MirrorTaint: Practical Non-intrusive Dynamic Taint Tracking for JVM-based Microservice Systems

Yicheng Ouyang
University of Illinois
Urbana-Champaign, USA
youyang8@illinois.edu

Kailai Shao
Ant Group
Shanghai, China
kailai.skl@antgroup.com

Kunqiu Chen
Southern University of Science and Technology
Shenzhen, China
11911626@mail.sustech.edu.cn

Ruobing Shen
Peking University
Beijing, China
ruobingshen@pku.edu.cn

Chao Chen, Mingze Xu
Ant Group
Shanghai, China
{chixi.cc,mingze.xmz}@antgroup.com

Yuqun Zhang*
Southern University of Science and Technology
Shenzhen, China
zhangyq@sustech.edu.cn

Lingming Zhang
University of Illinois
Urbana-Champaign, USA
lingming@illinois.edu

Abstract*—Taint analysis, i.e., labeling data and propagating the labels through data flows, has been widely used for analyzing program information flows and ensuring system/data security. Due to its important applications, various taint analysis techniques have been proposed, including static and dynamic taint analysis. However, existing taint analysis techniques can be hardly applied to the rising microservice systems for industrial applications. To address such a problem, in this paper, we proposed the first practical non-intrusive dynamic taint analysis technique *MirrorTaint* for extensively supporting microservice systems on JVMs. In particular, by instrumenting the microservice systems, *MirrorTaint* constructs a set of data structures with their respective policies for labeling/propagating taints in its *mirrored* space. Such data structures are essentially non-intrusive, i.e., modifying no program meta-data or runtime system. Then, during program execution, *MirrorTaint* replicates the stack-based JVM instruction execution in its *mirrored* space on-the-fly for dynamic taint tracking. We have evaluated *MirrorTaint* against state-of-the-art dynamic and static taint analysis systems on various popular open-source microservice systems. The results demonstrate that *MirrorTaint* can achieve better compatibility, quite close precision and higher recall (97.9%/100.0%) than state-of-the-art Phosphor (100.0%/9.9%) and FlowDroid (100%/28.2%). Also, *MirrorTaint* incurs lower runtime overhead than Phosphor (although both are dynamic techniques). Moreover, we have performed a case study in Ant Group, a global billion-user FinTech company, to compare *MirrorTaint* and their mature developer-experience-based data checking system for automatically generated fund documents. The result shows that the developer experience can be incomplete, causing the data checking system to only cover 84.0% total data relations, while *MirrorTaint* can automatically find 99.0% relations with 100.0% precision. Lastly, we also applied *MirrorTaint* to successfully detect a recently wide-spread Log4j2 security vulnerability.

***Index Terms*—dynamic taint analysis, microservice, JVM**

I. INTRODUCTION

Nowadays, the microservice architecture [1], which refers to decomposing software into small independent services that communicate over well-defined APIs (the resulting components are called microservices), has become dominating in industrial applications. In contrast to developing traditional monolithic applications which can be quite inefficient (especially for big code base) [2], the microservice architecture is advanced in enabling a lightweight development paradigm of industrial applications. According to the JetBrains survey in 2021 upon 31,743 developers from 183 countries/regions, 35% of all respondents develop microservices, among whom 88% adopt microservice for their system design, where Java is the most popular programming language (41%) [3]. Big tech companies, e.g., Uber [4], [5], Twitter [6], and Paypal [7], all build their microservices upon Java Virtual Machines (JVMs) [8]. They also contribute many frameworks for developing JVM-based microservices, e.g., Spotify's appollo [9], Netflix OSS [10], and Alibaba's SOFABoot framework [11], which have been widely used by industrial cloud applications.

However, JVM-based microservice systems can be vulnerable to data security threats, which can cause severe damages, e.g., million-dollar or even billion-dollar losses. For example, a vulnerability [12] in widely-used Java logging library Log4j2 [13] reported on November 24, 2021 allows easily executing malicious code from attackers' servers when logging certain strings inputted by users. Such vulnerability posed severe threats to almost all big companies like Google, IBM, Intel, Apple, Microsoft, Tesla, VMware, Zoom, and Cloudflare [14], [15] with millions of attacks per hour globally [16]. Volume developers worldwide have worked intensely to resolve such issues in their respective applications and the caused loss is still beyond estimation till now.

Taint analysis, which has been widely adopted in practice to explore the dependencies between data during program execution [17]–[30], is in nature the solution for such data security threat to Java-based microservice systems above. A typical taint analysis technique firstly marks the source variables with taints (aka. sourcing). Next, it propagates such

*Yuqun Zhang is the corresponding author.

taints between variables through pre-defined propagation rules. Finally, the target variables are inspected for whether they are attached with taints (aka. taint sinking). In general, taint analysis techniques can be categorized into two dimensions: static taint tracking and dynamic taint tracking. In particular, static taint analysis techniques [29]–[31] apply static program analysis to leverage control-flow graphs and call graphs for further taint analysis. On the other hand, dynamic taint analysis techniques [17]–[28] track taints by modifying the runtime systems (e.g., operating systems and virtual machines) or program source/intermediate code. Such techniques, if built precisely, could sensitively track the spread range of program information and facilitate multiple security tasks, e.g., privacy leak detection of operation/back-end systems [18], [28].

Ideally, to solve the aforementioned Log4j2 vulnerability, one can simply adopt the idea of taint analysis – tainting the untrustworthy inputs from outside, and terminating the ongoing execution when identifying Log4j2 attempts to evaluate tainted variables. However, JVM-based microservice architectures can hardly be fully supported by the existing dynamic or static taint analysis techniques. To illustrate, static taint analysis techniques [29]–[31] can incur false positives due to its unawareness of the real-time execution paths. Specifically, such techniques usually collect taints through all the possible paths, resulting in inevitable impreciseness. On the other hand, while dynamic taint analysis techniques can be essentially more precise compared with static taint analysis techniques, they tend to result in poor portability by modifying the specific execution runtime systems for various microservice systems, which is impractical and even unacceptable in industrial production environment where risks cannot be taken to affect stability of runtime systems. Intuitively, such issues can be potentially addressed by modifying the source code to extensively track taints. However, it can be quite impractical since the source code can often be unavailable. Note that while recent effort, i.e., Phosphor [17], attempts to modify program bytecode for enhancing portability and feasibility, it still cannot be applied to extensively support microservice systems because it requires to instrument the Java Development Kit (JDK) and modify the program meta-data (e.g., class fields, method parameters and return types). For example, such meta-data modifications can easily fail the object serialization/de-serialization process widely adopted in microservice communication. Moreover, they can also fail the meta-data inspections/modifications adopted by widely-used dependency injection (DI) [32] and Aspect-Oriented Programming (AOP) [33] techniques for microservice systems, e.g., Spring-based systems [34].

To make dynamic taint analysis practical to microservice systems, our insight is to mirror the application memory space and enable the taint operations in that mirrored memory space to retain program meta-data and make dynamic taint analysis practical on microservice applications. To this end, in this paper, we present the first non-intrusive dynamic taint analysis technique, *MirrorTaint*, which keeps meta-data in JVM classes *intact* during instrumentation and can be widely used in various microservice systems. In particular, *MirrorTaint* contains three components in its mirrored memory space (named *mirrored* space) to dynamically track the taints: TaintHeap, TaintStackFrame, and StackFrameRegister. TaintHeap and TaintStackFrame are two specifically designed data structures for mirroring the heap and stack in JVM to store the taints of variables. Moreover, they also re-implement the associated JVM instructions in terms of taints instead of associated variables at runtime, as their "mirror"s. To realize such execution replication, we also design StackFrameRegister, a temporary taint storage to pass the taints across JVM stack frames. These components enable *MirrorTaint* to track taints without modifying the meta-data of the programs in field- and object-sensitive manners. Additionally, in order to further support microservice scenarios, *MirrorTaint* also supports automatically taint sourcing and sinking for input and output data of various APIs, as well as user-defined taint propagation rules. Noticing that by adopting such an automatic sourcing and sinking mechanism, *MirrorTaint* can track taints across different services in a post-analysis manner, even though the data communicated between services is not tainted. However, it is still limited in 1) not supporting implicit information flow, 2) relying on pre-defined rules to handle native methods and 3) incurring relatively high memory costs.

We evaluate *MirrorTaint* on a set of popular (reflected by star numbers) and available open-source microservice systems on GitHub. The evaluation result presents that *MirrorTaint* can achieve very close precision (97.9%) and superior recall (100.0%) for taint tracking while two compared state-of-the-art techniques Phosphor and FlowDroid either result in limited performance or even fail to be applied. Meanwhile, *MirrorTaint* only incurs an overhead of 32.7% on average. We further conduct a case study in a global billion-user FinTech company Ant Group to apply *MirrorTaint* to their industrial microservice applications, and compare its explored data relations with a mature developer-experience-based data checking system widely used in the company. As a result, while such a system only covers 84.0% relations, *MirrorTaint* achieves 100.0% precision and 99.0% recall in exploring data relations. We also apply *MirrorTaint* to a recent influential Log4j2 security vulnerability and find that *MirrorTaint* precisely detects that vulnerability. To summarize, our paper makes the following contributions:

- To our best knowledge, *MirrorTaint* is the first work to build mirrored memory structures for JVM-based programs to perform dynamic taint analysis, i.e., creating an isolated *mirrored* space for taint storage/propagation and keeping original program meta-data for non-intrusive execution of microservice applications.
- We have implemented *MirrorTaint*, the first practical dynamic taint analysis technique which can be extensively applied to modern JVM-based microservice systems. It leverages extensive bytecode engineering based on ASM [35] and javaagent [36] to mirror the application memory space and operate taints in this *mirrored* space while keeping the

program meta-data intact.

- We evaluate *MirrorTaint* upon real-world open-source microservice benchmarks as well as large-scale industrial microservice applications. All the evaluation results suggest that *MirrorTaint* can substantially outperform state-of-the-art dynamic and static taint analysis techniques in terms of compatibility and recall for tracked taints under quite limited overhead. Moreover, for industrial applications, *MirrorTaint* shows its superiority in exploring industrial sensitive data relations and significantly outperforms a mature developer-experience-based data checking system in Ant Group. Additionally, its ability to detect the recent Log4j2 vulnerability further shows its value in practice.

II. BACKGROUND AND RELATED WORK

In this section, we are going to describe the background about JVM, Java bytecode, and taint analysis.

A. JVM and Java Bytecode

Java Virtual Machine (JVM) [8] executes the programs written by languages that can be compiled into Java bytecode [37].

1) Data Types and Meta-data: There are two general data types in Java bytecode: the primitive and reference types. Specifically, the primitive types include `boolean`, `byte`, `short`, `char`, `int`, `long`, `float` and `double`. Each primitive type corresponds to a wrapper type (essentially reference type), i.e., `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, and `Double`, respectively. On the other hand, the reference types contain the references/addresses of objects, e.g., class, interface, and array.

The meta-data of Java bytecode refers to the information of fields, methods, annotations, inheritance, etc., such as the descriptor/signature of fields/methods and the number of methods and fields. Many techniques are proposed to inspect and modify such bytecode meta-data such as Java reflection [38], ASM [35], javaassit [39], etc. Note that when performing code instrumentation, such bytecode-level meta-data can be changed, causing the execution failure of the original program.

2) JVM Memory and Execution Model: A typical JVM memory structure contains 5 components: method area, heap, JVM stacks, pc registers, and native method stacks. The heap memory and JVM stacks are used to store runtime variables. JVM stacks contain stack frames with local and temporary variables, while the heap stores all objects.

JVM constructs its execution model to execute bytecode instructions. Typically, once a Java method is invoked/terminated, its corresponding JVM stack frame would be pushed/popped to/from the JVM stack. Specifically, a JVM stack frame contains local variable array and operand stack for storing the local/temporary variables. Upon the invocation of a method, its corresponding stack frame is initialized and the head of local variable array is initialized with the arguments passed by the caller. Figure 1 shows an example of the execution model. When adding two local variables a and b of the `int` type and storing the result into the local variable c, firstly a and b should be loaded to the operand stack (instructions `ILOAD 1` and `ILOAD 2`). Next, instruction `IADD` is executed so that a and b are popped from the operand stack and the result of their addition is pushed to the operand stack. Finally, the result is stored into local variable array, where variable c is located (instruction `ISTORE 3`).

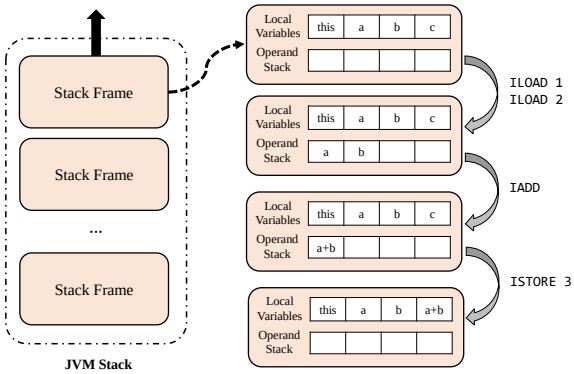


Fig. 1: JVM Execution Model

B. Taint Analysis

1) Static Taint Analysis: Many of the static taint analysis tools target Android applications. Lu et al. [31] studied a vulnerability type called component hijacking vulnerability found in Android apps and proposed a static data flow analysis approach to automatically detect them. Arzt et al. [29] proposed FlowDroid, a novel static taint analysis tool which precisely models Android lifecycle with multidimensional sensitivity to support both Android and generic Java applications. Sridharan et al. [30] proposed F4F, which firstly generates a specification for framework-related behaviors and then performs deeper static taint analysis on the framework-based applications. Livshits et al. [40] built a static taint analyzer based on user-provided specification of vulnerabilities. Although it finds many vulnerabilities in open-source applications and Java libraries, its effectiveness relies on the completeness of the given specification. Targeting vulnerable web applications, Jovanovic et al. [41] proposed Pixy to perform flow-sensitive and context-sensitive static analysis to detect cross-site scripting vulnerabilities in PHP scripts. However, it suffers from around 50% false positive rate.

Although static taint analysis techniques are usually advanced in requiring no program execution and resulting in high recall, they suffer from high false positive rates and large analysis overhead. Moreover, since they are not designed to cope with the techniques leading to undetermined program behaviors, e.g., aspect-oriented programming [33] and dependency injection [32] which are widely used in microservice systems, they can be unfit for microservice systems.

2) Dynamic Taint Analysis: A group of dynamic taint analysis techniques are built upon modifying the hardwares or runtime systems. Suh et al. [27] implemented dynamic taint analysis in the hardware level to achieve low space and performance overhead via a new register and instructions to the processor to track information flows. Enck et al. [18] implemented a system-wide dynamic taint tracking tool TaintDroid

on Android systems by integrating 4 granularities of taint propagation in terms of variables, messages, methods, and files. For the typical operation system threat that multiple threads in a JVM process sharing the same security domain even with illegal information flows, Azadmanesh et al. [42] proposed SEJVM, a modified JVM to dynamically track information flows and associate them with confidentiality polices. Lam et al. [43] proposed GIFT, a compiler for C programs, which is integrated with general dynamic information flow framework and provides interfaces for user-defined tag initialization, propagation, and processing. Although these techniques extend the ability of hardware and runtime systems, their practicability can be limited due to the strong coupling with specific runtime systems. For example, TaintDroid has to make efforts on supporting different Android system versions which can be rather cost-inefficient.

Taint tracking logics have also been implemented by instrumenting executable binaries. Clause et al. [23] proposed a general dynamic taint tracking framework and implemented Dytan prototype tool to track taints in x86 binaries. Kemerlis et al. [26] proposed Libdft to manage tags in a shadow memory and track taints through binary instrumentation. Newsome et al. [44] proposed TaintCheck to detect overwrite attacks, which uses shadow memory with page-table-like structure for each byte of memory to track data operation instructions and taint propagation. Note that these techniques mainly target binary programs and are not applicable to the stack-based memory layout of JVMs since they will have to modify JVMs which is not portable and may bring stability/cost issues in production environment to hinder the execution of microservices.

As for dynamic taint analysis for JVM-based programs, Bell et al. [17] proposed state-of-the-art Phosphor which instruments JDKs and dynamically tracks taints by modifying the bytecode of the executing programs. However, Phosphor suffers from program meta-data pollution (details shown in Section III).

III. MOTIVATION

Many microservice frameworks adopt bytecode inspections/modifications based techniques, e.g., the widely-used Aspect-Oriented Programming (AOP) [33] and Dependency Injection (DI) [32] techniques which use JDK reflection, ASM, and other bytecode operation libraries to construct invocations and dynamically create instances at runtime.

Existing static taint analysis techniques such as FlowDroid [29] hardly support generic microservice systems. For example, when certain fields are left uninitialized for DI at runtime, such techniques cannot figure out the concrete type of the field, missing taints passed to it. Additionally, such techniques are usually insufficiently precise either. For example, static taint analysis tools cannot accurately predict element states inside the container-type data structures such as Array, Map and Collection and thus can hardly find out whether their inclusive elements are tainted.

Existing dynamic taint analysis techniques modifying operating systems [18], [27] and virtual machines [42], [45], [46]

```
1  - public class Example {
2  + public class Example implements TaintedWithObjTag{
3  +     public Taint PHOSPHOR_TAG; // Taint for Example object
4
5      static int val;
6  +     public static Taint valPHOSPHOR_TAG; // Taint for field val
7
8      int doMath(int in) {
9  -         return in + val;
10 +         return this.doMath$$PHOSPHORTAGGED(Taint.emptyTaint(), in,
11 +             Taint.emptyTaint(), new TaintedIntWithObjTag()).val;
12     }
13
14 +     TaintedIntWithObjTag doMath$$PHOSPHORTAGGED(Taint var1, int in,
15 +             Taint in$$PHOSPHORTAGGED, TaintedIntWithObjTag var4) {
16 +         Taint ret$$PHOSPHORTAG =
17 +             Taint.combineTags(in$$PHOSPHORTAGGED, valPHOSPHOR_TAG);
18 +         var4.taint = ret$$PHOSPHORTAG;
19 +         var4.val = in + val;
20 +         return var4;
21 +     }
22 }
```

Fig. 2: Phosphor Instrumentation Example

can also hardly support generic microservice systems under various environments due to their unportability. As source code can be often unavailable in deployment environments, dynamic taint analysis techniques based on modifying the source code [43], [47] can be quite impractical.

Although state-of-the-art Phosphor [17] attempts to modify the bytecode of the applications, it also needs to modify the program meta-data during bytecode instrumentation and can hinder the meta-data inspection techniques widely adopted in microservice systems such as AOP and DI, severely impacting its applicability on microservice systems.

Figure 2 shows an example of Phosphor instrumenting a class. First, the class is implementing a new interface (Line 2). Next, the class instances' taints are stored in an inserted field in the class (Line 3) and the primitive field's taint is stored in the inserted field inside the same class (Line 6). The method is further instrumented into *Phosphored* method (Lines 14-21) whose parameter list is expanded to allow taints to pass through method invocations and the return type is changed to a customized container type to contain both the taint and the returned value. Actually, we have observed many cases where such meta-data modification breaks the functionalities of microservice frameworks when the frameworks check the number of implemented interfaces, the signatures of fields/methods, the elements in stack traces, etc. For example, in the `processRaftService` method of a class [48] in a microservice framework, the number of the interfaces implemented by the application class is asserted to be 1, which causes the Phosphor-instrumented application to crash because Phosphor makes the class implement an extra interface `TaintedWithObjTag`.

Because of such a flaw in its design, Phosphor cannot completely handle the meta-data inspection issues as such issues can be endless regarding the diversity of various microservice architectures and inspection approaches. For example, by investigating its commit logs, we found that Phosphor has been struggling for such issues since 2015 [49]. Figure 3 shows that in the commit "`Bug fixes for recent versions of Spring`" [50], when Spring framework uses an `AnnotationMetadataReadingVisitor`

```
public class HidePhosphorFromASMCV extends ClassVisitor {
  ...
  boolean enabled;
  @Override
  public void visit(int version, int access, String name, String
      signature, String superName, String[] interfaces) {
    super.visit(version, access, name, signature, superName,
      interfaces);
    enabled = name.equals("org/springframework/core/type/classreading
      /AnnotationMetadataReadingVisitor");
  }
  @Override
  public MethodVisitor visitMethod(final int access, final String
      name, final String descriptor, String signature, String[]
      exceptions) {
    if(enabled && name.equals("visitMethod")){
      // Try to instrument AnnotationMetadataReadingVisitor to hide
      generated phosphored methods.
    }
    return mv;
  }
}
```

Fig. 3: Phosphor Meta-data Issue Patch Example

to check the methods' annotations, Phosphor attempts to patch the issues by instrumenting the visitor to hide its instrumented methods from this specific visitor introduced by the new version of Spring. Such issues can be widespread upon any microservice framework, rendering the patching attempts from Phosphor unbounded.

In contrast, our approach, while instrumenting the bytecode to track the taint dynamically, keeps the class meta-data intact in the mean time to support various microservice systems. Instead of modifying the original classes to store and pass the taints, *MirrorTaint* replicates the stack-based JVM execution in its isolated *mirrored* space for non-intrusive dynamic taint tracking.

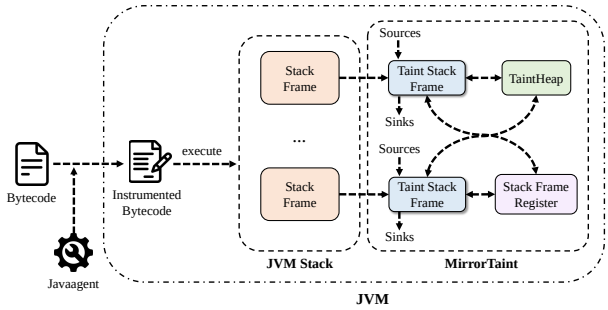


Fig. 4: The Overall Workflow of *MirrorTaint*

IV. APPROACH

MirrorTaint essentially refers to that its taints can follow the JVM operations on variables as their "mirrors". As the existing dynamic taint analysis techniques can inevitably incur substantial modifications on bytecode meta-data, our insight is to store taints in an isolated *mirrored* space and perform propagation by replicating the operations on JVM heap and stack to keep meta-data intact. More specifically, we attempt to construct "mirrored" data structures similar as JVM heap and stack called TaintHeap and TaintStackFrame, i.e., replicate executions just as JVM heap and stack in terms of taints instead of original variables. Additionally, in order to prevent modifying the method parameters/return types to pass the taints of variables, we introduce another data structure called StackFrameRegister to temporarily store the taints for method invocations. Such 3 components together enable *MirrorTaint* to track the taints dynamically in a non-intrusive way.

Figure 4 shows the overall workflow of *MirrorTaint*. Firstly, as a javaagent [36], our tool instruments the bytecode of the original program at runtime. Secondly, it taints target variables for tracking. Next, when invoking methods, TaintStackFrame is initialized along with the JVM stack frame and interacts with TaintHeap and StackFrameRegister to store and propagate the taints. Finally, *MirrorTaint* outputs the sink results when the invocation is terminated.

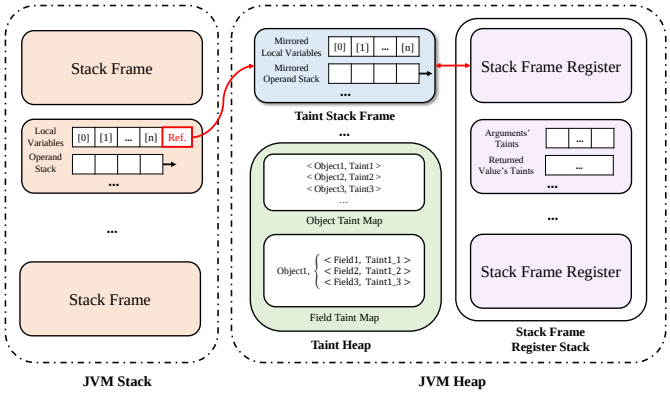


Fig. 5: Data Structures for *MirrorTaint*

A. Taint Storage

1) Data Structures: Figure 5 presents the data structures of TaintHeap, TaintStackFrame and StackFrameRegister. Corresponding to JVM heap, TaintHeap (marked green) is essentially constructed by two global maps—ObjectTaintMap which stores the objects and their corresponding taints as well as FieldTaintMap which stores the field names and the primitive type field taints of objects. Corresponding to the JVM stack, TaintStackFrame (marked blue) contains the similar structure as JVM stack frame. As variables in JVM stack frame are mainly stored in local variable array and operand stack, TaintStackFrame also constructs mirrored local variables and mirrored operand stack to store their taints. In particular, as each JVM stack frame corresponds to a TaintStackFrame object, *MirrorTaint* uses ASM to increase the size of local variable array and stores the reference of TaintStackFrame in the expanded space (the TaintStackFrame object pointed by the reference is stored in heap). Since such expanded space does not override any existing data in local variable array and is only accessed by *MirrorTaint*, it will not affect the original program execution. Moreover, StackFrameRegisters (marked purple) are stored in a global StackFrameRegister stack in the heap. Corresponding to a JVM stack frame, each StackFrameRegister contains a list of taints for method arguments and another taint for method return value.

2) Storage Principles: *MirrorTaint* adopts the following principles in terms of our proposed three data structures:

For TaintStackFrame: At runtime, variable v in the local variable array or operand stack and its taint in mirrored local variables or mirrored operand stack should be identically indexed.

Algorithm 1 Sourcing Variable

```
1: function SOURCE(Variable)
2:     if Variable is a primitive then
3:         Put a taint at the corresponding place in TaintStackFrame
           for Variable;
4:     else if Variable is an terminating type object then
5:         Store the pair of Variable and taint into TaintHeap;
6:     else if Variable is an non-terminating type object then
7:         TAGOBJECT(Variable, 1, ∅);
8:
9: function TAGOBJECT(Variable, depth, searchedList)
10:    if searchedList.contains(Variable) then return
11:    if depth > DEPTH_THRESHOLD then return
12:    searchedList.add(Variable);
13:    if Variable is of terminating type then
14:        Store a new taint along with Variable into TaintHeap;
15:    else if Variable is of Collection type or object Array then
16:        for element in Variable do
17:            TAGOBJECT(element, depth+1, searchedList);
18:    else if Variable is of Map type then
19:        for element in Variable.values() do
20:            TAGOBJECT(element, depth+1, searchedList);
21:    else if Variable is an primitive Array then
22:        for element in Variable do
23:            storePrimitiveFieldToTaintHeap(Variable,index,taint);
24:    else
25:        for field in Variable.fields do
26:            if field is an object then
27:                TAGOBJECT(field, depth+1, searchedList);
28:            else if field is a primitive then
29:                storePrimitiveFieldToTaintHeap(Variable,
                   field.name, taint);
```

For StackFrameRegister: When taints are passed across JVM stack frames, they are first transferred to StackFrameRegister by a TaintStackFrame, and then fetched from the StackFrameRegister by another TaintStackFrame.

For TaintHeap: Before propagating the taint of an object, the taint must be fetched from TaintHeap first; if an object is tainted after taint propagation, the object and taint should be stored in TaintHeap as the key and value.

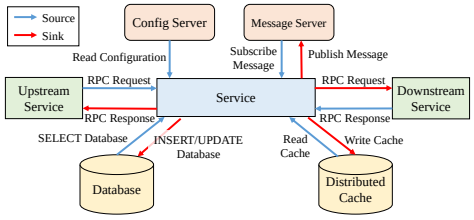


Fig. 6: Source and Sink Types of Microservice Services

B. Source and Sink Automatically for Microservice Systems

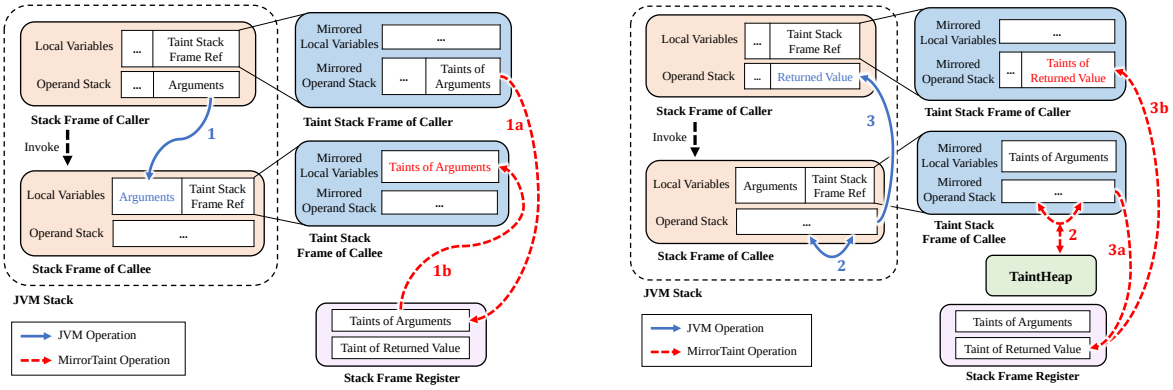
Noticing that manually configuring arbitrary source/sink methods for performing *MirrorTaint* can be impractical since any source/sink configuration change demands re-deploying *MirrorTaint* in production environments which can be rather inconvenient, we propose a mechanism for automatic sourcing and sinking. In particular, we model each service as a black box, i.e., ignoring the details inside each service and only focusing on the data flows in and out of it. Accordingly, our automatic sourcing and sinking mechanism is designed to source all the input data and sink all the output data of a service to deploy *MirrorTaint* once and for all. Figure 6 presents the inputs/outputs which need to be sourced/sinked, marked as blue/red arrows. The input data include the arguments passed by an upstream service, the returned result of a downstream RPC invocation, the data from storage (databases and distributed caches), and the data from config server and message server. On the other hand, the output data include its returned value, the arguments of a downstream RPC invocation, the data written to storage and the message published to the message server.

We further design an algorithm to correctly source different types of variables as presented in Algorithm 1. As "nested" composition can frequently occur in objects (e.g., class A contains a field of class B, class B contains a field of class C, etc), for a fine-grained sourcing process, we recursively source the objects and their inclusive objects under the termination condition that either the variable is already sourced or the recursion depth reaches a threshold (16 by default) or a field of "terminating type", including the primitive types, the wrapper types of primitives, our specified terminating types (i.e., String, StringBuilder, Date, BigDecimal, and Enum), and other user-designated types, is reached. As presented in Algorithm 1, we determine whether the variable is of primitive-type or terminating-type first. If a variable to be sourced is a primitive, its taint will be created in TaintStackFrame (Lines 2-3); if the variable is an terminating-type object, its taint will be created and stored into TaintHeap (Lines 4-5). Otherwise, as the variable is a non-terminating-type object, it will be passed to the recursive method `TAGOBJECT` to source (Line 7). In method `TAGOBJECT`, the terminating conditions are firstly checked (Lines 10-11, 13-14). At the same time, the variable, if has not been sourced before, is added to the list storing the variables already sourced (Line 12). After that, *MirrorTaint* further determines whether the variable is of container-type (i.e., Collection, Map, and Array). If so, its elements are directly stored into the FieldTaintMap of TaintHeap (if they are primitives) or passed to method `TAGOBJECT` recursively (Lines 15-23). Otherwise, each field of the variable is tagged by storing it to the TaintHeap (if it is primitive) or invoking method `TAGOBJECT` (Lines 24-29).

The sinking algorithm is quite similar to Algorithm 1 except that storing the taints to TaintHeap is replaced by fetching the taints from TaintHeap or TaintStackFrame and outputting them to the sink result. After the execution of Microservice APIs, the logs of sink results will be dumped for further processing.

C. Taint Propagation

1) Interprocedural Taint Propagation: Figure 7 presents our approach for interprocedural taint propagation. In Figure 7a, before invoking a method, according to the storage principles, the mirrored operand stack of the caller should contain the taints corresponding to the associated arguments, which are stored in the operand stack of the caller. Then, *MirrorTaint* firstly transfers those taints to the StackFrameRegister (marked as red dashed arrow 1a). To prepare the method



(a) Before Method Invocation. **(b) After Method Invocation.**

Fig. 7: Interprocedural Taint Propagation Workflow of *MirrorTaint*

invocation, JVM will create a JVM stack frame for the callee and transfer the arguments from the operand stack of the caller to the local variable array of the callee (marked as blue solid arrow 1). Accordingly, *MirrorTaint* will transfer the taints of the arguments from the StackFrameRegister to the mirrored local variables of the callee.

Figure 7b demonstrates the process during and after method invocation. When invoking a method, the callee executes the instructions to operate the variables in local variable array and operand stack (marked as blue solid arrow 2). Meanwhile, *MirrorTaint* performs such operations on mirrored local variables and mirrored operand stack as well to align with JVM and propagates taints on mirrored operand stack and mirrored local variables (marked as red dashed arrow 2). Note that *MirrorTaint* will query TaintHeap before object taint propagation and store the resulting taints into TaintHeap after propagation if taints are propagated to an object. Next, before the JVM stack frame of the callee and TaintStackFrame are removed upon the completion of the method invocation, JVM will pass the returned value to the caller's operand stack (marked as blue solid arrow 3). Correspondingly, *MirrorTaint* passes the taint of the returned value to the StackFrameRegister (marked as red dashed arrow 3a). Finally, *MirrorTaint* fetches the taint of the returned value from the StackFrameRegister and stores it to the mirrored operand stack of the caller (marked as red dashed arrow 3b), ensuring the mirrored operand stack can align with the operand stack. Thus, taint tracking components are totally separated from the original programs. Note that since the references of the TaintStackFrames are stored in corresponding JVM stack frames, once the references are removed together with JVM stack frames after the method invocations, the TaintStackFrame will be freed by the JVM garbage collector because there is no reference pointing to it. Also, *MirrorTaint* will remove the corresponding StackFrameRegisters from the StackFrameRegister stack after method invocations.

2) Intraprocedural Taint Propagation: Intraprocedural taint propagation refers to propagating the taints inside a method during its execution. In order to update the TaintHeap and TaintStackFrame at runtime, we overwrite all the taint-related bytecode instructions for their executions upon our proposed "mirrored" data structures. Besides the taint propagation of instructions, *MirrorTaint* also allows the users to define their own rules of taint propagation as "shortcuts" for specific methods. In particular, *MirrorTaint* includes a series of pre-defined propagation rules (as presented in our repository [51]) for commonly-used methods in JDK to avoid JDK instrumentation (which may bring stability issues in production environment) and bypass native methods.

D. Instruction Overwriting

202 Java bytecode instructions can be used in class files since Java SE 8 [52]. *MirrorTaint* modifies 191 of them (i.e., all the taint-related ones) to not only track taints in its TaintHeap and TaintStackFrame, but also update mirrored local variables and mirrored operand stack at runtime to align with the JVM local variable array and operand stack. Specifically, to overwrite an instruction, *MirrorTaint* uses ASM to append the original instruction with additional instruction(s) handling mirrored space. For example, as the instruction `ALOAD` loads a reference-type variable from the local variable array to the operand stack, *MirrorTaint* will append `ALOAD` with additional instructions to load the taint of the target variable from mirrored local variables to mirrored operand stack correspondingly. The whole list of instrumented instructions can be found in our repository [51].

E. Cross-service Taint Tracking

Thanks to the automatic sourcing and sinking mechanism described in Section IV-B, *MirrorTaint* is able to perform cross-service taint tracking in a post-analysis manner. Firstly, *MirrorTaint* performs taint analysis in each service independently. As a result, every time a service is called (i.e., an API in it is invoked), the relations between *ALL* its input and output data during the invocation can be revealed upon the termination of the invocation. Note that each of such input/ouput data is assigned with an identifier, e.g., `[SERVICE]#[CLASS]-#[METHOD]#[ARGUMENT_ID]#[FIELD]`. Then, *MirrorTaint* leverages the microservice tracing service (e.g., Spring Cloud Sleuth [53]) to dynamically track the invocation traces (i.e., route of requests) for cross-service communications. Every time when the taint tracking results of the previous step

are dumped, the information of the current trace is recorded as well. Next, *MirrorTaint* links the taint tracking results of the caller/callee services by the following criteria: (1) their taint tracking results are dumped in the same trace; (2) the sink data identifier in the caller service matches the source data identifier in the callee service. In this way, by matching the output data (sinked) of one service and the input data (sourced) of another service, *MirrorTaint* is able to show how the taint can be propagated across different services in the whole invocation trace.

V. EVALUATION

Our evaluation addresses the following research questions:

- **RQ1** How does *MirrorTaint* perform compared to other taint analysis tools on open-source microservice systems in terms of precision and recall?
- **RQ2** How does *MirrorTaint* perform compared to Phosphor in terms of overhead on open-source microservice systems?
- **RQ3** How does *MirrorTaint* perform on commercial systems?

A. Benchmarks

Table I presents the benchmarks adopted for the evaluation, including 8 open-source projects and 5 industrial projects. In order to collect open-source microservice systems, we collect the benchmarks from GitHub by the following process: Firstly we search the projects with the keyword "microservice", so that the displayed projects are all related to microservice. Next, since *MirrorTaint* aims at the applications executed on JVMs and most of the microservice applications running on JVMs are written in Java, we further filter the displayed results by the Java language. Furthermore, we sort the search results by the star numbers in order to select more influential projects. Then, we manually inspect each result in the first 5 pages to select the projects that are microservice applications instead of tools, plugins, libraries or frameworks. Finally, we clone and execute the projects following the instructions on their documentations. We notice that only 8 applications meet our requirements because 1) microservice systems are intrinsically hard to build due to its complexity and distributed design and 2) outdated third-party components like databases, message brokers, and other services can cause problems in our environment. As a result, we collect these 8 popular open-source microservice applications as our open-source benchmarks.

Additionally, in order to investigate the effectiveness of *MirrorTaint* in industry, we cooperated with a world-leading FinTech company Ant Group, which provides services to more than one billion users. In Ant Group, fund consignment is one important business with more than 90 fund companies as consignors. Specifically, the users purchase funds on the platform provided by Ant Group which, as a broker, generates multiple standardized fund documents with the rigorous format regulated by Ant Group's residential country. Such files contain the transaction data (including customers' trading records and acknowledgments of receipt) for each consignor which are transferred to them through FTP at the end of each day. Subsequently, the transactions take effect for the users by processing these documents. In order to protect the users' financial security, it is vital for Ant Group to ensure the data correctness of programs for generating fund documents since a small error can spread to all the consignors easily.

TABLE I: Subjects

Open-source Microservice System	Description	LoC	Star
ctripcorp/apollo	Configuration management system	55473	27.7k
sqshq/piggymetrics	Financial advisor app	3292	12.1k
zlt2000/microservices-platform	Enterprise-class microservice application	17732	3.9k
microservices-demo/microservices-demo	Online sock shop	3577	3.3k
febsteam/FEBS-Cloud	Permission management system	9924	1.6k
techa03/goodsKill	Online flash sale system	7340	1.4k
macrozheng/mall-swarm	Shopping mall management system	65720	9.8k
GoogleLLP/SuperMarket	Online supermarket app	2399	1.8k
Industry Microservice System	**Description**	**LoC**	**TPM**
finfundtrade	Fund transaction system	252516	2693302.4
finfundtaskcenter	Fund clearing system	295259	200405.7
finfundprotocol	Fund protocol system	84544	5182782.6
finfundmng	Fund back-end management system	65953	182569.3
finvirtualta	Fund virtual transfer agent system	57589	2026439.1

In Ant Group, we deploy *MirrorTaint* on the pre-launch environments of 5 core microservice applications involved in generating fund documents such as transaction, protocol, and clearing to explore the source-sink data relations. Table I presents the statistics about our studied projects, including the star number in GitHub for the open-source projects, the average Transaction Per Minute (TPM) for industrial microservice systems, and their lines of code (LoC). Note that the detailed studied services and APIs of open-source systems are shown later in Table II, while those of industrial systems are omitted in this paper for brevity due to their excessively large amount.

B. Implementation

We implement *MirrorTaint* in Java and utilize ASM [35] and javaagent [36] to instrument JVM bytecode at runtime. With over 20,000 lines of Java code, *MirrorTaint* has been carefully implemented to support both open-source and industrial microservice systems so that the input/output of microservice APIs are sourced/sinked as described in Section IV-B.

C. Experimental Setup

1) Environment: All of our experiments are performed on an Ubuntu Server 20.04 LTS with Intel Xeon CPU E5-4610 and 320 GB memory. While *MirrorTaint* essentially can run on different JVMs, i.e., is not specific to JVM implementations, we adopt Hotspot JDK8u252-b09 as the JVM to perform our evaluations in this paper due to page limit. We execute each evaluation task for 20 times to obtain their average results.

2) Approaches for Comparison: We determine to adopt the state-of-the-art dynamic taint analysis technique Phosphor [17] and static taint analysis technique FlowDroid [29] for performance comparison. Note that although there are other potential options, they are selected because (1) they represent state-of-the-art dynamic and static taint analysis, and (2) their source code is publicly available for successful execution.

3) Metrics: Following prior work [17], [29], we adopt the widely-used metrics for evaluating our studied approaches: precision, recall, and time overhead. Since the output of taint analysis is data relations (aka. source-sink pairs [54]–[56], which are used to check data correctness, track suspicious outer input, prevent data leaks, etc), we use the reported

data relations to calculate precision and recall. Specifically, precision refers to the fraction of correctly found data relations among the all the relations found, while recall refers to the fraction of correctly found data relations among all the ground-truth relations. Time overhead refers to the extra execution time when applying taint analysis tools to the benchmarks.

D. Result Analysis

1) RQ1: Precision and Recall on Open-source Benchmarks: As stated in Section IV-B, the sink sites (where the data is likely to be exposed to external environments) are tightly associated with communications between microservice system components. Since such communications can be time-consuming [57], [58], we determine to use the 5 most time-consuming APIs of each open-source project as the benchmarks. For the API having its corresponding test(s), we directly use such tests. Otherwise we write a test to invoke the API to simulate user operations. Specifically, for a test case T, the ground truth is defined by the pairs of the tainted variable at the sink site and the corresponding sourced variable which propagates the taint to it on the T's execution path. Therefore, we collected the test execution code coverage and manually analyzed the data-flow along the exact test execution path to find such ground truth relations and compare it with the analysis results of *MirrorTaint*, Phosphor and FlowDroid. Their precision and recall results are shown in the columns 4-7 in Table II where "**F**" and "**M**" respectively denote found and missed data relations. Additionally, the numbers of false positives are denoted with parentheses in the column "**F**".

We can observe from the results that *MirrorTaint* has found all the relations while incurring 9 false positives. We find such false positives are caused by "over-taint" which means some variables are unnecessarily tainted when a variable is correctly tainted. For example, as the "String" class is immutable in Java, when tainting a constant String variable, other variables sharing the same constant String value will be tainted as well. Moreover, because of the Java Integer caching mechanism [59], where the Integer objects from -128 to 127 are cached internally and reused when creating a new Integer, multiple Integer variables can be tainted together when they share the same Integer objects. As a result, *MirrorTaint* achieves 97.9% precision and 100.0% recall.

Note that FlowDroid and Phosphor actually report the results of analysis at a coarser-grained statement granularity, while *MirrorTaint*'s variable-granularity reports present not only sink/source statements but also the specific values and variables that are tainted at the sink statement. For example, consider an object O sourced at source statement S_{sr} and its taint found at sink statement S_{sk}. While FlowDroid and Phosphor report S_{sr} and S_{sk}, *MirrorTaint* also reports the presence of taint in the fields (including recursive ones) of O at S_{sk}. For an illustration of the comparison of the results of these tools, refer to an example provided in our repository [51]. Therefore, *MirrorTaint* and FlowDroid/Phosphor derive different ground-truth results and we have to define the "M" column differently ("$\underline{\text{M}}$" in table) for FlowDroid and Phosphor, i.e., missed reports for sink-source statements instead of missed variable relations. As the result shows, FlowDroid produces empty results for most of the APIs. We find that FlowDroid fails to support taint tracking in asynchronized invocations and polymorphism scenarios. Notably, since the sink-source statements found by FlowDroid are quite simple, FlowDroid does not incur any false positives. Finally, while FlowDroid results in 100% precision, it only enables 28.2% recall and misses 51 records. On the other hand, Phosphor fails to execute 7 out of the 8 benchmarks. Their failure messages all imply meta-data-related issues which can hardly be fixed once and for all. As for the benchmark which can be executed with Phosphor, Phosphor finds all the sink-source statements, leading to a precision of 100.0% and a recall of 9.9%.

In order to investigate the contributions made by the components of *MirrorTaint*, we implement a variant of *MirrorTaint* marked as *MirrorTaint*$_{TH}$ by disabling the TaintStackFrame (*MirrorTaint* cannot work without TaintHeap). As *MirrorTaint*$_{TH}$ cannot store the taints of primitive types without TaintStackFrame, it loses the relations when primitive wrapper types are cast into primitive types. However, such cases are not common. As shown in Table II, *MirrorTaint* without TaintStackFrame can still achieve 98.1% precision and 97.9% recall. Note that it enables higher precision than *MirrorTaint* because it misses some data relations that are false positives in *MirrorTaint*'s results. Therefore, we can infer that *MirrorTaint*$_{TH}$ is capable of exploring most relations.

In a nutshell, *MirrorTaint* is close in precision and superior in recall compared to state-of-the-art Phosphor and FlowDroid.

2) RQ2: Overhead on Open-source Benchmarks: For the overhead of *MirrorTaint*/Phosphor (static techniques like FlowDroid do not dynamically execute the application, thus incomparable), we collect the time/memory costs of the original API executions, followed by the time/memory costs of the same process while applying *MirrorTaint*'s and Phosphor's javaagents. As presented in the last two columns of Table II, where $\mathbf{T}_O$/$\mathbf{M}_O$, $\mathbf{T}_{MirrorTaint}$/$\mathbf{M}_{MirrorTaint}$ and $\mathbf{T}_{Phos}$/$\mathbf{M}_{Phos}$ refer to the runtime/memory costs of the original execution, *MirrorTaint*, and Phosphor respectively. The numbers in parentheses indicate the overhead percentage. Specifically, for the only microservice application (i.e., SockShop) Phosphor can run on, *MirrorTaint* incurs 83.4% runtime overhead and 167.6% memory overhead on average, while Phosphor incurs 138.6% and 40.36%. Overall, *MirrorTaint* achieves an average 32.7% runtime overhead and 127.9% memory overhead on all studied APIs. Note that such memory overhead is normally acceptable following prior work (e.g., DYTAN [23]).

Interestingly, although high memory overhead cases (e.g., greater than 500%) are observed from the memory overhead results, such cases are mainly distributed in the benchmarks with low/moderate memory usage (less than 1000kB) as shown in Figure 8. On the benchmarks consuming more than 1000 kB, *MirrorTaint*'s memory overhead is much more stable, i.e., having a relatively low percentage (around 100%), which implies the scalability of *MirrorTaint*.

3) RQ3: MirrorTaint in Industry:

TABLE II: Experimental results for *MirrorTaint*, *MirrorTaint*$_{TH}$, FlowDroid, and Phosphor

Benchmark	Service	MethodName	*MirrorTaint*		*MirrorTaint*$_{TH}$		FlowDroid		Phosphor		Runtime Cost (ms)			Memory Cost (kB)		
			F	M	F	M	F	M	F	M	T_O	$T_{MirrorTaint}$	T_{Phos}	M_O	$M_{MirrorTaint}$	M_{Phos}
apollo	apollo	findNamespace	30	0	28	2	2	0	✗		56.9	75.1 (32.0%)	N/A	363.2	3782.5 (941.5%)	N/A
		findActiveReleases	14	0	14	0	0	1			32.3	39.7 (22.9%)		1182.1	1990.9 (68.4%)	
		getInstancesCountByNamespace	13	0	13	0	0	1			27.1	32.1 (18.5%)		173.0	224.2 (29.6%)	
		findBranch	3	0	3	0	0	1			23.4	30.3 (29.5%)		822.3	2119.7 (157.8%)	
		getByRelease	12	0	12	0	0	2			20.6	22.0 (6.8%)		81.9	227.7 (178.0%)	
piggymetrics	account-service	saveCurrentAccount	18	0	17	1	2	0	✗		45.6	55.5 (21.7%)	N/A	274.2	2028.6 (639.8%)	N/A
		createNewAccount	2	0	2	0	1	0			97.6	104.2 (6.8%)		308.5	1008.4 (226.9%)	
	auth-service	createUser	1	0	1	0	0	1			81.6	86.2 (5.6%)		152.1	415.0 (172.9%)	
		getUser	10	0	10	0	1	0			4.0	7.4 (85.0%)		9.7	19.1 (97.1%)	
	notification-service	saveCurrentNotificationsSettings	11 (1)	0	11 (1)	0	2	0			2.7	5.9 (118.5%)		210.9	640.7 (203.8%)	
ZLT-MP	UaaServer	list	2	0	2	0	0	1	✗		14.8	29.7 (100.7%)	N/A	422.5	936.6 (121.7%)	N/A
	UserCenter	saveOrUpdate	17 (1)	0	17 (1)	0	0	1			5.1	7.6 (49.0%)		2587.8	3205.6 (23.9%)	
		findByMobile	17	0	17	0	0	1			20.1	23.3 (15.9%)		513.4	822.9 (60.3%)	
		findUsers	17	0	17	0	0	1			19.7	32.6 (65.5%)		717.5	1818.7 (153.5%)	
		getLoginAppUser	25 (1)	0	25 (1)	0	0	1			18.9	24.2 (28.0%)		689.7	1004.8 (45.7%)	
SockShop	Orders	newOrder	47	0	46	1	0	2	2	0	92.6	140.3 (51.5%)	209.0 (125.7%)	1705.6	3521.5 (106.5%)	2108.8 (23.6%)
	Carts	get	7	0	7	0	0	1	1	0	3.7	8.0 (116.2%)	8.6 (132.4%)	188.2	989.7 (425.9%)	419.5 (122.9%)
		delete	6	0	6	0	0	1	1	0	6.0	14.9 (148.3%)	16.1 (168.3%)	100.7	816.2 (710.4%)	317.6 (215.3%)
		mergeCarts	5	0	5	0	0	1	1	0	15.2	37.8 (148.7%)	41.7 (174.3%)	287.9	1704.0 (491.9%)	504.7 (75.3%)
	Shipping	postShipping	4	0	4	0	2	0	2	0	8.5	30.1 (254.1%)	25.2 (196.5%)	590.4	655.6 (11.0%)	681.8 (15.5%)
FEBS-Cloud	auth	addOauthCliendetails	7	0	7	0	0	1	✗		76.6	102.7 (34.1%)	N/A	2947.6	49116.4 (1566.3%)	N/A
	server-job	jobList	9 (1)	0	9 (1)	0	0	1			4.8	8.4 (75.0%)		865.7	2612.2 (201.7%)	
	server-system	userList	16	0	16	0	0	1			7.7	13.1 (70.1%)		980.3	5145.9 (424.9%)	
		index	11 (3)	0	10 (2)	1	0	4			13.2	22.4 (69.7%)		2499.4	7000.6 (180.1%)	
		addUser	9	0	9	0	0	1			74.9	81.8 (9.2%)		2993.8	5460.0 (82.4%)	
goodskill	goodsservice	executeSeckill	7	0	7	0	0	3	✗		62.2	111.6 (79.4%)	N/A	82714.4	121789.9 (47.2%)	N/A
	goodsweb	doWithSychronized	4	0	1	3	1	3			330.6	385.1 (16.5%)		839.4	2043.2 (143.4%)	
		getDirectoryPermissionList	10 (2)	0	10 (2)	0	0	4			123.2	214.5 (74.1%)		11470.1	14156.8 (23.4%)	
		execute	10	0	10	0	2	2			78.5	84.8 (8.0%)		1122.1	1624.0 (44.7%)	
		roleLess	11	0	11	0	0	3			7.9	11.8 (49.4%)		711.4	1356.9 (90.7%)	
mall-swarm	portal	login	8	0	8	0	1	1	✗		126.0	130.6 (3.7%)	N/A	4391.1	8723.0 (98.7%)	N/A
	admn	register	13	0	13	0	0	2			60.6	75.6 (24.8%)		1315.0	4771.6 (262.9%)	
		login	10	0	10	0	1	2			132.9	162.9 (22.6%)		594.5	12181.9 (1949.2%)	
		updatePassword	8	0	8	0	0	1			61.8	83.6 (35.3%)		975.2	7175.1 (635.8%)	
		update	7	0	7	0	1	0			112.6	133.6 (18.7%)		3324.9	6477.4 (94.8%)	
SuperMarket	cart	updateCart	4	0	4	0	1	1	✗		18.3	26.3 (43.7%)	N/A	488.2	4542.1 (830.4%)	N/A
		addCart	7	0	7	0	2	1			22.8	29.5 (29.4%)		527.5	3832.5 (626.6%)	
	order	addOrder	11	0	11	0	0	2			96.4	173.5 (80.0%)		512.1	6283.3 (1127.0%)	
	product	pageManage	9	0	8	1	0	2			9.0	17.2 (91.1%)		1024.3	7195.7 (602.5%)	
	user	registUser	4	0	4	0	1	0			12.3	16.0 (30.1%)		2039.2	5331.3 (161.4%)	
Sum / Average			436 (9)	0	427 (8)	9	20	51	7	0	50.7	67.3 (32.7%)	N/A	3342.9	7618.8 (127.9%)	N/A

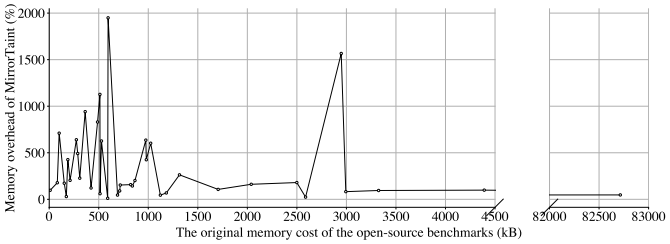


Fig. 8: *MirrorTaint*'s Memory Overhead on Open-source Benchmarks in terms of Memory Usage of APIs

TABLE III: *MirrorTaint* and the Data Checking System on Covering Data Relations for 01/03 Fund Documents

Fund Document	# Fields	# Actual Relations	MirrorTaint			Developer-experience-based Data Checking System		
			# Found Relations	# Correct Relations	# Missed Relations	# Covered Relations	# Missed Relations	Relation Coverage
01 Fund Doc A	17	22	22	22	0	16	6	72.7%
01 Fund Doc B	22	28	28	28	0	23	5	82.1%
01 Fund Doc C 01 Fund Doc D 01 Fund Doc E 01 Fund Doc F	17	23	23	23	0	17	6	73.9%
03 Fund Doc A	27	59	57	57	2	52	7	88.1%
03 Fund Doc B	13	17	17	17	0	15	2	88.2%
03 Fund Doc C 03 Fund Doc D 03 Fund Doc E 03 Fund Doc F	28	45	45	45	0	40	5	88.9%
Sum	124	194	192	192	2	163	31	84.0%

Case Study in Ant Group. To investigate the applicability and effectiveness of *MirrorTaint* in industry, we apply *MirrorTaint* to perform cross-service taint tracking to the microservice systems for generating the fund documents in Ant Group where `01 fund documents` and `03 fund documents` are most hazardous as they are responsible for account registration/closure and fund transactions (e.g., fund purchase and redemption) respectively.

Ant Group has built a mature developer-experience-based data checking system with a collection of checking rules summarized by multiple teams. Such data checking system is executed to check data correctness in the fund documents before delivering the documents to fund companies. However, such an approach is susceptible to missing potential checking rules and even incorporates wrong rules due to incomplete and unreliable human experience. For instance, taint analysis can reveal the relationship between the actual amount of paid money A, recorded in fund documents, and the product price B stored in a database, and the discount C stored in another database. A should equal B minus C, but developers/experts may not know this relationship and miss the correctness check. In order to investigate the effectiveness of the data checking system and *MirrorTaint*, we perform a case study on 12 `01/03 fund documents` of 6 business tasks (such as standard funds and exchange-traded funds) which are generated by adopting the 5 systems shown in Table I.

Table III shows the study results. Because the `fund documents` of C, D, E, F are similar (i.e., the classes generating them share the same super class), we put them together in one table cell. In order to obtain the complete number of data relations (shown in the "# Actual Relations" column), we invited 3 experienced developers to check them carefully. As shown in the table, *MirrorTaint* almost explores all the ground-truth data relations, achieving 99.0% recall and 100.0% precision, while the developer-experience-based data checking system misses 31 relations (which are all explored by *MirrorTaint*) and only achieves 84.0% relation coverage. The 2 data relations missed by *MirrorTaint* are the constant source tracking cases (`new BigDecimal(0)` and `new Money(0)`), which are

not included in the sourcing scope of *MirrorTaint*.

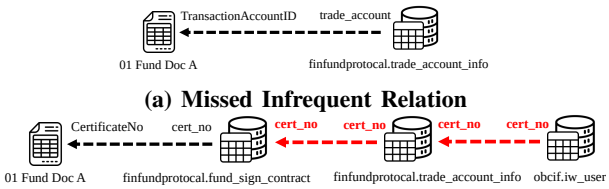
We found the data relations missed by the data checking system can be divided into two categories: infrequent relation omissions and source relation omissions. Figure 9a shows an infrequent relation omission. It shows that the data in the `01 Fund Document A` of field `TransactionAccountID` is passed from field `trade_account` of table `finfundprotocol.trade_account_info`. Such a relation only appears when there exists account closure application records in the `01 Fund Document A`, which is infrequent and can be easily missed by human experience. Figure 9b shows two source relations missed by the data checking system. Although the data checking system has explored the relation from field `cert_no` in table `finfundprotocal.fund_sign_contract` to field `CertificateNo` in `01 Fund Document A`, it still misses the other two upstream relations which are explored by *MirrorTaint* (highlighted with red). All such results show that *MirrorTaint* covers much more data relations than developer experience, and is more reliable in ensuring data correctness.

Case Study on Log4j2 Vulnerability. At the end of 2021, a reported vulnerability (CVE-2021-44228 [12]) in a widely-used Java logging library Log4j2 [13] caused global panic and was described as the most serious vulnerability in decades by mass media [60]–[62]. Specifically, as a logging library, Log4j2 supports a feature called "lookup" to evaluate variables or expressions embedded in logging text, e.g., "`${date:MM-dd-yyyy}`" can be logged as runtime date by Log4j2. However, among many different kinds of lookup, the vulnerability allows JNDI (Java Naming and Directory Interface) lookup (e.g., "`${jndi:ldap://xxx.xxx/xxx}`") to download and execute malicious code from attackers' servers.

In this paper, we also reproduce the vulnerability to investigate the potential of *MirrorTaint* in detecting such attacks. Specifically, we reproduce the attack scenario in an API of an earlier version of a microservice-based application in Ant Group which suffers from this vulnerability as shown in Figure 10 (sensitive information is hidden for security reasons). Dangers can occur when requesting this API with malicious input such as `${jndi:ldap://xxx.xxx/xxx}`. Since the outer inputs are untrustworthy, we source the arguments of the API which receives input data from users and sink the argument of the Log4j2 sensitive `lookup` method. After executing the test case, *MirrorTaint* has identified the lookup to be unsafe as taint is found in the argument of `lookup` method. Its complete output log can be found in our repository [51]. Additionally, we also try to apply FlowDroid and Phosphor on this API. While Phosphor still fails on execution, FlowDroid reports no taint as it fails to track the taint under polymorphism scenarios (invoking interface methods).

VI. THREATS TO VALIDITY

The threats to external validity mainly lie in the limited set of studied open-source microservice projects and the generalizability of the approach. Therefore, we also performed a case study applying *MirrorTaint* to 5 microservice systems



(a) Missed Infrequent Relation

(b) Missed Source Relations

Fig. 9: Missed Relations by Existing Data Checking System

```
1 @GetMapping("/")
2 public String index(@RequestHeader("Api-Version") String version) {
3     ...
4     logger.info("Received a request for version " + version);
5     ...
6 }
```

Fig. 10: One Microservice API Triggering Log4j2 CVE

in the 1-billion-user Ant Group as benchmarks. As for the generalizability, it is worth noting that *MirrorTaint* can be easily extended to any JVM-based program. For such a purpose, one only needs to modify the automatic sourcing and sinking mechanism for different microservice inputs/outputs while retainning other *MirrorTaint* components. The threats to internal validity mainly lie in the approach implementation and ground-truth sink result production. Thus, we collaborated with experienced industrial engineers to develop our tool to ensure the implementation correctness. We also standardized the ground-truth sink result production procedure and had 4 authors analyze it individually and discuss the differences until reaching a consensus. The threats to construct validity mainly lie in the metrics used. Thus, following prior work [17], [29], we adopt widely-used metrics – precision/recall/overhead for evaluating our studied approaches.

VII. CONCLUSION

In this paper, we propose a practical non-intrusive dynamic taint tracking tool named *MirrorTaint* which automatically tracks the taints of the input and output data in microservice systems via a *mirrored* JVM space, and successfully avoids the meta-data modification. We compare its precision and recall on open-source benchmarks to state-of-the-art Phosphor and FlowDroid. The result shows that *MirrorTaint* is more compatible with microservice systems and achieves quite close precision and much higher recall than Phosphor and FlowDroid with only 32.7% average runtime overhead. Also, we apply *MirrorTaint* to 5 important microservice systems in the world-leading FinTech company Ant Group where the result shows that *MirrorTaint* can find 99.0% data relations with 100.0% precision. Additionally, the fact that *MirrorTaint* detects the severe Log4j2 vulnerability indicates its capability of detecting real-world vulnerabilities.

ACKNOWLEDGMENTS

This work is partially supported by the National Natural Science Foundation of China (Grant No. 61902169), Guangdong Provincial Key Laboratory (Grant No. 2020B121201001), as well as NSF grants CCF-2131943 and CCF-2141474. We also acknowledge support from Ant Group.

REFERENCES

[1] "Microservices," https://en.wikipedia.org/wiki/Microservices, 2022.

[2] "Microservices vs monolith: The ultimate comparison 2021." https://www.clickittech.com/devops/microservices-vs-monolith/, 2022.

[3] "The state of developer ecosystem 2021 - microservices." https://www.jetbrains.com/lp/devecosystem-2021/microservices/, 2022.

[4] "The architecture of uber's api gateway." https://eng.uber.com/architecture-api-gateway/, 2022.

[5] "The uber engineering tech stack, part i: The foundation." https://eng.uber.com/tech-stack-part-one-foundation/, 2022.

[6] "How airbnb and twitter cut back on microservice complexities." https://thenewstack.io/how-airbnb-and-twitter-cut-back-on-microservice-complexities/, 2022.

[7] "Spring boot @ paypal." https://www.infoq.com/presentations/paypal-spring-boot/, 2022.

[8] "Java virtual machine," https://en.wikipedia.org/wiki/Java_virtual_machine, 2022.

[9] "Apollo." https://github.com/spotify/apollo, 2022.

[10] "Netflix open source software center." https://netflix.github.io/, 2022.

[11] "Sofaboot." https://github.com/sofastack/sofa-boot, 2022.

[12] "Cve-2021-44228." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44228, 2022.

[13] "Apache log4j 2." https://logging.apache.org/log4j/2.x/, 2022.

[14] "The log4j bug exposes a bigger issue: Open-source funding." https://thenextweb.com/news/log4j-bug-internet-open-source-contributors-analysis, 2022.

[15] "Cisa log4j (cve-2021-44228) affected vendor software list." https://github.com/cisagov/log4j-affected-db/blob/develop/SOFTWARE-LIST.md, 2022.

[16] "Akamai recommendations for log4j mitigation." https://www.akamai.com/blog/security/akamai-recommendations-for-log4j-mitigation, 2022.

[17] J. Bell and G. Kaiser, "Phosphor: Illuminating dynamic data flow in off-the shelf jvms," in *Proceeding of the 29th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA 2014, 2014, acceptance rate: 28expectations. [Online]. Available: https://jonbell.net/publications/phosphor

[18] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2619091

[19] A. Reina, A. Fattori, and L. Cavallaro, "A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors," in *ACM European Workshop on Systems Security (EuroSec)*. ACM, Apr. 2013.

[20] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. USA: USENIX Association, 2012, p. 29.

[21] W. Chang, B. Streiff, and C. Lin, "Efficient and extensible security enforcement using dynamic data flow analysis," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 39–50. [Online]. Available: https://doi.org/10.1145/1455770.1455778

[22] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "Tainttrace: Efficient flow tracing with dynamic binary rewriting," in *11th IEEE Symposium on Computers and Communications (ISCC'06)*, 2006, pp. 749–754.

[23] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 196–206. [Online]. Available: https://doi.org/10.1145/1273463.1273490

[24] M. Ganai, D. Lee, and A. Gupta, "Dtam: Dynamic taint analysis of multi-threaded programs for relevancy," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2393596.2393650

[25] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *21st Annual Computer Security Applications Conference (ACSAC'05)*, 2005, pp. 9 pp.–311.

[26] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "Libdft: Practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, ser. VEE '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 121–132. [Online]. Available: https://doi.org/10.1145/2151024.2151042

[27] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XI. Association for Computing Machinery, 2004, p. 85–96.

[28] M. Azadmanesh and M. Sharifi, "Towards a system-wide and transparent security mechanism using language-level information flow control," 01 2010, pp. 19–26.

[29] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–269. [Online]. Available: https://doi.org/10.1145/2594291.2594299

[30] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg, "F4f: Taint analysis of framework-based web applications," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 1053–1068. [Online]. Available: https://doi.org/10.1145/2048066.2048145

[31] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 229–240. [Online]. Available: https://doi.org/10.1145/2382196.2382223

[32] "Dependency injection," https://en.wikipedia.org/wiki/Dependency_injection, 2022.

[33] "Aspect-oriented programming," https://en.wikipedia.org/wiki/Aspect-oriented_programming, 2022.

[34] "Spring," https://spring.io/, 2022.

[35] "Asm," https://asm.ow2.io/, 2022.

[36] "Package java.lang.instrument." https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html, 2022.

[37] "Java bytecode," https://en.wikipedia.org/wiki/Java_bytecode, 2022.

[38] "Java reflection," https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/index.html, 2022.

[39] "Javassist," https://www.javassist.org/, 2022.

[40] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, ser. SSYM'05. USA: USENIX Association, 2005, p. 18.

[41] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (SP'06)*, 2006, pp. 6 pp.–263.

[42] M. R. Azadmanesh and M. Sharifi, "Towards a system-wide and transparent security mechanism using language-level information flow control," in *Proceedings of the 3rd International Conference on Security of Information and Networks*, ser. SIN '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 19–26. [Online]. Available: https://doi.org/10.1145/1854099.1854107

[43] L. C. Lam and T.-c. Chiueh, "A general dynamic information flow tracking framework for security applications," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, 2006, pp. 463–472.

[44] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software." in *NDSS*, vol. 5. Citeseer, 2005, pp. 3–4.

[45] D. Chandra and M. Franz, "Fine-grained information flow analysis and enforcement in a java virtual machine," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 2007, pp. 463–475.

[46] S. K. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum, "A virtual machine based information flow control system for policy enforcement," *Electron. Notes Theor. Comput. Sci.*, vol. 197, no. 1, p. 3–16, Feb. 2008. [Online]. Available: https://doi.org/10.1016/j.entcs.2007.10.010

[47] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks," in *15th USENIX Security Symposium (USENIX Security 06)*. Vancouver, B.C. Canada: USENIX Association, Jul. 2006. [Online]. Available: https://www.usenix.org/conference/15th-usenix-security-symposium/taint-enhanced-policy-enforcement-practical-approach

[48] "RaftAnnotationBeanPostProcessor.java." https://www.programcreek.com/java-api-examples/?code=sofastack%2Fsofa-registry%2Fsofa-registry-master%2Fserver%2Fserver%2Fmeta%2Fsrc%2Fmain%2Fjava%2Fcom%2Falipay%2Fsofa%2Fregistry%2Fserver%2Fmeta%2Frepository%2Fannotation%2FRaftAnnotationBeanPostProcessor.java, 2022.

[49] "Reflection fixes," https://github.com/gmu-swe/phosphor/commit/081ff88f884497621eefd9e8016b9256e99e6a3f, 2022.

[50] "Bug fixes for recent versions of spring," https://github.com/gmu-swe/phosphor/commit/ca362a4a978778884a478f44b2479d5f302c00fd, 2022.

[51] "MirrorTaint repository." https://github.com/MirrorTaint/MirrorTaint, 2022.

[52] "The java virtual machine instruction set." https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html, 2022.

[53] "Spring cloud sleuth." https://spring.io/projects/spring-cloud-sleuth, 2022.

[54] L. Luo, E. Bodden, and J. Späth, "A qualitative analysis of android taint-analysis results," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 102–114.

[55] X. Fu and H. Cai, "A dynamic taint analyzer for distributed systems," ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 1115–1119. [Online]. Available: https://doi.org/10.1145/3338906.3341179

[56] S. Wei and B. G. Ryder, "Practical blended taint analysis for javascript," ser. ISSTA 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 336–346. [Online]. Available: https://doi.org/10.1145/2483760.2483788

[57] "Overcoming io overhead in micro-services." https://kislayverma.com/programming/overcoming-io-overhead-in-micro-services/, 2022.

[58] "Microservice trade-offs." https://martinfowler.com/articles/microservice-trade-offs.html, 2022.

[59] "Java integer cache." https://javapapers.com/java/java-integer-cache/, 2022.

[60] "Why is the log4j cybersecurity flaw the 'most serious' in decades?" https://nypost.com/2021/12/20/why-is-the-log4j-cybersecurity-flaw-the-most-serious-in-decades/, 2022.

[61] "Log4j could be the most serious security threat ever seen, cisa head warns." https://www.techradar.com/news/log4j-could-be-the-most-serious-threat-ever-seen-cisa-head-warns, 2022.

[62] "Log4j vulnerability causes global panic." https://www.israeldefense.co.il/en/node/52976, 2022.