for single-process software (e.g., DiaPro [33, 37], Diver [28, 32]) offer variable cost-effectiveness tradeoffs to provide flexible choices and hence satisfy varying user requirements and budgets to some extent. Following a similar path but targeting distributed systems, D^{2}Abs[30] provides four versions of a dynamic dependence analysis each offering a different level of cost-effectiveness tradeoff. While its current four options in total may still not meet all the varying cost-effectiveness needs for the diverse world of distributed systems, D^{2}Abs did point to a rewarding avenue toward future research for addressing the challenge here—future distributed-program analysis may provide a number of (potentially more than four) variants (e.g., via algorithmic configuration variations) that each works at a unique level of cost-effectiveness. In this way, the analysis tool will fit more needs and use scenarios.

Self-adaptive design. While valuable as a mitigating solution, providing multiple cost-effectiveness tradeoff options tends to be a short-term strategy—the options need to be manually predefined and are difficult to be predefined in advance such that they meet the various needs of users; it is also hard to predict how many options are sufficient. A longer-term solution, as recently debuted in Seads [84], is to *adaptively and automatically* set cost-effectiveness tradeoffs for the varying given distributed system executions. To achieve practical scalability and cost-effectiveness, Seads automatically adjusts its analysis configurations during the program execution, using a type of reinforcement learning (i.e., Q-learning) strategy. Seads was found to lie in between DistIA and D^{2}Abs; that is to say, Seads is more precise than DistIA and more scalable than D^{2}Abs.

Although showcasing a promising methodology at a very-high level, Seads suffers from low precision due to delays in tuning its configuration, making the current query unable to benefit from the higher-precision configuration (the tuning only benefits future queries). Seads cannot adapt to different subject executions to push the precision by maximally utilizing the budget due to its using generic reinforcement learning in a black-box manner, applying the same algorithmic setting (e.g., reward initialization) for any given subject and execution. Also, it does not adapt to different subjects at all—it uses a constant reward function and initializes it the same way regardless of what subject system is fed to the analysis. In this sense, Seads is not a truly self-adaptive analysis.

Looking forward, we believe a more promising future direction is to develop truly self-adaptive analysis, which should automatically monitor relevant changes in dynamic/uncertain environments and then adapt itself to continuously meet cost and effectiveness requirements for ensuring optimal adaptations. Importantly, such analyses should adapt to both varying subjects and variations (in run-time behaviors) during a particular subject execution. As a specific example in this future direction, one may realize the true self-adaptation by considering and controlling algorithmic configurations in a much more fine-grained manner than what Seads currently does. For example, in reference to the configuration design in Seads [84], instead of considering only binary values for each configuration parameter, we may consider more options (e.g., the various depth values for the calling context for context-sensitivity rather than just context-sensitive versus context-insensitive; similarly, the degree of object sensitivity can be considered as well). The finer granularity here will lead to finer tuning of cost-effectiveness tradeoff hence more likely attaining optimal self-adaptation.

Leveraging machine learning and deep learning. Machine learning (ML), especially deep learning (DL), has been applied to and achieved significant successes in advancing program analysis (e.g., source code analysis) [225], including that for distributed systems. One immediate example is the ML-based cost-effectiveness balancing in Seads [84] as we already discussed extensively. The nature of ML also makes it suitable for predicting the dynamics of distributed software hence further assisting with understanding the systems. For example, studies [15, 67, 83, 85] utilize ML algorithms, including reinforcement learning, supervised learning, unsupervised learning, and neural network based techniques, to analyze distributed systems of different kinds (e.g., common distributed systems and cloud systems).

Furthermore, deep learning (DL) has gained growing momentum in recent years because of its significant promise for automatically adapting agents to varying environments. In the context of reinforcement learning, deep learning approaches typically exploit deep neural networks to model their decision-making capabilities by learning policy functions; one main benefit of using DL is to encode high-dimensional (complex) environments. More broadly, to capture the (static and/or dynamic) characteristics of distributed software and then analyze the software, DL may be a good design choice. For instance, studies [108, 236] have explored relevant DL techniques for enhancing respective distributed program analysis. In fact, according to our survey results (Figure 5), ML/DL currently has only seen little use in distributed-program analysis; we believe much potential has yet to be tapped and should be in the future.

7.2.2 Accommodating Emerging System Architectures. The landscape of real-world distributed systems continues to expand quickly, spawning a growing diversity of such systems in terms of system architectures. For example, many Internet of Things (IoT) systems/platforms (e.g., AWS IoT, FIWARE, IBM Watson IoT Platform, Microsoft Azure IoT Hub, OpenMTC, SiteWhere) have been developed and deployed in recent years, which has drawn much attention from the relevant research (e.g., systems and security) communities. Developing program analysis and its applications for IoT systems is a timely future research theme. For another example, edge computing has emerged lately as a new system architecture in the general domain of distributed computing, which features several advantages over the traditional (cloud-based) computing paradigm, such as improved response time and reduced energy consumption [192]. Thus, the growing need for program analysis techniques/tools targeting edge-computing systems is also on the horizon.

Similar needs are also emerging for block-chain systems and distributed machine learning (e.g., federated learning) systems. Currently, program analyses working with these systems are rare. Yet given the historical evidence of how program analysis has enabled the wide range of useful and powerful applications, we believe that future research on distributed-program analysis should attend these new/emerging computing architectures of distributed systems.

7.2.3 Testing Techniques. As mentioned earlier, dynamic analysis has demonstrated tremendous successes in supporting the development and quality assurance of distributed systems in the past—again, the majority of the surveyed distributed-program analyses are dynamic. Yet as we discussed earlier (§7.1), the actual capabilities and benefits of those dynamic analysis techniques/tools are bounded by the availability and quality of run-time test inputs that trigger the system executions underlying the analysis. And the current status is that the availability and quality are critically lacking.

Thus, an urgent and promising research direction is to focus on improving the quality of distributed system run-time input data, including providing them where they are not readily available. One viable approach is to manually collect/curate such inputs, but a more promising avenue is to automatically generate them. In particular, fuzzing as a random-testing (i.e., test-generation) technique has greatly evolved and is becoming increasingly popular for various software application domains. Yet, practical fuzzers for distributed systems are not as available as they should be—even the most recently developed fuzzer [133] which works with multi-language systems still do not work sufficiently with distributed systems. Thus, future research should arise to fill this critical gap.

7.2.4 Evaluations. According to our survey results and our analysis of them, an urgent need observed is that of evaluation subjects that represent real-world distributed systems and their execution scenarios. In a few prior works [2, 16, 90, 107, 127, 146, 165, 188, 193, 199], some benchmark suites were developed for evaluating corresponding program analysis techniques/tools for distributed software. As a specific example, Newsham et al. [165] developed two benchmarks designed to measure overheads and verify the time scalability with history size, respectively. However, these benchmarks are mostly not representative of real-world systems. Thus, development/curation of *realistic* benchmarks should be part of future research on distributed-program analysis. Some recent works, although not addressing distributed systems, demonstrated a fruitful direction in this regard—automatically generating such benchmarks [167, 168] or curating a *standard* benchmark manually [131, 132]. Similar efforts in future research would be highly valuable for advancing the field of distributed-program analysis as well.

Besides filling the gap regarding benchmarks, another impending need for addressing limitations with evaluating existing distributed-program analysis techniques/tools is to develop widely recognized/accepted evaluation metrics and even empirical standards. The lack of such commonly adopted metrics/standards has impeded the progress of program analysis for distributed software, as the use of inconsistent and widely-ranging metrics/procedures make the existing evaluations hard to validate and existing techniques/tools hard to compare. In [80], Fu and Cai defined a novel set of metrics for common distributed systems aimed to measure/characterize their interprocess communications (IPC), a critical aspect of their run-time behaviors. While those metrics have not been shown to be immediately useful for evaluating distributed-program analysis, effort of similar spirit should be promoted in future research.

8 Threats to Validity

In the previous sections, we have presented a series of findings from our article selection, attribution framework formulation, and the attribution of surveyed papers. Given the manual nature of this entire workflow and the scale of our survey, we do not claim that we can rule out our potential biases—other researchers conducting the same survey might end up with with different findings and conclusions. Thus, our survey results are subject to various validity threats, pertaining to each of the major steps/phases of our survey workflow (as shown in Figure 3), as elaborated below.

Literature search. In the process of collecting relevant articles, we selected papers through keyword searches. One threat is related to our search engine (i.e., Google Scholar) not being designed for supporting literature reviews [25, 198] hence possibly having produced errors in our literature search. We mitigated this threat by utilizing multiple keywords mentioned earlier (§4.1). Furthermore, we have attempted to increase the representativeness of papers selected by the forward and backward snowballing process that identified additional relevant articles cited by or citing original papers (§4.2). Also, selection (inclusion and exclusion) criteria have been exploited to reduce this threat (§4.3).

Survey taxonomy derivation. A potential threat to the validity of our survey taxonomy derivation is that this process is subjective and also depends on the search keywords used in the *Phase 1* of our literature search (Figure 4). However, the resulting derivation can be validated through its usefulness as reflected in our taxonomy/characterization results. In addition, we have carefully checked corresponding attributes and items in each aspect within the taxonomy, as listed in Table 1, demonstrating the applicability of our survey taxonomy derivation process.

Paper attribution. Like our survey taxonomy derivation, the paper attribution process is also subjective and may be difficult to reproduce. A threat to validity in this respect is the duplication in the process and the resulting paper-attribution statistics. If one and the same paper (content) was published multiple times (e.g., via extension/revision) in different conference proceeding(s), journal(s), and other digital libraries, it should be taken into account only once in the statistics. We have addressed this threat by ensuring uniqueness in the paper selection and attribution.

The bias of the original researchers concerning certain concepts and choices may also be a potential threat to the validity of our survey results. As a remedy, we have examined the relations among attributes and items in Table 4, Table 5, and Table 6, to avoid confounding overlaps. For example, the definitions of a common distributed system and a specialized distributed system are mutually exclusive (§A.2). In other words, a distributed system cannot be both a common distributed system and a specialized distributed system at the same time.

Survey result analysis. During our analysis of (i.e., discussion on) the survey results, we might have missed certain limitations/challenges of existing work and hence ignored corresponding future research directions. This implies a potential threat. To mitigate the threat, after carefully reviewing the selected articles, we recorded/listed all relevant limitations and challenges in the field of program analysis for distributed software, and then thought over across the aspects and attributes in our taxonomy when identifying future research directions according to how to overcome those limitations/challenges.

9 Conclusion

In this paper, we presented the first systematic literature survey on program analysis for distributed software systems. We selected 153 articles spanning from 1995 to 2024, derived a novel taxonomy, and characterized those articles on the basis of the taxonomy including three main aspects (i.e., approach, application, and evaluation). Then, we mapped the current relevant literature to each attribute under every aspect and, if applicable, each item (sub-attribute) under every attribute, while summarizing major findings accordingly. Next, we went above the immediate observations and discussed our insights into the key limitations and challenges with existing techniques/tools in the field of distributed-program analysis. Finally, in accordance with the discussion and following the insights, we shed light on future research directions in this field towards addressing open problems corresponding to those limitations/challenges, concerning the technical development, empirical evaluation, and wider applications of program analysis for distributed software systems.

Acknowledgment

We thank our associate editor and reviewers for insightful and constructive comments. This work was supported in part by the U.S. Office of Naval Research (ONR) under Grant N000142212111 and N000142512252.

References

[1] Jenny Abrahamson, Ivan Beschastnikh, Yuriy Brun, and Michael D Ernst. Shedding light on distributed system executions. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 598–599, 2014.

[2] Coşku Acay, Rolph Recto, Joshua Gancher, Andrew C Myers, and Elaine Shi. Viaduct: An Extensible, Optimizing Compiler for Secure Distributed Programs. In *Proceedings of ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 740–755, 2021.

[3] Emma Ahrens, Marius Bozga, Radu Iosif, and Joost-Pieter Katoen. Local Reasoning About Parameterized Reconfigurable Distributed Systems. *Proceedings of the ACM on Programming Languages*, 6(130):145–174, 2022.

[4] Abdel-Rahman Al-Ghuwairi, Yousef Sharrab, Dimah Al-Fraihat, Majed AlElaimat, Ayoub Alsarhan, and Abdulmohsen Algarni. Intrusion Detection in Cloud Computing Based on Time Series Anomalies Utilizing Machine Learning. *Journal of Cloud Computing*, 12(1):127, 2023.

[5] Abdullah Al Maruf, Alexander Bakhtin, Tomas Cerny, and Davide Taibi. Using Microservice Telemetry Data For System Dynamic Analysis. In *International Conference on Service-Oriented System Engineering*, pages 29–38, 2022.

[6] Abdullah Mujawib Alashjaee, Salahaldeen Duraibi, and Jia Song. IoT-Taint: IoT Malware Detection Framework Using Dynamic Taint Analysis. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1220–1223. IEEE, 2019.

[7] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers Principles, Techniques & Tools*. Pearson Education, 2007.

[8] Karim Ali and Ondřej Lhoták. Averroes: Whole-program analysis without the whole program. In *European Conference on Object-Oriented Programming*, pages 378–400. Springer, 2013.

[9] John R Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189, 1983.

[10] Fahad Alqahtani, Salahaldeen Duraibi, Predrag T Tošić, and Frederick T Sheldon. Information Flow Control to Secure Data in the Cloud. In *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1288–1294. IEEE, 2020.

[11] Fahad Alqahtani and Frederick Sheldon. CloudMonitor: Data Flow Filtering as a Service. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1454–1457. IEEE, 2019.

[12] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and David Maier. Blazes: Coordination Analysis and Placement for Distributed Programs. *ACM Transactions on Database Systems (TODS)*, 42(4):1–31, 2017.

[13] Sepehr Amir-Mohammadian. A Semantic Framework for Direct Information Flows in Hybrid-Dynamic Systems. In *Proceedings of the 7th ACM on Cyber-Physical System Security Workshop*, pages 5–15, 2021.

[14] Apache. Voldemort. https://github.com/voldemort, 2024. Accessed: 2024-07-24.

[15] Merve Astekin, Harun Zengin, and Hasan Sözer. Evaluation of Distributed Machine Learning Algorithms for Anomaly Detection from Large-Scale System Logs: A Case Study. In *2018 IEEE international Conference on Big Data (Big Data)*, pages 2071–2077. IEEE, 2018.

[16] Emre Ates, Lily Sturmann, Mert Toslali, Orran Krieger, Richard Megginson, Ayse K Coskun, and Raja R Sambasivan. An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 165–170, 2019.

[17] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, volume 10, pages 1–14, 2010.

[18] Bamberg University. Open Chord. http://sourceforge.net/projects/open-chord/, 2024. Accessed: 2024-07-24.

[19] Soubhagya S. Barpanda and Durga P. Mohapatra. Dynamic slicing of distributed object-oriented programs. *IET software*, 5(5):425–433, 2011.

[20] Christophe Bédard, Pierre-Yves Lajoie, Giovanni Beltrame, and Michel Dagenais. Message Flow Analysis with Complex Causal Links for Distributed ROS 2 Systems. *Robotics and Autonomous Systems*, 161:104361, 2023.

[21] Luciano Bello and Alejandro Russo. Towards a Taint Mode for Cloud Computing Web Applications. In *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*, pages 1–12, 2012.

[22] Zachary Benavides, Keval Vora, and Rajiv Gupta. DProf: Distributed Profiler with Strong Guarantees. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–24, 2019.

[23] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with CSight. pages 468–479, 2014.

[24] Ivan Beschastnikh, Perry Liu, Albert Xing, Patty Wang, Yuriy Brun, and Michael D Ernst. Visualizing Distributed System Executions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(2):1–38, 2020.

[25] Pearl Brereton, Barbara A Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from Applying the Systematic Literature Review Process within the Software Engineering Domain. *Journal of systems and software*, 80(4):571–583, 2007.

[26] Vincent Bushong, Dipta Das, and Tomas Cerny. Reconstructing The Holistic Architecture Of Microservice Systems Using Static Analysis. In *International Conference on Cloud Computing and Services Science*, 2022.

[27] Haipeng Cai. *Cost-effective Dependency Analysis for Reliable Software Evolution.* Ph.d. dissertation, University of Notre Dame, 2015.

[28] Haipeng Cai. Hybrid program dependence approximation for effective dynamic impact prediction. *IEEE Transactions on Software Engineering*, 44(4):334–364, 2017.

[29] Haipeng Cai. A reflection on the predictive accuracy of dynamic impact analysis. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 562–566. IEEE, 2020.

[30] Haipeng Cai and Xiaoqin Fu. D^{2}Abs: A framework for dynamic dependence analysis of distributed programs. *IEEE Transactions on Software Engineering*, 48(12):4733–4761, 2022.

[31] Haipeng Cai, Siyuan Jiang, Raul Santelices, Ying-Jie Zhang, and Yiji Zhang. SensA: Sensitivity analysis for quantitative change-impact prediction. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 165–174. IEEE, 2014.

[32] Haipeng Cai and Raul Santelices. Diver: Precise dynamic impact analysis using dependence-based trace pruning. In *Proceedings of International Conference on Automated Software Engineering*, pages 343–348, 2014.

[33] Haipeng Cai and Raul Santelices. A framework for cost-effective dependence-based dynamic impact analysis. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 231–240. IEEE, 2015.

[34] Haipeng Cai and Raul Santelices. TracerJD: Generic Trace-based Dynamic Dependence Analysis with Fine-grained Logging. In *Proceedings of International Conference on Software Analysis, Evolution, and Reengineering*, pages 489–493, 2015.

[35] Haipeng Cai and Raul Santelices. Method-level program dependence abstraction and its application to impact analysis. *Journal of Systems and Software*, 122:311–326, 2016.

[36] Haipeng Cai, Raul Santelices, and Siyuan Jiang. Prioritizing change-impact analysis via semantic program-dependence quantification. *IEEE Transactions on Reliability*, 65(3):1114–1132, 2015.

[37] Haipeng Cai, Raul Santelices, and Douglas Thain. DiaPro: Unifying Dynamic Impact Analyses for Improved and Variable Cost-Effectiveness. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(2):1–50, 2016.

[38] Haipeng Cai and Douglas Thain. DistEA: Efficient Dynamic Impact Analysis for Distributed Systems. *arXiv preprint arXiv:1604.04638*, 2016.

[39] Haipeng Cai and Douglas Thain. DistIA: A Cost-Effective Dynamic Impact Analysis for Distributed Programs. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 344–355. IEEE, 2016.

[40] Apache Cassandra. Apache cassandra. *Website. Available online at http://planetcassandra.org/what-is-apache-cassandra*, 13, 2014.

[41] Z Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. Sensitive Information Tracking in Commodity IoT. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1687–1704, 2018.

[42] Tomas Cerny, Amr S Abdelfattah, Vincent Bushong, Abdullah Al Maruf, and Davide Taibi. Microvision: Static Analysis-Based Approach To Visualizing Microservices In Augmented Reality. In *International Conference on Service-Oriented System Engineering*, pages 49–58, 2022.

[43] Tomas Cerny and Davide Taibi. Microservice-Aware Static Analysis: Opportunities, Gaps, and Advancements. In *Joint Post-Proceedings of the International Conference on Microservices*, 2023.

[44] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.

[45] Rong Chang, Chuanxu Yang, and Yuankang Lei. An adaptive selection method for repair nodes in distributed storage systems. In *Journal of Physics: Conference Series*, volume 1744, page 032209. IOP Publishing, 2021.

[46] Ned Chapin, Joanne E Hale, Khaled Md Khan, Juan F Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, 13(1):3–30, 2001.

[47] Animesh Chaturvedi, Aruna Tiwari, Dave Binkley, and Shubhangi Chaturvedi. Service Evolution Analytics: Change and Evolution Mining of a Distributed System. *IEEE Transactions on Engineering Management*, 68(1):137–148, 2020.

[48] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, and Ming Wen. Automatic Root Cause Analysis via Large Language Models for Cloud Incidents. In *European Conference on Computer Systems*, pages 674–688, 2024.

[49] Maria Christakis, Alkis Gotovos, and Konstantinos Sagonas. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 154–163. IEEE, 2013.

[50] Laurent Christophe, Coen De Roover, Elisa Gonzalez Boix, and Wolfgang De Meuter. Orchestrating Dynamic Analyses of Distributed Processes for Full-Stack JavaScript Programs. *ACM SIGPLAN Notices*, 53(9):107–118, 2018.

[51] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[52] Marshall Copeland, Julian Soh, Anthony Puca, Mike Manning, and David Gollob. Microsoft azure. *New York, NY, USA:: Apress*, 2015.

[53] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[54] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, and Roberto Natella. Run-Time Failure Detection via Non-Intrusive Event Analysis in a Large-Scale Cloud Computing Platform. *Journal of Systems and Software*, 198:111611, 2023.

[55] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design.* Addison-Wesley Publishing Company, 5th edition, 2011.

[56] Valentin Cristea, Florin Pop, Ciprian Dobre, and Alexandru Costan. Distributed Architectures for Event-Based Systems. In *Reasoning in event-based distributed systems*, pages 11–45. Springer, 2011.

[57] Ismet Dagli, Andrew Depke, Andrew Mueller, Md Sahil Hassan, Ali Akoglu, and Mehmet Esat Belviranli. Contention-Aware Performance Modeling for Heterogeneous Edge and Cloud Systems. In *Proceedings of the Workshop on Flexible Resource and Application Management on the Edge*, pages 27–31, 2023.

[58] Cleidson RB de Souza and David F Redmiles. An Empirical Study Of Software Developers' Management Of Dependencies And Changes. In *Proceedings of the 30th international conference on Software engineering*, pages 241–250, 2008.

[59] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[60] Dorothy E Denning and Peter J Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977.

[61] Chandan Dhal, Xiaoqin Fu, and Haipeng Cai. A Control-Theoretic Approach to Auto-Tuning Dynamic Analysis for Distributed Services. In *IEEE/ACM International Conference on Software Engineering: Companion*, pages 330–331, 2023.

[62] Praveen M Dhulavvagol, Vijayakumar H Bhajantri, and SG Totad. Performance Analysis of Distributed Processing System using Shard Selection Techniques on Elasticsearch. *Procedia Computer Science*, 167:1626–1635, 2020.

[63] Piergiuseppe Di Pilla, Remo Pareschi, Francesco Salzano, and Federico Zappone. Listening to What the System Tells Us: Innovative Auditing for Distributed Systems. *Frontiers in Computer Science*, 4:1020946, 2023.

[64] Luca Di Stefano, Rocco De Nicola, and Omar Inverso. Verification Of Distributed Systems Via Sequential Emulation. *ACM Transactions on Software Engineering and Methodology*, 31(3):1–41, 2022.

[65] Edsger W Dijkstra, Wim HJ Feijen, and A.J.M Van Gasteren. Derivation of a Termination Detection Algorithm for Distributed Computations. *Information processing letters*, 16(5):217–219, 1983.

[66] Edsger W Dijkstra, Wim HJ Feijen, and A.J.M Van Gasteren. Derivation of a Termination Detection Algorithm for Distributed Computations. In *Control Flow and Data Flow: concepts of distributed programming*, pages 507–512. Springer, 1986.

[67] Frank Doelitzscher. Security Audit Compliance for Cloud Computing, 2014.

[68] Pradeep Dogga, Karthik Narasimhan, Anirudh Sivaraman, Shiv Saini, George Varghese, and Ravi Netravali. Revelio: Ml-Generated Debugging Queries For Finding Root Causes In Distributed Systems. *Proceedings of Machine Learning and Systems*, 4:601–622, 2022.

[69] Mohsen Dorodchi, Maryam Abedi, and Bojan Cukic. Trust-based Development Framework for Distributed Systems and IoT. In *Proceedings of the 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 437–442. IEEE, 2016.

[70] Tânia Esteves, Francisco Neves, Rui Oliveira, and João Paulo. Cat: content-aware tracing and analysis for distributed systems. In *Proceedings of the 22nd International Middleware Conference*, pages 223–235, 2021.

[71] Hongzhou Fang, Yuanfang Cai, Rick Kazman, and Jason Lefever. Identifying Anti-Patterns in Distributed Systems with Heterogeneous Dependencies. In *International Conference on Software Architecture: Companion*, pages 116–120, 2023.

[72] Wenhan Feng, Qiugen Pei, Yu Gao, Dong Wang, Wensheng Dou, Jun Wei, Zheheng Liang, and Zhenyue Long. FaultFuzz: A Coverage Guided Fault Injection Tool for Distributed Systems. In *IEEE/ACM International Conference on Software Engineering: Companion*, pages 129–133, 2024.

[73] Jay Fenlason and Richard Stallman. GNU gprof. *GNU Binutils. Available online: http://www.gnu.org/software/binutils*, 1988.

[74] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[75] Colin J Fidge. Timestamps in message-passing systems that preserve the partial ordering, 1987.

[76] Xiaoqin Fu. On the Scalable Dynamic Taint Analysis for Distributed Systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1247–1249, 2019.

[77] Xiaoqin Fu. Towards Scalable Defense of Information Flow Security for Distributed Systems. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 438–442, 2019.

[78] Xiaoqin Fu. *Scalable and Cost-Effective Data Flow Analysis for Distributed Software: Algorithms and Applications*. Ph.d. thesis, Washington State University, 2022.

[79] Xiaoqin Fu and Haipeng Cai. A Dynamic Taint Analyzer for Distributed Systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1115–1119, 2019.

[80] Xiaoqin Fu and Haipeng Cai. Measuring Interprocess Communications in Distributed Systems. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 323–334. IEEE, 2019.

[81] Xiaoqin Fu and Haipeng Cai. Scaling Application-Level Dynamic Taint Analysis to Enterprise-Scale Distributed Systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pages 270–271, 2020.

[82] Xiaoqin Fu and Haipeng Cai. FlowDist: Multi-Staged Refinement-Based Dynamic Information Flow Analysis for Distributed Software Systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2093–2110, 2021.

[83] Xiaoqin Fu, Haipeng Cai, and Li Li. Dads: Dynamic Slicing Continuously-Running Distributed Programs with Budget Constraints. In *Proceedings of European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1566–1570, 2020.

[84] Xiaoqin Fu, Haipeng Cai, Wen Li, and Li Li. Seads: Scalable and Cost-effective Dynamic Dependence Analysis of Distributed Systems via Reinforcement Learning. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(1):1–45, 2020.

[85] Xiaoqin Fu, Boxiang Lin, and Haipeng Cai. DistFax: A Toolkit For Measuring Interprocess Communications And Quality Of Distributed Systems. In *IEEE/ACM International Conference on Software Engineering: Companion*, pages 51–55, 2022.

[86] Xiaoqin Fu, Asif Zaman, and Haipeng Cai. DistMeasure: A framework for runtime characterization and quality assessment of distributed software via interprocess communications. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–53, 2025.

[87] Munehiro Fukuda, Collin Gordon, Utku Mert, and Matthew Sell. An Agent-Based Computational Framework for Distributed Data Analysis. *Computer*, 53(3):16–25, 2020.

[88] Daniel Galin. *Software Quality Assurance: From Theory to Implementation.* Addison-Wesley, 2003.

[89] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed Systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 539–550, 2018.

[90] Diego Garbervetsky, Edgardo Zoppi, and Benjamin Livshits. Toward Full Elasticity in Distributed Static Analysis: The Case of Callgraph Analysis. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 442–453, 2017.

[91] Joshua Garcia, Daniel Popescu, Gholamreza Safi, William GJ Halfond, and Nenad Medvidovic. Identifying Message Flow in Distributed Event-Based Systems. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 367–377, 2013.

[92] Manuel Gieseking. *Correctness Of Data Flows In Asynchronous Distributed Systems: Model Checking And Synthesis.* PhD thesis, Universität Oldenburg, 2022.

[93] Anitha Gollamudi, Stephen Chong, and Owen Arden. Information Flow Control for Distributed Trusted Execution Environments. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 304–30414. IEEE, 2019.

[94] Susan L Graham, Peter B Kessler, and Marshall K McKusick. gprof: a Call Graph Execution Profiler. *ACM Sigplan Notices*, 39(4):49–57, 2004.

[95] Jiawei Tyler Gu, Xudong Sun, Wentao Zhang, Yuxuan Jiang, Chen Wang, Mandana Vaziri, Owolabi Legunsen, and Tianyin Xu. Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management. In *ACM Symposium on Operating Systems Principles*, pages 96–112, 2023.

[96] Rong Gu, Zhiqiang Zuo, Xi Jiang, Han Yin, Zhaokang Wang, Linzhang Wang, Xuandong Li, and Yihua Huang. Towards Efficient Large-Scale Interprocedural Program Static Analysis on Distributed Data-Parallel Computation. *IEEE Transactions on Parallel and Distributed Systems*,

32(4):867–883, 2020.

[97] Shushi Gu, Fugang Wang, Qinyu Zhang, Tao Huang, and Wei Xiang. Global Repair Bandwidth Cost Optimization of Generalized Regenerating Codes in Clustered Distributed Storage Systems. *IET Communications*, 15:2469–2481, 2021.

[98] Zhenyu Guo, Dong Zhou, Haoxiang Lin, Mao Yang, Fan Long, Chaoqiang Deng, Changshu Liu, and Lidong Zhou. G2: A Graph Processing System for Diagnosing Distributed Systems. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[99] Christophe Hauser, Frédéric Tronel, Colin Fidge, and Ludovic Mé. Intrusion Detection in Distributed Systems, an Approach based on Taint Marking. In *2013 IEEE International Conference on Communications (ICC)*, pages 1962–1967. IEEE, 2013.

[100] Hongxia He, Xi Li, Peng Chen, Juan Chen, Ming Liu, and Lei Wu. Efficiently Localizing System Anomalies for Cloud Infrastructures: A Novel Dynamic Graph Transformer based Parallel Framework. *Journal of Cloud Computing*, 13(1):115, 2024.

[101] Michael Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

[102] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[103] Junjie Huang, Jinyang Liu, Zhuangbin Chen, Zhihan Jiang, Yichen Li, Jiazhen Gu, Cong Feng, Zengyin Yang, Yongqiang Yang, and Michael R Lyu. FaultProfIT: Hierarchical Fault Profiling of Incident Tickets in Large-scale Cloud Systems. In *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*, pages 392–404, 2024.

[104] Lexiang Huang and Timothy Zhu. tprof: Performance Profiling via Structural Aggregation and Automated Analysis of Distributed Systems Traces. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 76–91, 2021.

[105] Yu Huang, Tao Wang, Zihui Yin, Eric Mercer, and Benjamin Ogles. Improving The Efficiency Of Deadlock Detection In MPI Programs Through Trace Compression. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):400–415, 2022.

[106] ISO/IEC. ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010, 2011. Accessed: 2024-07-24.

[107] KR Jayaram and Patrick Eugster. Program Analysis for Event-based Distributed Systems. In *ACM International Conference on Distributed Event-based System*, pages 113–124, 2011.

[108] Danlin Jia, Manoj Pravakar Saha, Janki Bhimani, and Ningfang Mi. Performance and Consistency Analysis for Distributed Deep Learning Applications. In *2020 IEEE 39th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2020.

[109] Yijiang Jia, Kefei Li, Ping Lu, Baodi Xie, Huayang Wang, and Jincai Chen. An Anomaly Detection Method Based on Adaptive Log and Dual Feature Fusion Analysis for Distributed Systems. In *International Conference on Information Systems and Computing Technology*, pages 111–114, 2023.

[110] JIRA. Cassandra-6023. https://issues.apache.org/jira/browse/CASSANDRA6023. Accessed: 2024-07-24.

[111] Pallavi Joshi, Malay Ganai, Gogul Balakrishnan, Aarti Gupta, and Nadia Papakonstantinou. SETSUDŌ: Perturbation-based Testing Framework for Scalable Distributed Systems. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, pages 1–14, 2013.

[112] Mariam Kamkar and Patrik Krajina. Dynamic Slicing of Distributed Programs. In *Proceedings of International Conference on Software Maintenance*, pages 222–229. IEEE, 1995.

[113] Byeongjee Kang and Kyungmin Bae. Symbolic Reachability Analysis Of Distributed Systems Using Narrowing And Heuristic Search. In *International Workshop on Formal Techniques for Safety-Critical Systems*, pages 34–44, 2022.

[114] Florian Kelbert and Alexander Pretschner. Data Usage Control for Distributed Systems. *ACM Transactions on Privacy and Security (TOPS)*, 21(3):1–32, 2018.

[115] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Acm Sigplan Notices*, volume 47, pages 121–132. ACM, 2012.

[116] Anum Khurshid, Abdul Nasir Khan, Fiaz Gul Khan, Mazhar Ali, Junaid Shuja, and Atta ur Rehman Khan. Secure-CamFlow: A Device-Oriented Security Model to Assist Information Flow Control Systems in Cloud Environments for IoTs. *Concurrency and Computation: Practice and Experience*, 31(8):e4729, 2019.

[117] Jemin Kim and Joonseok Park. Enhancing Security of Web-Based IoT Services via XSS Vulnerability Detection. *Sensors*, 23(23):9407, 2023.

[118] Fabian Knorr, Peter Thoman, and Thomas Fahringer. Declarative Data Flow in a Graph-Based Distributed Memory Runtime System. *International Journal of Parallel Programming*, 51(2):150–171, 2023.

[119] Fabio Kon and Roy H Campbell. Dependence Management in Component-Based Distributed Systems. *IEEE Concurrency*, 8(1):26–36, 2000.

[120] Jacob Kreindl, Daniele Bonetta, and Hanspeter Mössenböck. Towards Efficient, Multi-Language Dynamic Taint Analysis. In *ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, pages 85–94, 2019.

[121] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems.* Cambridge University Press, 2011.

[122] Jinxi Kuang, Jinyang Liu, Junjie Huang, Renyi Zhong, Jiazhen Gu, Lan Yu, Rui Tan, Zengyin Yang, and Michael R Lyu. Knowledge-aware Alert Aggregation in Large-scale Cloud Systems: a Hybrid Approach. In *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*, pages 369–380, 2024.

[123] Michał Kucab, Piotr Boryło, and Piotr Chołda. Hardware-Assisted Static and Runtime Attestation for Cloud Deployments. *IEEE Transactions on Cloud Computing*, 2023.

[124] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[125] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 312–324, 2019.

[126] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[127] Edward Lee and Ivan Levchenko. Applying a Deterministic Approach for Distributed Systems Black-Box Testing. Technical report, Stanford University, 2020.

[128] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 517–530, 2016.

[129] Guangpu Li, Haopeng Liu, Xianglan Chen, Haryadi S Gunawi, and Shan Lu. DFix: Automatically Fixing Timing Bugs in Distributed Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 994–1009, 2019.

[130] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. PCatch: Automatically Detecting Performance Cascading Bugs. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[131] Wen Li, Li Li, and Haipeng Cai. On the Vulnerability Proneness of Multilingual Code. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 847–859, 2022.

[132] Wen Li, Li Li, and Haipeng Cai. PolyFax: A Toolkit for Characterizing Multi-Language Software. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Tool Demos*, pages 1662–1666, 2022.

[133] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai. PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[134] Bingzhe Liu, Gangmuk Lim, Ryan Beckett, and P Brighten Godfrey. Kivi: Verification for Cluster Management. In *USENIX Annual Technical Conference*, pages 509–527, 2024.

[135] Haopeng Liu. *Modeling and Tackling Timing Bugs in Multi-threaded Systems and Distributed Systems.* PhD thesis, The University of Chicago, 2019.

[136] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. *ACM SIGARCH Computer Architecture News*, 45(1):677–691, 2017.

[137] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems. *ACM SIGPLAN Notices*, 53(2):419–431, 2018.

[138] Jinyang Liu, Shilin He, Zhuangbin Chen, Liqun Li, Yu Kang, Xu Zhang, Pinjia He, Hongyu Zhang, Qingwei Lin, and Zhangwei Xu. Incident-Aware Duplicate Ticket Aggregation for Cloud Systems. In *IEEE/ACM International Conference on Software Engineering*, pages 2299–2311, 2023.

[139] Jinyang Liu, Zhihan Jiang, Jiazhen Gu, Junjie Huang, Zhuangbin Chen, Cong Feng, Zengyin Yang, Yongqiang Yang, and Michael R Lyu. Prism: Revealing Hidden Functional Clusters from Massive Instances in Cloud Systems. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 268–280, 2023.

[140] Panos E Livadas and Theodore Johnson. An Optimal Algorithm for the Construction of the System Dependence Graph. *Information Sciences*, 125(1-4):99–131, 2000.

[141] Teng Long, Xingtao Ren, Qing Wang, and Chao Wang. Verifying The Safety Properties Of Distributed Systems Via Mergeable Parallelism. *Journal of Systems Architecture*, 130:102646, 2022.

[142] Jie Lu, Haofeng Li, Chen Liu, Lian Li, and Kun Cheng. Detecting Missing-Permission-Check Vulnerabilities In Distributed Cloud Systems. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 2145–2158, 2022.

[143] Jie Lu, Chen Liu, Feng Li, Lian Li, Xiaobing Feng, and Jingling Xue. CloudRaid: Detecting Distributed Concurrency Bugs via Log Mining and Enhancement. *IEEE Transactions on Software Engineering*, 48(2):662–677, 2020.

[144] Shangyun Lu, Bradley Nott, Aaron Olson, Alberto Todeschini, Hossein Vahabi, Yair Carmon, and Ludwig Schmidt. Harder or Different? A Closer Look at Distribution Shift in Dataset Reproduction. In *ICML Workshop on Uncertainty and Robustness in Deep Learning*, volume 5, page 15, 2020.

[145] Nyalia Lui and James H Hill. A Generalized Approach for Non-Intrusive Real-Time Instrumentation of Standards-Based Distributed Middleware. In *IEEE International Symposium on Real-Time Distributed Computing (ISORC)*, pages 158–166. IEEE, 2020.

[146] Jeffrey F Lukman, Huan Ke, Cesar A Stuardo, Riza O Suminto, Daniar H Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, et al. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[147] Yu Luo. *approf: A Non-intrusive Call Hierarchy Profiler for Distributed Systems.* PhD thesis, University of Toronto (Canada), 2018.

[148] Tao Lyu, Liyi Zhang, Zhiyao Feng, Yueyang Pan, Yujie Ren, Meng Xu, Mathias Payer, and Sanidhya Kashyap. Monarch: A Fuzzing Framework for Distributed File Systems. In *USENIX Annual Technical Conference*, pages 529–543, 2024.

[149] Haojun Ma, Hammad Ahmad, Aman Goel, Eli Goldweber, Jean-Baptiste Jeannin, Manos Kapritsos, and Baris Kasikci. Sift: Using Refinement-Guided Automation To Verify Complex Distributed Systems. In *USENIX Annual Technical Conference*, pages 151–166, 2022.

[150] Jonathan Mace and Rodrigo Fonseca. Universal Context Propagation for Distributed System Instrumentation. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–18, 2018.

[151] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 378–393, 2015.

[152] Somayya Madakam, Vihar Lake, Vihar Lake, Vihar Lake, et al. Internet of Things (IoT): A Literature Review. *Journal of Computer and Communications*, 3(05):164, 2015.

[153] Ilias Maglogiannis, Constantinos Delakouridis, and Leonidas Kazatzopoulos. Enabling Collaborative Medical Diagnosis Over the Internet via Peer-to-Peer Distribution of Electronic Health Records. *Journal of Medical Systems*, 30(2):107–116, 2006.

[154] Amit Mandal, Pietro Ferrara, Yuliy Khlyebnikov, Agostino Cortesi, and Fausto Spoto. Cross-Program Taint Analysis for IoT Systems. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 1944–1952, 2020.

[155] Francisco Maturana and KV Rashmi. Bandwidth Cost of Code Conversions in Distributed Storage: Fundamental Limits and Optimal Constructions. In *2021 IEEE International Symposium on Information Theory (ISIT)*, pages 2334–2339. IEEE, 2021.

[156] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. Greybox Fuzzing of Distributed Systems. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1615–1629, 2023.

[157] Barton P Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. *ACM Sigplan Notices*, 23(7):135–144, 1988.

[158] VP Mochalov, N Yu Bratchenko, and SV Yakovlev. Analytical Model of Integration System for Program Components of Distributed Object Applications. In *2018 International Russian Automation Conference (RusAutoCon)*, pages 1–4. IEEE, 2018.

[159] Durga P Mohapatra, Rajeev Kumar, Rajib Mall, DS Kumar, and Mayank Bhasin. Distributed Dynamic Slicing of Java Programs. *Journal of Systems and Software*, 79(12):1661–1678, 2006.

[160] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-based Systems*. Springer Science & Business Media, 2006.

[161] Gail C Murphy and David Notkin. Lightweight Lexical Source Model Extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, 1996.

[162] Francisco Neves, Nuno Machado, et al. Falcon: A Practical Log-based Analysis Tool for Distributed Systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 534–541. IEEE, 2018.

[163] Francisco Neves, Nuno Machado, Ricardo Vilaça, and José Pereira. Horus: Non-Intrusive Causal Analysis of Distributed Systems Logs. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 212–223. IEEE, 2021.

[164] Francisco Neves, Ricardo Vilaça, and José Pereira. Detailed Black-Box Monitoring of Distributed Systems. *ACM SIGAPP Applied Computing Review*, 21:24–36, 2021.

[165] Zack Newsham, Augusto Born De Oliveira, Jean-Christophe Petkovich, Ahmad Saif Ur Rehman, Guy Martin Tchamgoue, and Sebastian Fischmeister. Intersert: Assertions on Distributed Process Interaction Sessions. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 216–223. IEEE, 2017.

[166] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Science & Business Media, 2004.

[167] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. Generating Realistic Vulnerabilities via Neural Code Editing: An Empirical Study. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1097–1109, 2022.

[168] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. VulGen: Realistic Vulnerable Sample Generation via Pattern Mining and Deep Learning. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023.

[169] Yu Nong, Rainy Sharma, Abdelwahab Hamou-Lhadj, Xiapu Luo, and Haipeng Cai. Open science in software engineering: A study on deep learning-based vulnerability detection. *IEEE Transactions on Software Engineering*, 49(4):1983–2005, 2022.

[170] Alberto Núñez, Pablo C Cañizares, Pablo Gómez-Abajo, Esther Guerra, and Juan de Lara. Analyzing The Reliability Of Simulated Distributed Systems Using Metamorphic Testing. In *International Workshop on Metamorphic Testing*, pages 34–41, 2022.

[171] Yicheng Ouyang, Kailai Shao, Kunqiu Chen, Ruobing Shen, Chao Chen, Mingze Xu, Yuqun Zhang, and Lingming Zhang. MirrorTaint: Practical Non-Intrusive Dynamic Taint Tracking for JVM-Based Microservice Systems. In *IEEE/ACM International Conference on Software Engineering*, pages 2514–2526, 2023.

[172] Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. Trace Aware Random Testing for Distributed Systems. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[173] Muhammed Maruf Öztürk. MFRLMO: Model-Free Reinforcement Learning for Multi-Objective Optimization of Apache Spark. *EAI Endorsed Transactions on Scalable Information Systems*, 11(5), 2024.

[174] Ioannis Papagiannis and Peter Pietzuch. CloudFilter: Practical Control of Sensitive Data Propagation to the Cloud. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, pages 97–102, 2012.

[175] Vasilis Pappas, Vasileios P Kemerlis, Angeliki Zavou, Michalis Polychronakis, and Angelos D Keromytis. CloudFence: Data Flow Tracking as a Cloud Service. In *International Workshop on Recent Advances in Intrusion Detection*, pages 411–431. Springer, 2013.

[176] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. A Qualitative Study of Dependency Management and Its Security Implications. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1513–1531, 2020.

[177] Thomas FJM Pasquier, Jean Bacon, and David Eyers. FlowK: Information Flow Control for the Cloud. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 70–77. IEEE, 2014.

[178] João Carlos Pereira, Nuno Machado, and Jorge Sousa Pinto. Testing for Race Conditions in Distributed Systems via SMT Solving. In *International Conference on Tests and Proofs*, pages 122–140. Springer, 2020.

[179] Daniel Popescu, Joshua Garcia, Kevin Bierhoff, and Nenad Medvidovic. Impact Analysis for Distributed Event-Based Systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 241–251, 2012.

[180] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education, 8th edition, 2014.

[181] Kleanthis Psarris and Konstantinos Kyriakopoulos. An Experimental Evaluation of Data Dependence Analysis Techniques. *IEEE Transactions on Parallel and Distributed Systems*, 15(3):196–213, 2004.

[182] Sheng Qi, Xuanzhe Liu, and Xin Jin. Halfmoon: Log-Optimal Fault-Tolerant Stateful Serverless Computing. In *ACM Symposium on Operating Systems Principles*, pages 314–330, 2023.

[183] Noëlle Rakotondravony, Johannes Köstler, and Hans P Reiser. Towards a Generic Architecture for Interactive Cost-Aware Visualization of Monitoring Data in Distributed Systems. In *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems*, pages 25–30, 2017.

[184] Simon Robillard and Hélène Coullon. SMT-Based Planning Synthesis For Distributed System Reconfigurations. In *FASE*, pages 268–287, 2022.

[185] Yineng Rong and X San Liang. Panel Data Causal Inference Using a Rigorous Information Flow Analysis for Homogeneous, Independent and Identically Distributed Datasets. *IEEE Access*, 9:47266–47274, 2021.

[186] Barbara Ryder. Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages. In *Compiler Construction*, pages 126–137, 2003.

[187] Barbara G Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, 3:216–226, 1979.

[188] Malte Sandstede. Online Analysis of Distributed Dataflows with Timely Dataflow. *arXiv preprint arXiv:1912.09747*, 2019.

[189] Nikolaos Sapountzis, Ruimin Sun, Xuetao Wei, Yier Jin, Jedidiah Crandall, and Daniela Oliveira. MITOS: Optimal Decisioning for the Indirect Flow Propagation Dilemma in Dynamic Information Flow Tracking Systems. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1090–1100. IEEE, 2020.

[190] Micah Schiewe, Jacob Curtis, Vincent Bushong, and Tomas Cerny. Advancing Static Code Analysis With Language-Agnostic Component Identification. *IEEE Access*, 10:30743–30761, 2022.

[191] Mariana Sharp and Atanas Rountev. Static Analysis of Object References in RMI-based Java Software. *IEEE Transactions on Software Engineering*, 32(9):664–681, 2006.

[192] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things*, 3(5):637–646, 2016.

[193] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, 2010.

[194] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic Evaluation of Distributed Systems. In *5th International Workshop on Systems Software Verification (SSV 10)*, 2010.

[195] Marjan Sirjani, Ehsan Khamespanah, and Fatemeh Ghassemi. Reactive Actors: Isolation for Efficient Analysis of Distributed Systems. In *2019 IEEE/ACM 23rd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 1–10. IEEE, 2019.

[196] Yannis Smaragdakis and George Balatsouras. Pointer Analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.

[197] SourceForge. NioEcho. http://rox-xmlrpc.sourceforge.net/niotut/index.html#Thecode, 2024. Accessed: 2024-07-24.

[198] Mark Staples and Mahmood Niazi. Experiences Using Systematic Review Guidelines. *Journal of Systems and Software*, 80(9):1425–1437, 2007.

[199] Cesar A Stuardo, Tanakorn Leesatapornwongsa, Riza O Suminto, Huan Ke, Jeffrey F Lukman, Wei-Chiu Chuang, Shan Lu, and Haryadi S Gunawi. ScaleCheck: A Single-Machine Approach for Discovering Scalability Bugs in Large Distributed Systems. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 359–373, 2019.

[200] Riza O. Suminto, Shan Lu, Cindy Rubio-González, and Haryadi S. Gunawi. Database-Backed Program Analysis for Finding Cascading Outage Bugs in Distributed Systems. Technical report, University of Chicago, 2021.

[201] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. Anvil: Verifying Liveness of Cluster Management Controllers. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 649–666, 2024.

[202] Zhuo Sun. *A Method and Tool for Finding Concurrency Bugs Involving Multiple Variables with Application to Modern Distributed Systems*. PhD thesis, Florida International University, 2018.

[203] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Pearson Education, 2nd edition, 2007.

[204] Ruize Tang, Xudong Sun, Yu Huang, Yuyang Wei, Lingzhi Ouyang, and Xiaoxing Ma. SandTable: Scalable Distributed System Model Checking with Specification-Level State Exploration. In *European Conference on Computer Systems*, pages 736–753, 2024.

[205] WALA Team. Wala: A static analysis framework. https://github.com/wala/WALA, 2024. Accessed: 2024-07-24.

[206] Fernando A Teixeira, Fernando MQ Pereira, Hao-Chi Wong, José MS Nogueira, and Leonardo B Oliveira. SIoT: Securing Internet of Things through Distributed Systems Analysis. *Future Generation Computer Systems*, 92:1172–1186, 2019.

[207] Fernando Augusto Teixeira et al. *Securing Networked Embedded Systems Through Distributed Systems Analysis*. PhD thesis, Universidade Federal de Minas Gerais, 2015.

[208] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. Stardust: Tracking Activity in a Distributed Storage System. *ACM SIGMETRICS Performance Evaluation Review*, 34(1):3–14, 2006.

[209] Jeff Tian. *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*. Wiley-Interscience, 1st edition, 2005.

[210] Shukun Tokas, Olaf Owe, and Toktam Ramezanifarkhani. Static Checking Of GDPR-Related Privacy Compliance For Object-Oriented Distributed Systems. *Journal of Logical and Algebraic Methods in Programming*, 125:100733, 2022.

[211] Mert Toslali, Emre Ates, Alex Ellis, Zhaoqi Zhang, Darby Huye, Lan Liu, Samantha Puterman, Ayse K Coskun, and Raja R Sambasivan. Automating Instrumentation Choices for Performance Problems in Distributed Applications with VAIF. In *Proceedings of the ACM Symposium on Cloud*

Computing, pages 61–75, 2021.

[212] Simon Tragatschnig, Huy Tran, and Uwe Zdun. Impact Analysis for Event-based Systems using Change Patterns. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 763–768, 2014.

[213] Vidhya Tekken Valapil. *Achieving Reliable Distributed Systems: Through Efficient Run-time Monitoring and Predicate Detection.* Michigan State University, 2020.

[214] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. In *CASCON First Decade High Impact Papers*, pages 214–224, 2010.

[215] Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A Multiparty Session Typing Discipline for Fault-Tolerant Event-Driven Distributed Programming. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021.

[216] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. Model Checking Guided Testing for Distributed Systems. In *European Conference on Computer Systems*, pages 127–143, 2023.

[217] Dong Wang, Yu Gao, Wensheng Dou, and Jun Wei. DisTA: Generic Dynamic Taint Tracking For Java-Based Distributed Systems. In *International Conference on Dependable Systems and Networks*, pages 547–558, 2022.

[218] Hongyi Wang, Qingfeng Jing, Rishan Chen, Bingsheng He, Zhengping Qian, and Lidong Zhou. Distributed Systems Meet Economics: Pricing in the Cloud. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

[219] Jie Wang, Yunguang Wu, Gang Zhou, Yiming Yu, Zhenyu Guo, and Yingfei Xiong. Scaling Static Taint Analysis to Industrial SOA Applications: A Case Study at Alibaba. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1477–1486, 2020.

[220] Nan Wang and Aimin Pan. Research and Implementation of New Distributed Information Interactive Platform. In *Journal of Physics: Conference Series*, volume 1883, page 012061. IOP Publishing, 2021.

[221] Dawei Wei, Huansheng Ning, Feifei Shi, Yueliang Wan, Jiabo Xu, Shunkun Yang, and Li Zhu. Dataflow Management in the Internet of Things: Sensing, Control, and Security. *Tsinghua Science and Technology*, 26(6):918–930, 2021.

[222] Bill Wilder. *Cloud Architecture Patterns: using Microsoft Azure.* O'Reilly Media, Inc., 2012.

[223] Wolfgang Wögerer. A Survey of Static Program Analysis Techniques. Technical report, Technische Universität Wien, 2005.

[224] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A Survey on Software Fault Localization. *TSE*, 42(8):707–740, 2016.

[225] Hongfa Xue, Shaowen Sun, Guru Venkataramani, and Tian Lan. Machine Learning-Based Analysis of Program Binaries: A Comprehensive Study. *IEEE Access*, 7:65889–65912, 2019.

[226] Cui-Qing Yang and Barton P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *International Conference on Distributed Computing Systems*, pages 366–373, 1988.

[227] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI'09*, pages 213–228, 2009.

[228] Tuba Yavuz and Christopher Brant. Security Analysis Of IoT Frameworks Using Static Taint Analysis. In *ACM Conference on Data and Application Security and Privacy*, pages 203–213, 2022.

[229] Sorrachai Yingchareonthawornchai, Duong N Nguyen, Vidhya Tekken Valapil, Sandeep S Kulkarni, and Murat Demirbas. Precision, Recall, and Sensitivity of Monitoring Partially Synchronous Distributed Systems. In *International Conference on Runtime Verification*, pages 420–435, 2016.

[230] Mingguang Yu and Xia Zhang. Anomaly Detection for Cloud Systems with Dynamic Spatiotemporal Learning. *Intelligent Automation & Soft Computing*, 37(2), 2023.

[231] Jinfeng Yuan, Weizhong Qiang, Hai Jin, and Deqing Zou. CloudTaint: An Elastic Taint Tracking Framework for Malware Detection in the Cloud. *The Journal of Supercomputing*, 70(3):1433–1450, 2014.

[232] Xinhao Yuan and Junfeng Yang. Effective Concurrency Testing for Distributed Systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1141–1156, 2020.

[233] Angeliki Zavou, Vasilis Pappas, Vasileios P Kemerlis, Michalis Polychronakis, Georgios Portokalidis, and Angelos D Keromytis. Cloudopsy: An Autopsy of Data Flows in the Cloud. In *International Conference on Human Aspects of Information Security, Privacy, and Trust*, pages 366–375, 2013.

[234] Shudong Zhang, Dongxue Liu, Lijuan Zhou, Zhongshan Ren, and Zipeng Wang. Diagnostic Framework for Distributed Application Performance Anomaly Based on Adaptive Instrumentation. In *International Conference on Computer Communication and the Internet*, pages 164–169, 2020.

[235] Tony Nuda Zhang, Upamanyu Sharma, and Manos Kapritsos. Performal: Formal Verification of Latency Properties for Distributed Systems. *Proceedings of the ACM on Programming Languages*, 7(PLDI):368–393, 2023.

[236] Yuhao Zhang, Frank McQuillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano, and Arun Kumar. Distributed Deep Learning on Data Systems: A Comparative Analysis of Approaches. *Proceedings of the VLDB Endowment*, 14(10):1769–1782, 2021.

[237] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 629–644, 2014.

[238] Chenxing Zhong, He Zhang, Chao Li, Huang Huang, and Daniel Feitosa. On Measuring Coupling Between Microservices. *Journal of Systems and Software*, 200:111670, 2023.

[239] Zexin Zhong, Jiangchao Liu, Diyu Wu, Peng Di, Yulei Sui, Alex X Liu, and John CS Lui. Scalable Compositional Static Taint Analysis for Sensitive Data Tracing on Industrial Micro-Services. In *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*, pages 110–121, 2023.

A More Background on Program Analysis and Distributed Systems

In this appendix section, we provide additional background regarding our survey topic, concerning concepts and algorithms in program analysis and charactristics of distributed software systems that are relevant to the analysis of distributed programs.

A.1 Fundamental Analysis Concepts and Algorithms

Among the large variety of program analyses, there are a relatively few ones that serve in supporting/enabling roles, referred to as *fundamental analyses*. Accordingly, the supported/enabled analyses are *client/application analyses*. Below we exemplify such analyses with two instances: pointer/alias analysis and dependence analysis.

A.1.1 Pointer Analysis and Alias Analysis. Pointer analysis (i.e., points-to or reference analysis) is a type of static program analysis that resolves which memory locations (objects) a pointer/reference variable (i.e., pointers) can point to. It approximately computes the set of such objects and their relationships to determine the possible run-time values of pointers [101]. Then, the resulting information can be used to determine the aliasing relationships in the program [196] (i.e., alias analysis). If two variables point to the same memory location, they are considered *aliases* to each other. Changing the value pointed to by one of these two pointer variables would indirectly change the value pointed to by the other. For example, as shown in Figure 1, in the NioEcho client (class NioClient), the statement at Line 7 leads to an alias: in particular, the `List` variable `queue` becomes an alias of the `List` object returned by the call `this.pendingData.get(socket)` through the assignment. Both pointer and alias analyses are fundamental analysis.

A.1.2 Dependence Analysis. Developers often analyze dependencies among program entities of a software system to help them better understand the behaviors and structure of the system, for various purposes like implementation, testing, debugging, and other maintenance/evolution tasks [58, 176]. Dependence analysis is an analysis that aims to compute these (data and control) dependencies [27, 28]. For a complex program such as a common distributed program, which runs in multiple threads/processes, the users (e.g., developers) need to understand (thus the dependence analysis should computer) both *explicit* (induced by explicit object reference or function invocation, typically within an individual thread/process) [37] and *implicit* (induced by implicit reference/invocation, e.g., via message passing, typically across different threads/processes) [38] dependencies. Data/control dependence analysis is a fundamental form of control/data flow analysis, respectively. Meanwhile, it is also considered an application/client analysis of points-to/alias analysis.

A.2 Distributed Software Systems

Due to increasing requirements for computational scalability and performance, there are more and more real-world software systems designed as *distributed* systems today [55]. Formally, software systems that perform general-purpose distributed computations are broadly defined as distributed systems [55]—since this is the textbook definition, we refer to them as common distributed systems. As opposed to these systems, there also exist *specialized* distributed systems, such as RMI-based systems [191], distributed event-based (DEB) systems [160], cloud systems [218], and Internet of Things (IoT) systems [152], etc. Important for the analysis of distributed programs, distributed systems have some key features [55]: (1) the executions of components (i.e., processes) in a distributed system are concurrent; (2) the components/processes interact/communicate but generally asynchronously in nature; and (3) hardware and software resources can be shared across distributed components/processes. For example, the distributed system NioEcho has a server and one or more clients that each simply sends messages to the server, as shown in Figure 1. NioEcho components/processes (e.g., the server and one client) communicate and coordinate their actions by socket-based message passing through relevant statements (e.g., Lines 7 and 8 in the `NioClient` class).

A.2.1 Architectures. There are mainly three types of architectures of distributed software systems: *peer-to-peer* (P2P), *client-server* (C/S), and *N-tier*. *Peer-to-peer* (P2P) is a popular type of distributed system architecture in which each process/node has equivalent responsibilities and capabilities. P2P differs from C/S in which some processes/nodes are dedicated to serving other processes/nodes [153]. For instance, OpenChord is a typical *peer-to-peer* distributed software, providing network services through distributed hash tables [18].

Client-server (C/S) is a type of network architecture in which each process/node on the network is either a server or a client—the popular browser/server (B/S) architecture can be considered a special instance of C/S (commonly for web applications), where the client is a web browser. Servers are relatively more powerful for controlling and managing relevant resources (e.g., disk drives, printers, network traffic), and clients rely on servers for those resources [153]. For example, NioEcho is a typical *client-server* distributed program, which includes one or more clients and a server [197].

The *n-tier* architecture breaks up an application into tiers, providing flexibility and reusability for developers who only need to modify or add a specific tier (layer), rather than to rewrite the whole application when they decide to change the application. In the term *n-tier*, "n" can be any number (larger than 1) of distinct tiers used in a specific architecture, such as 2-tier, 3-tier, or 4-tier, and so on [153]. For example, Microsoft Azure is a typical *n-tier* distributed system that provides cloud computing services [52, 222].

A.2.2 Lamport Timestamps for Distributed Software. There are many different approaches to managing the timing in distributed systems. One of them maintains a logical clock for all processes with a simple algorithm, called Lamport Timestamps (LTS) [124]. In LTS, each process maintains an integer value, initially zero, which periodically increments, once after every atomic event. The value is attached to the record of the execution of each event (e.g., a message passed from one process to another) as its timestamp [75], centrally or separately. In brief, LTS works as follows: (1) A process increments its counter for each event in it; (2) When sending a message, a process includes its counter value with the message; and (3) On receiving a message, the counter of the recipient is updated (e.g., increased by one). In terms of the communication mechanism, LTS may be implemented to work either asynchronously or synchronously.

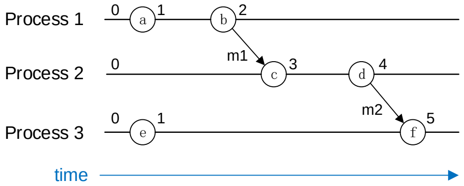

Fig. 6. Lamport logical clocks in a distributed program running in three processes: *Process 1*, *Process 2*, and *Process 3*.

To illustrate LTS, suppose there are three processes concurrently running during the execution of a distributed program, as shown in Figure 6. Each process has its logical clock initialized to zero and the clock value (i.e., timestamp) increments by 1 for each event—e.g., 1 for event *a*, 2 for event *b*, etc. When the message *m1* was sent from *Process 1* to *Process 2*, the timestamp 2 was piggybacked to *m1*. Next, the (message-receiving) event *c* (in *Process 2*) is given a timestamp 3, which is the greater value between the piggybacked timestamp 2 and its local timestamp 0 (initial value) incremented by 1—max(0,2)+1=3. Then, the timestamp of the event *d* is 4 (= 3 + 1). Next, the message *m2* was sent from *Process 2* to *Process 3* with the clock value 4 piggybacked. Finally, the event *f* (in *Process 3*) has its timestamp 5 (= max(1,4) + 1), where 1 is the timestamp of the previous event *e* in the same process (i.e., *Process 3*) [75, 124].

Table 3. Paper Attribution Results (Years of 2022-2024)

Link	Year	Approach															Application					Evaluation			
		Analysis Methodology		Analysis Modality		Program Representation				Algorithmic Parameter		Analysis Data			Analysis Scope		Functional Testing	Fault Localization	Security Support	Performance Diagnosis	Maint./Evol. Support	Dataset		Metric	
		Code -based	Learning -based	Static	Dynamic	CFG/ICFG	CG	PDG/SDG	IFG	Analysis Sensitivity	Data Granularity	Run-time Event	System Log	Artifact	Common System	Specialized System						Benchmark Suite	Real-world System	Effectiven.	Efficiency
[148]	2024	✓			✓						✓	✓			✓		✓		✓				✓	✓	✓
[204]		✓		✓							✓			✓	✓		✓						✓	✓	✓
[72]		✓			✓	✓				✓	✓	✓		✓	✓		✓						✓	✓	
[100]			✓		✓						✓	✓			✓				✓			✓		✓	
[48]			✓		✓						✓	✓	✓			✓		✓			✓		✓	✓	✓
[201]		✓		✓		✓				✓				✓	✓		✓						✓	✓	✓
[134]		✓		✓										✓		✓	✓	✓					✓	✓	✓
[103]			✓	✓									✓	✓	✓			✓			✓		✓	✓	
[122]			✓	✓							✓			✓		✓					✓		✓	✓	✓
[156]	2023	✓			✓	✓				✓		✓			✓		✓		✓				✓	✓	✓
[235]		✓		✓		✓				✓		✓			✓					✓			✓		✓
[71]		✓			✓			✓		✓	✓	✓		✓		✓					✓	✓	✓		
[239]		✓		✓		✓	✓		✓	✓				✓		✓			✓			✓	✓	✓	✓
[216]		✓			✓							✓		✓	✓		✓	✓					✓	✓	
[171]		✓			✓	✓				✓		✓				✓			✓			✓	✓	✓	
[238]		✓		✓			✓	✓		✓				✓		✓					✓	✓		✓	
[182]		✓		✓										✓		✓				✓		✓	✓		✓
[43]		✓		✓			✓	✓						✓		✓					✓	✓			
[117]		✓			✓	✓				✓		✓		✓	✓			✓	✓			✓		✓	
[20]		✓			✓						✓	✓				✓				✓		✓	✓		✓
[118]		✓		✓			✓							✓	✓						✓	✓			✓
[63]			✓		✓						✓		✓		✓				✓				✓	✓	
[109]			✓		✓						✓		✓		✓				✓			✓		✓	✓
[54]		✓			✓							✓	✓			✓	✓						✓	✓	
[230]			✓		✓			✓			✓	✓			✓				✓				✓	✓	
[4]			✓	✓										✓	✓				✓			✓		✓	
[95]		✓			✓							✓			✓		✓						✓	✓	✓
[123]		✓			✓	✓				✓		✓			✓				✓				✓	✓	✓
[57]		✓	✓		✓	✓				✓		✓		✓	✓					✓		✓			✓
[138]			✓		✓						✓	✓		✓	✓						✓		✓	✓	
[139]			✓	✓			✓							✓	✓				✓	✓			✓	✓	
[170]	2022	✓			✓						✓	✓		✓	✓		✓					✓		✓	
[85]		✓	✓		✓			✓	✓	✓	✓	✓		✓	✓						✓	✓	✓	✓	
[78]		✓	✓		✓	✓	✓	✓		✓	✓	✓		✓	✓				✓		✓	✓	✓	✓	✓
[217]		✓			✓	✓	✓			✓		✓			✓				✓			✓	✓		✓
[64]		✓		✓										✓	✓		✓					✓		✓	
[113]		✓		✓										✓	✓		✓					✓		✓	✓
[184]		✓		✓				✓		✓				✓		✓					✓		✓	✓	✓
[68]			✓		✓						✓	✓		✓	✓			✓					✓	✓	
[210]		✓		✓						✓				✓	✓				✓						
[141]		✓		✓		✓				✓				✓		✓	✓					✓	✓	✓	✓
[3]		✓		✓										✓		✓	✓						✓	✓	
[105]		✓		✓		✓				✓				✓		✓	✓					✓		✓	✓
[149]		✓		✓										✓	✓		✓					✓		✓	✓
[42]		✓		✓			✓							✓		✓					✓	✓		✓	
[26]		✓		✓			✓			✓				✓		✓					✓	✓		✓	
[190]		✓		✓			✓							✓		✓					✓	✓		✓	✓
[142]		✓		✓			✓		✓	✓			✓	✓	✓				✓				✓	✓	✓
[5]		✓			✓			✓			✓	✓	✓		✓		✓	✓			✓	✓		✓	
[228]		✓		✓			✓	✓	✓	✓				✓	✓				✓				✓	✓	
[92]		✓		✓										✓	✓		✓					✓	✓	✓	✓

B Mapping of Surveyed Papers to Survey Taxonomy

In section, we provide the detailed paper attribution results, summarizing the mapping of the surveyed papers according to the survey taxonomy we derived.

Table 4. Paper Attribution Results (Years of 2020-2021)

Link	Year	Approach															Application					Evaluation			
		Analysis Methodology		Analysis Modality		Program Representation				Algorithmic Parameter		Analysis Data			Analysis Scope							Dataset		Metric	
		Code -based	Learning -based	Static	Dynamic	CFG/ICFG	CG	PDG/SDG	IFG	Analysis Sensitivity	Data Granularity	Run-time Event	System Log	Artifact	Common System	Specialized System	Functional Testing	Fault Localization	Security Support	Performance Diagnosis	Maint./Evol. Support	Benchmark Suite	Real-world System	Effectiven.	Efficiency
[236]	2021		✓		✓											✓				✓			✓	✓	✓
[97]		✓			✓				✓							✓				✓			✓	✓	
[229]		✓			✓						✓	✓			✓			✓					✓	✓	
[185]		✓			✓						✓	✓				✓				✓			✓	✓	✓
[2]		✓			✓							✓			✓				✓			✓		✓	✓
[45]		✓			✓				✓							✓				✓	✓				
[220]		✓			✓						✓	✓			✓				✓		✓				
[155]		✓			✓				✓			✓				✓		✓		✓					
[30]		✓			✓						✓	✓			✓						✓		✓	✓	✓
[82]		✓			✓	✓				✓	✓	✓		✓	✓			✓	✓				✓		✓
[215]		✓			✓						✓	✓				✓						✓	✓		✓
[13]		✓			✓						✓	✓				✓			✓					✓	
[211]		✓			✓						✓	✓	✓		✓			✓		✓			✓	✓	✓
[70]		✓			✓						✓	✓	✓		✓			✓		✓			✓		✓
[200]		✓		✓		✓	✓									✓		✓	✓				✓		✓
[164]		✓			✓						✓	✓				✓				✓			✓		✓
[163]		✓			✓								✓		✓			✓				✓			✓
[104]		✓			✓						✓		✓		✓		✓			✓		✓		✓	✓
[213]	2020	✓			✓						✓	✓			✓		✓	✓					✓		✓
[154]		✓		✓			✓	✓								✓			✓				✓	✓	✓
[108]			✓		✓											✓				✓	✓	✓		✓	
[62]		✓			✓															✓			✓	✓	✓
[219]		✓		✓			✓			✓					✓			✓	✓			✓		✓	✓
[96]		✓		✓			✓			✓					✓		✓	✓					✓		✓
[234]		✓			✓										✓					✓			✓	✓	✓
[87]		✓		✓												✓				✓			✓	✓	
[145]		✓			✓							✓				✓				✓			✓	✓	✓
[47]		✓			✓									✓	✓						✓		✓	✓	
[143]		✓			✓							✓	✓		✓			✓					✓	✓	✓
[189]		✓			✓							✓				✓				✓		✓	✓	✓	✓
[84]		✓	✓		✓			✓		✓	✓	✓		✓	✓			✓	✓		✓		✓	✓	✓
[83]		✓	✓		✓			✓		✓	✓	✓		✓	✓			✓			✓		✓	✓	✓
[81]		✓			✓	✓				✓	✓	✓		✓	✓			✓	✓		✓		✓		✓
[232]		✓			✓										✓		✓	✓					✓	✓	✓
[24]		✓			✓							✓	✓	✓	✓					✓	✓		✓	✓	✓
[178]		✓			✓							✓	✓		✓		✓					✓		✓	✓
[127]		✓			✓							✓			✓		✓						✓	✓	✓

Table 5. Paper Attribution Results (Years of 2015-2019)

Link	Year	Approach															Application					Evaluation			
		Analysis Methodology		Analysis Modality		Program Representation				Algorithmic Parameter		Analysis Data			Analysis Scope							Dataset		Metric	
		Code -based	Learning -based	Static	Dynamic	CFG/ICFG	CG	PDG/SDG	IFG	Analysis Sensitivity	Data Granularity	Run-time Event	System Log	Artifact	Common System	Specialized System	Functional Testing	Fault Localization	Security Support	Performance Diagnosis	Maint./Evol. Support	Benchmark Suite	Real-world System	Effectiven.	Efficiency
[206]	2019	✓		✓		✓		✓								✓			✓				✓	✓	✓
[79]		✓			✓	✓				✓	✓	✓		✓	✓			✓	✓				✓	✓	✓
[16]		✓		✓											✓					✓		✓		✓	
[11]		✓			✓											✓			✓						
[129]		✓		✓		✓						✓			✓			✓				✓	✓	✓	✓
[22]		✓			✓					✓		✓			✓						✓		✓	✓	
[146]		✓			✓							✓			✓		✓				✓	✓		✓	✓
[93]		✓			✓								✓		✓				✓				✓	✓	
[6]		✓			✓											✓			✓						
[80]		✓			✓	✓				✓	✓	✓			✓				✓		✓		✓		
[135]		✓			✓			✓				✓			✓			✓					✓	✓	✓
[76]		✓			✓	✓				✓		✓		✓	✓			✓	✓				✓		
[188]		✓			✓							✓	✓		✓			✓				✓		✓	✓
[195]		✓			✓			✓				✓							✓				✓	✓	
[199]		✓			✓							✓			✓							✓	✓	✓	✓
[116]		✓			✓											✓			✓				✓		✓
[125]		✓			✓							✓	✓		✓						✓		✓		✓
[77]		✓	✓		✓	✓						✓			✓			✓	✓						
[172]		✓			✓							✓			✓		✓	✓				✓	✓	✓	
[202]	2018	✓			✓							✓		✓	✓			✓					✓	✓	✓
[89]		✓			✓										✓			✓					✓	✓	
[158]					✓							✓			✓				✓						
[147]		✓			✓		✓						✓		✓					✓			✓	✓	✓
[114]		✓			✓							✓				✓		✓	✓			✓			✓
[15]			✓		✓								✓		✓			✓					✓	✓	✓
[162]		✓			✓								✓		✓							✓	✓		✓
[137]		✓			✓							✓			✓		✓						✓	✓	✓
[50]		✓			✓							✓				✓					✓		✓	✓	
[130]		✓			✓			✓				✓			✓								✓	✓	✓
[41]		✓		✓			✓			✓		✓				✓		✓	✓				✓	✓	
[150]		✓		✓											✓								✓	✓	✓
[12]	2017	✓			✓										✓			✓			✓		✓		✓
[136]		✓			✓			✓				✓			✓		✓	✓					✓	✓	
[165]		✓			✓										✓		✓	✓				✓			✓
[90]		✓		✓			✓			✓					✓										✓
[183]		✓			✓							✓			✓					✓			✓		✓
[229]	2016	✓			✓						✓	✓		✓	✓			✓					✓	✓	
[38]		✓			✓						✓	✓		✓	✓						✓		✓	✓	✓
[39]		✓			✓						✓	✓			✓						✓		✓	✓	✓
[151]	2015	✓			✓							✓	✓		✓					✓			✓	✓	✓
[207]		✓		✓		✓		✓								✓			✓				✓	✓	✓

Table 6. Paper Attribution Results (Years of 1995-2014)

Link	Year	Approach															Application					Evaluation			
		Analysis Methodology		Analysis Modality		Program Representation				Algorithmic Parameter		Analysis Data			Analysis Scope		Functional Testing	Fault Localization	Security Support	Performance Diagnosis	Maint./Evol. Support	Dataset		Metric	
		Code -based	Learning -based	Static	Dynamic	CFG/ICFG	CG	PDG/SDG	IFG	Analysis Sensitivity	Data Granularity	Run-time Event	System Log	Artifact	Common System	Specialized System						Benchmark Suite	Real-world System	Effectiven.	Efficiency
[231]	2014	✓			✓							✓				✓		✓					✓	✓	✓
[177]		✓			✓											✓		✓	✓			✓			✓
[212]		✓			✓							✓	✓			✓		✓			✓		✓		✓
[237]		✓			✓	✓	✓						✓		✓								✓	✓	✓
[175]	2013	✓			✓											✓		✓	✓				✓	✓	✓
[233]		✓			✓											✓			✓						
[91]		✓		✓		✓	✓						✓			✓					✓		✓	✓	✓
[99]		✓			✓										✓			✓	✓				✓	✓	
[111]		✓			✓							✓			✓		✓						✓	✓	✓
[174]	2012	✓			✓										✓				✓						
[179]		✓			✓	✓	✓					✓				✓					✓		✓	✓	
[21]		✓			✓											✓			✓		✓				
[19]	2011	✓			✓			✓				✓			✓		✓	✓			✓		✓		✓
[98]		✓			✓							✓			✓	✓					✓	✓		✓	✓
[107]		✓			✓								✓			✓		✓					✓	✓	✓
[193]	2010	✓			✓							✓	✓			✓			✓			✓			✓
[194]		✓			✓							✓			✓					✓			✓	✓	
[227]	2009	✓			✓							✓			✓			✓	✓				✓	✓	✓
[159]	2006	✓			✓			✓				✓			✓								✓		✓
[208]		✓			✓							✓	✓			✓		✓		✓			✓	✓	✓
[119]	2000	✓			✓										✓						✓				
[44]	1996	✓			✓							✓						✓							
[112]	1995	✓			✓			✓				✓		✓	✓			✓			✓				