ZTD_{JAVA}: Mitigating Software Supply Chain Vulnerabilities via Zero-Trust Dependencies

Paschal C. Amusuo[†]
Purdue University

Kyle A. Robinson
Purdue University

Tanmay Singla
Purdue University

Huiyun Peng
Mount Holyoke College

Aravind Machiry
Purdue University

Santiago Torres-Arias
Purdue University

Laurent Simon
Google

James C. Davis
Purdue University

***Abstract*—Third-party libraries like Log4j accelerate software application development but introduce substantial risk. Vulnerabilities in these libraries have led to Software Supply Chain (SSC) attacks that compromised resources within the host system. These attacks benefit from current application permissions approaches: third-party libraries are implicitly trusted in the application runtime. An application runtime designed with Zero-Trust Architecture (ZTA) principles — secure access to resources, continuous monitoring, and least-privilege enforcement — could mitigate SSC attacks, as it would give zero implicit trust to these libraries. However, no individual security defense incorporates these principles at a low runtime cost.**

This paper proposes *Zero-Trust Dependencies* to mitigate SSC vulnerabilities: we apply the NIST ZTA to software applications. First, we assess the expected effectiveness and configuration cost of Zero-Trust Dependencies using a study of third-party software libraries and their vulnerabilities. Then, we present a system design, ZTD_{SYS}, that enables the application of Zero-Trust Dependencies to software applications and a prototype, ZTD_{JAVA}, for Java applications. Finally, with evaluations on recreated vulnerabilities and realistic applications, we show that ZTD_{JAVA} can defend against prevalent vulnerability classes, introduces negligible cost, and is easy to configure and use.

I. Introduction

Integrating third-party libraries (TPLs) such as Log4j as dependencies accelerates software application development but introduces risks [1]–[3]. Dependencies are implicitly trusted by default and execute with the application's permissions. Vulnerabilities in dependencies, termed *software supply chain (SSC) vulnerabilities* [4], may cause undesirable application behavior [5]. They result in SSC attacks [6]. To mitigate analogous risks from assets in cloud systems and corporate networks, the US National Institute of Standards and Technology (NIST) has recommended [7] (and industry [8]–[10] and academia have adopted [11]–[14]) the **Zero-Trust Architecture** (ZTA) to place zero implicit trust in system actors. We propose applying ZTA *within* a software application to mitigate the risk of SSC vulnerabilities (Figure 1).

Many security defenses reduce the risks of using TPLs within an application. However, none sufficiently mitigates SSC vulnerabilities as they do not enable the ZTA principles of secure resource access, continuous monitoring, and least privileges for the application's dependencies and at low runtime cost (§II-C). *Application-level* sandboxes [15]–[18] do not operate on dependencies. *Import-restriction-based* [19]–[21] and *debloating-based* [22], [23] approaches do not validate when dependencies access resources. Existing *isolation-based* techniques [24]–[27] do not enable discovery of dependencies' least privileges, and they introduce high runtime costs.

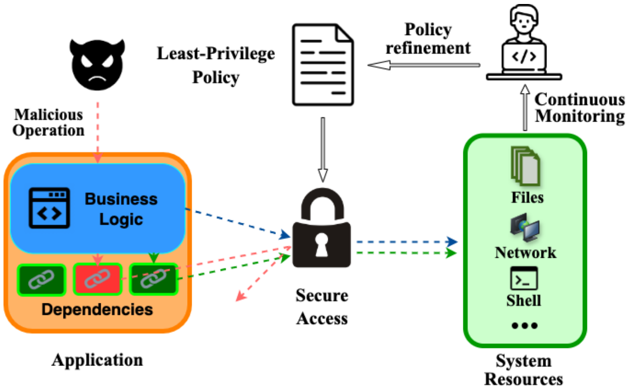

Fig. 1: The *Zero-Trust Dependencies* (ZTD) concept. To mitigate attacks exploiting vulnerable dependencies, a ZTD system provides *secure access* via runtime authorization, makes authorization decisions using a *least-privileges access* policy, and facilitates *continuous monitoring* of unexpected accesses.

This paper proposes *Zero-Trust Dependencies* (Figure 1), a concept that applies NIST's Zero-Trust Architecture to an application's dependencies to prevent software supply chain attacks. We begin with a feasibility study of the Zero-Trust Architecture idea, clarifying its suitability and design considerations through a study of software supply chain vulnerabilities and application dependency chains (§III). Motivated by our findings, we state the threat model (§IV-A), propose Zero-Trust Dependencies as a mitigation (§IV-B), and design a ZTD system, ZTD_{SYS} to enable secure resource access (§IV-D), discover least privileges (§IV-E), and continuously monitor dependency behaviors (§IV-F).

We implement a prototype, ZTD_{JAVA}, for Java applications (§V). We evaluate its effectiveness on recreated vulnerabilities, micro and macro performance costs, and the effort to audit policies of real applications. Our results show that fine-grained enforcement of a library's observed behavior (representing its least privileges) prevents all reproduced vulnerability exploits. ZTD_{JAVA} introduces much lower overhead than the state-of-art for Java. Additionally, only an average of five dependencies in each application require explicit policy specification.

In summary, we contribute:

- A feasibility study of applying Zero-Trust Architecture

[†] Some work performed as a Student Researcher at Google.

principles within a software application (§III).

- The *Zero-Trust Dependencies* concept and our ZTD_{SYS} design for mitigating SSC vulnerabilities (§IV).
- A prototype for Java, ZTD_{JAVA} (§V), and an evaluation of its cost and effectiveness on realistic Java applications (§VI).

Significance: Operationalizing a recommended security architecture will enable its adoption in software engineering, leading to more secure applications. Our feasibility study, design, and evaluation will inform engineers of the utility and cost of adopting this security architecture in their applications.

II. BACKGROUND

Here we define software supply chains and vulnerabilities (§II-A), introduce Zero-Trust Architecture as a mitigation (§II-B), and discuss limitations of existing defenses (§II-C).

A. Software Supply Chains (SSC) and Vulnerabilities

From a technical standpoint, *Software Supply Chains* encompass the systems and devices involved in producing a final software product [28]. This includes the application's source code, build tools, final packaged artifact, and third-party libraries used as dependencies [29]. The use of third-party libraries introduces new attack surfaces. They can be deliberately [30]–[32] or unintentionally (*e.g.,* Listing 1) introduced. These vulnerabilities are *Software Supply Chain (SSC) vulnerabilities* [4] and they enable *SSC Attacks* [6], [33] that compromise the host system and lead to consequences such as execution of arbitrary code, file manipulation and exfiltration of sensitive data. SSC attacks typically succeed because vulnerable dependencies inherit the application's permissions, obtaining more privileges than they need in typical use [20].

SSC vulnerabilities can be mitigated by securing the structure and use of a software supply chain. Per Okafor *et al.* [28], secure supply chains need transparency (where dependencies come from and what risks they contain), validation (reliability of dependencies), and separation (dependency isolation to avoid domino effects). *Transparency* can be improved through techniques such as software bills of materials (SBOMs) [34] and the Open-Source Vulnerabilities project [35]. *Validity* is promoted through software signing [36], [37] to verify the provenance of third-party libraries and the end-to-end security model [38] that prevents unauthorized alteration of source code. Our zero-trust dependencies concept, inspired by NIST's zero-trust architecture, improves *separation* between dependencies.

B. Zero-Trust Architecture as a Conceptual Framework

§II-A suggests that SSC vulnerability exploits succeed because dependencies are granted permission to access more resources than they typically need. The Zero-Trust Architecture (ZTA) [7], [40] protects access to resources in the context of a network and may analogously effectively mitigate SSC vulnerabilities. As defined by the USA's National Institute of Standards and Technology (NIST) in SP-800-207 [7], Zero-Trust Architecture is a set of cybersecurity concepts focused on protecting services (*resources*) within a system, grants no implicit trust to any user or assets (*subjects*) and requires subjects to gain explicit authorization to access any resource. Following Google [8], we summarize ZTA in 3 principles:

1) *Secure and Context-based Access:* Every access to a resource should be authorized.
2) *Continous Monitoring:* Organizations should monitor the state and activities of the subjects and use the insights gained to improve the creation and enforcement of policies.
3) *Least-Privilege Policy Enforcement:* Access policies should grant minimum access rights to subjects.

```
1   public void exploitTarget() {
2     String command = // exploit command
3
4     HttpUrlConnection con =
5       new URL(URL).openConnection();
6
7     con.setRequestMethod("POST");
8     con.setRequestProperty(
9     "spring.cloud.function.routing-expression",
10    "T(java.lang.Runtime).getRuntime().exec
11    (\"+command+\")");
12
13    int response = con.getResponseCode();
14  }
```

Listing 1: An exploit for a code Injection vulnerability (CVE-2022-22963) in Spring Cloud Function (SCF) [39]. On lines 8-11, an attacker executes a malicious command by using SCF's routing-expression functionality. Executing shell commands is not a privilege SCF needs, and thus would be blocked by a Zero-Trust Architecture.

The Zero-Trust Architecture has been applied to security-sensitive domains [41] including network infrastructure [42], [43] and cloud systems [8], [9], [11]. When applied to software applications, ZTA can mitigate SSC vulnerability exploits as it authorizes only legitimate access to operating system resources, based on a dependency's minimum privileges set.

C. Limitations of Existing Application Security Defenses

Existing security defenses reduce some risks from third-party libraries, but do not prevent SSC vulnerability exploits as they do not enforce zero-trust principles on all dependencies. We summarize these techniques in Table I.

Application-level defenses enforce security policies on the application [15], [18], [44] or its logical units [45]–[47]. However, as dependencies inherit the application's permissions, vulnerable dependencies may compromise sensitive system resources by leveraging permissions provided to the application. In addition, specifying security policies that enforce least privileges in modern applications is difficult and error prone [49], [50]. The Java Security Manager (JSM) [18], an application-level defense for Java applications, allows specifying fine-grained policies for secure resource access in Java applications. It was deprecated in 2021 due to lack of use, ascribed to its brittle permission model, difficult programming model, and poor performance [51].

Dependency-level defenses operate on dependencies and can constrain their behaviors at runtime. We distinguish three kinds. *Import-restriction-based* defenses [20], [21], [48] intercept dependencies as they are imported into an application and

Table I: Analysis of existing security defenses by ZTA principles. Columns indicate if each technique family provides secure resource access, supports least priv. discovery and enforcement, enables continuous monitoring for dependencies, and has low runtime costs.

Family	**Design**	**Res. Access**	**Lst-Prv. Enf.**	**Dep. Mon.**	**Low Cost**
[15], [16], [18], [44]	Access control	✓	✗	✗	✓
[45]–[47]	App decomp.	✓	✗	✗	✓
[20], [21], [48]	Import restrict.	✗	✗	✓	✓
[22], [23]	Debloating	✗	✗	✓	✓
[24]–[27]	Isolation	✓	✗	✓	✗
ZTD$_{\text{JAVA}}$	Access control	✓	✓	✓	✓

remove unauthorized functions. Meanwhile, *debloating-based* defenses [22], [23] remove unused or vulnerable functions from a library's source code. These designs prevent malicious code introduction in dependencies and introduce lower run-time costs. However, they do not validate dependency resource accesses and cannot prevent exploits of accidental vulnerabilities where the attacker only controls data in the app. *Isolation-based* defenses [24]–[27] execute dependencies in isolated compartments and can enable secure access to resources. However, they face low adoption rates [52], in part, due to their high performance overhead and the effort required to discover the least privileges of a compartment.

III. ZERO-TRUST ARCHITECTURE FEASIBILITY ANALYSIS

Previous works have demonstrated the feasibility of secure context-based access [26], [27] (ZTA principle 1) and dependency monitoring [20], [27] (ZTA principle 2). However, no relevant security defense (§II-C) has assessed the feasibility of least-privilege definitions for dependencies.

This section measures two distinct aspects of applying the principle of least-privilege policy on dependencies. First, ZTA requires that SSC vulnerability exploits on a dependency involve *resources not needed by that dependency*, so that a least-privilege policy would mitigate these vulnerabilities without impacting the application. Second, ZTA requires that *the cost of correctly configuring least-privilege policies be reasonable*, *e.g.,* that only a few dependencies need privileges and require policy specification. Thus we ask:

RQ1 What proportion of *Java SSC vulnerabilities* could be mitigated by enforcing least-privilege policies?

RQ2 What proportion of *dependencies* will need explicit policy authorization in an application?

To answer these questions, we measure SSC vulnerabilities and third-party libraries in the Java ecosystem.

Why Java? We situate this feasibility study, and our subsequent embodiment of ZTD, within Java. Java is a popular programming language [53], [54], and SSC attacks have been the most impactful in the Java ecosystem. For example, the Log4j vulnerability affected millions of Java applications [55] with estimated costs in the billions [56]. While there have been SSC attacks in other ecosystems like JavaScript and Python [57], their impact has been smaller due to their smaller server-side industry footprints.

Novelty: Prior works have studied the life cycle [58], [59] and dependency-tree propagation behaviors [60], [61] of SSC vulnerabilities. This study instead describes how SSC vulnerabilities access resources in third-party libraries.

A. *Methodology*

1) RQ1: We evaluated RQ1 by assessing the proportion of Java SSC vulnerabilities that involved access to operating system resources and examining whether they can be mitigated by coarse or fine-grained least-privilege policy enforcement.

We study recent and high-severity vulnerabilities in popular Maven Central libraries, as they represent prevalent and high-impact threats to applications. We obtained the 4462 vulnerabilities from the Open-Source Vulnerabilities (OSV) database [35]. We filtered for vulnerabilities that were published within the last 5 years, had a CVSSv3 rating of high or critical, and affected the top 10,000 depended-upon Maven Central libraries from the Ecosystems database [62]. This yielded 539 vulnerabilities in 252 unique libraries.

To answer RQ1, we randomly analyzed 118 of the 539 vulnerabilities (22%) [†]. This number is comparable to the number of vulnerabilities prior works [63], [64] studied. First, we classified them using the taxonomy of web security vulnerabilities [65] and their exploitation impact [66]. We report the proportion of vulnerabilities whose impact enables malicious access to resources. For these vulnerabilities, we identified the resources they expose and the API that the vulnerable code uses to access the resource. We also compiled a list of APIs that vulnerable libraries used to access each resource type. Next, we used CodeQL [67] and the compiled list of APIs to search for API calls that access the exposed resource in the affected library. If no calls were found, a coarse least-privilege policy that denies access to the specific resource type would prevent an exploit. However, if API calls were found, a fine-grained policy would be needed that only authorizes access to legitimately needed resource objects.

2) RQ2: RQ2 estimates the proportion of dependencies that access sensitive resources and would need policy authorization. From a library perspective, we measured the proportion of popular Maven Central libraries with direct access to OS resources. From an application perspective, we measured the proportion of dependencies in real applications that access OS resources during runtime. The results provide an upper bound and a common estimate of the number of dependencies that would require explicit policies. High values would threaten the feasibility of ZTA when applied to software dependencies.

For the upper bound, we obtained 23,569 Maven Central libraries sorted by their number of dependents, from the Ecosystems [62] database for our analysis. 388 libraries could

---

[†] This represents an 8% confidence interval at a 95% confidence level.

Table II: Vulnerability classes in Java third-party libraries. The table shows the consequences of exploitation, the required access to the operating system, and the count. We also cite Common Weakness Enumerations (CWEs) associated with each vulnerability class.

Vuln. Class	**Consequence**	**OS Resource**	**#**	**Perc**
Deserialization [69]	Remote Code Exec.	Shell, FS	21	18%
Access Control [70]	Unauth. App Access	N/A	21	18%
Resource Exhaust. [71]	Denial of Service	N/A	18	15%
Code Injection [72]	Remote Code Exec.	Shell	13	11%
Path Traversal [73]	File Read/Write	FS	12	10%
XSS, CSRF [74]	Unauth. Web Access	N/A	11	9%
XXE [75]	Data Exfiltration	FS, Net	5	4%
Command Injec. [76]	Remote Code Exec.	Shell	4	3%
Impl. flaw [77]	Varies	N/A	8	7%
Others	Varies	N/A	5	4%
Total	-	-	118	100%

not be cloned, and 18,495 failed to build due to missing dependencies. Similar to RQ1, we used CodeQL [67] to analyze the remaining 4,686 libraries, identify sensitive API calls, and report the percentage of libraries that access each type of resource. For the common estimate, we used the 2023 DaCapo benchmark suite for Java, consisting of 22 applications, including a web server (Tomcat), web application framework (Spring), IDE (Eclipse), database (Cassandra), and message bus (Kafka), and workloads that imitate industry needs [68]. Four applications did not build or run and we could not get the list of dependencies for nine applications. We analyzed the dependencies of the remaining nine applications using our CodeQL scheme. We instrumented the nine applications to record the resources each dependency accessed, executed the applications, and compared the resources that the dependencies accessed at runtime with the resources that the dependencies can access as indicated in their CodeQL results.

B. Results

1) RQ1: Will least-priv policies mitigate SSC vulns?:

Finding 1 (RQ1): 46% of Java SSC vulnerabilities can compromise an operating system resource. 58% (of 46%) can be mitigated using a coarse-grained least-privilege policy. 42% require finer-grained least-privilege policies that can control the resource objects they access.

Table II shows the different vulnerability classes in Java third-party libraries and the operating system resources they expose. Five vulnerability classes, comprising 46% of vulnerabilities, provide access to the file system, network, and shell. They enable a malicious actor to manipulate files, exfiltrate data, and execute commands remotely. From a study of malicious libraries, Ohm *et al.* [66] found that these objectives represented 97% of the analyzed attackers' goals.

Table III: Operating system resources exposed by vulnerabilities. RQ1 (Vulns) shows the vuln. count and indicates whether the vuln. library can access the resource. RQ2 (Maven) first column shows the proportion of 4387 analyzed Maven Central libraries that can access these resources directly. RQ2 (Maven) second column (a/b) shows the number of dependencies (out of 103) that can access a resource (a) and that accessed the resource within a DaCapo application (b).

	RQ1 (Vulns)			**RQ2 (Maven)**	
OS Resource	**Vuln Count**	**No Access**	**Access**	**% Total**	**Caps usage**
File read	7	1	6	33%	46/4
File write	11	1	10	28%	46/1
Network connection	7	3	4	12%	6/0
Shell execution	36	27	9	9%	15/0

Table III (second section) shows the number of vulnerabilities that expose different resource types and the proportion of vulnerable libraries that need access. Some vulnerabilities expose multiple resources. In 27 of 36 cases (75%), the shell is exposed when not needed. A coarse-grained policy can prevent these accesses. In contrast, in the 25 instances where file I/O or network access is exposed, the vulnerable component did not need the respective resource in only 5 instances (20%). For cases where the affected library needs the exposed resource, a more fine-grained notion of privilege would be needed, *e.g.,* permitting access to some files or remote hosts but not others. These observations guide our policy definition in §IV-C2.

2) RQ2: How many deps. need explicit least-priv. policies?:

Finding 2 (RQ2): In our upper bound estimate, 33%, 12%, and 9% of Maven Central libraries would require permissions to directly access file, network, and shell resources, respectively. However, only 4% of the 103 dependencies in 9 applications required explicit least-privilege policies.

Our library-focused result is shown in Table III (last two columns). First, less than 35% of the popular third-party libraries of the Maven Central ecosystem directly access any particular resource. For example, while 36/55 vulnerabilities (65%) provide access to the Shell, only 16% (9/55) of the vulnerable libraries and 9% of popular Maven Central libraries require Shell permissions.

From an application perspective, in the 9 studied applications (comprising 103 dependencies), while up to 46 dependencies (45%) could access the file system, only 4 dependencies and 1 dependency read or wrote to a file within the application. Similarly, while 15 dependencies could execute shell commands, none of the dependencies executed a shell command while executed by the application.

C. Discussion

ZTA Effectiveness: Per Table II, 55/118 of SSC vulnerabilities allow resource compromise and can be mitigated by a secure access control system. Of these, 32 (58%) could be addressed with a coarse-grained least-privilege policy based on resource type access, while 23 (42%) require finer-grained policies based on specific resource objects. Recall that existing SSC security defenses support only coarse-grained privileges (§II-C) and thus are ineffective against 42% of SSC vulnerabilities.

ZTA Configuration Cost: To effectively apply ZTA, one must specify policies for application dependencies (defaulting to no trust). Hence, the configuration cost depends on the number of dependencies that require policy specification. A high configuration cost would make ZTA impracticable in this context. Table III shows that most dependencies do not require access to sensitive resources. When they might, the access is typically not used by their dependents.

Study Limitations: We note two cases in which our CodeQL measurement will fail. First, a resource might be accessed with an API not covered by our queries, which we mitigated by building queries based on real exploits. Second, a resource might be accessed through indirection (*e.g.,* callbacks or Java reflection). This threat is a consequence of using static analysis [20], [21], which we chose for its scalability.

IV. ZERO-TRUST DEPENDENCIES: CONCEPT AND DESIGN

This section introduces Zero-Trust Dependencies (ZTD) as a security architecture that mitigates SSC vulnerabilities. We outline the threat model in §IV-A, define Zero-Trust Dependencies in §IV-B, and discuss design considerations for different ZTD_{SYS} components (§IV-C – §IV-F).

A. System and Threat Model

System Model: An application depends on a vulnerable third-party library that can access system resources using untrusted (*i.e.,* user-controlled) application data (shown in §III). The resource access operation can be initiated by the vulnerable dependency but executed directly (using function calls or callbacks) or asynchronously (in independent or child threads) by other dependencies or the application itself [27].

Threat Model: We include some threats and exclude others.

- *In-scope:* The attacker controls either the source code of the vulnerable dependencies or the data that the application passes to the vulnerable dependency. Hence, they can compromise the confidentiality and integrity of the host system by exploiting the vulnerabilities in the dependency.
- *Out-of-scope:* We do not consider threats from the SSC vulnerabilities that cause denial-of-service or unauthorized application access shown in Table II. Denial-of-service attacks can compromise the application's availability but they can be easily detected by monitoring the application [78]. Sensitive application functions should require additional authorization and should not be accessible from code injection attacks.

This threat model is stronger than the import-restriction and debloating-based models (§II-C) as it also prevents exploits that only control data passed into the dependency.

B. The Zero-Trust Dependencies Concept

NIST's Zero-Trust Architecture protects resources within a network (§II-B). In this paper, we define *resources* [7] in the software application context as data and computing services provided by the operating system (OS) that an application can operate on. Data services include the file and network system, which enable access to confidential information that the OS or its applications produce or use. Computing services include the shell execution system that executes shell commands in the OS. Operations refer to Read, Write, and Execute (RWX) operations that can be performed on these resources.

To mitigate the use of SSC vulnerabilities to compromise system resources, we introduce *Zero-Trust Dependencies* as an adaptation of the NIST Zero-Trust Architecture (§II-B) to the context of software applications.

> **Zero-Trust Dependencies** is a software engineering paradigm that grants no implicit trust to dependencies in an application. It requires that dependencies possess explicit authorization to create or operate on resources within the application's operating system.

Zero-Trust Dependencies involves three principles:

1) **ZTD-P$_1$: Secure Access:** The access of dependencies to resources should be authorized using the configured access policies and the application's execution context.
2) **ZTD-P$_2$: Least Privileges Enforcement:** The access policies should reflect the least privilege set that dependencies require to operate within the application.
3) **ZTD-P$_3$: Continous Monitoring:** The software engineers should be able to continuously monitor the resource access of dependencies and use the insights gained to improve threat intelligence and policy specification.

The ZTD paradigm's **security guarantee** is: an application's dependencies cannot compromise the *Confidentiality* or *Integrity* [79] of the operating system's resources. Consequently, ZTD reduces the risks that applications face from using third-party libraries.

Dependency-level security defenses in §II-C do not enforce the three ZTD principles in an application. Hence, we also introduce the *ZTD System* (or ZTD_{SYS} for short).

> A **ZTD System** is the set of access control policy designs, algorithms, and tools that enable software engineers to apply the ZTD paradigm in their applications.

C. ZTD Policy Design

Zero-Trust Dependencies require policies that specify the least privileges of dependencies. We propose:

1) ZTD Permission Model: The ZTD permission model is designed to protect resources, within the operating system, that vulnerability exploitation may expose. We use the access matrix terminology from Sandhu *et al.* [80] to describe the ZTD permission model. Permissions are given to dependencies to perform operations on resource objects. Table IV shows the three resource types supported and the operations that can be performed. We focus on high-risk resource types to simplify policy files and reduce permission verification frequency. These resource types are high-risk because they enable code execution, data exfiltration, and file manipulation attacks, *i.e.,* the major targets of supply chain attacks [66].

```
# Coarse-grained Policy
{
  "com.app.bar": {
    "fs.read": true,
    "fs.read.denied": ["/tmp","/sensitive"],
    ...
  }

# Fine-grained Policy
  {
  "com.foo.baz": {
    "fs.write": true,
    "fs.write.allowed": ["app/logs"],
    "runtime.exec": true,
    "runtime.exec.transitive": ["whoami"]
  },
  ...
}
```

Listing 2: Policy Specification file. It supports coarse-grained (*e.g.,* 'fs.read') and fine-grained (*e.g.,* 'fs.read.allowed') permissions.

Table IV: Resource types supported by the ZTD permission model.

Resource Type	**Operation**	**Resource Objects**
File System	Read, write	Files
Network System	Connect	Network URLs
Shell System	execution	Shell commands

2) Permission Specification Granularity: Similar to the Java Security Manager's permission design [81], ZTD$_{\text{SYS}}$ allows specifying coarse and fine-grained permissions. Listing 2 shows samples of a policy file.

Coarse-grained Permissions authorize a dependency to access any object of the specified resource type. For example, a coarse file read permission (`fs.read`) in Listing 2 allows the `com.app.bar` dependency to read any file in the OS, except files specified in `fs.read.denied`.

Fine-grained Permissions authorize a dependency to access explicitly authorized resources. ZTD$_{\text{SYS}}$ provides *direct* and *transitive* permissions that specify if an object can be accessed directly or through another dependency. For example, Listing 2 grants the `com.foo.baz` dependency the permission to write files, but only allows writing to those in `fs.write.allowed`. The dependency is also granted the transitive runtime execution permission, so it can transitively execute commands specified in `runtime.exec.transitive`. Hence, while `com.foo.baz` cannot execute the `whomai` command directly, it can use another dependency that has direct permission to execute the command.

D. ZTD-P$_1$: Secure and Context-Sensitive Access Control

The System Model in §IV-A shows that multiple dependencies in an application can interact to perform a given task, using direct or asynchronous calls. ZTD$_{\text{SYS}}$ relies on a context-sensitive access control system that considers this set of interacting dependencies when making access decisions and prevents malicious dependencies from leveraging permissions granted to other dependencies. This section and Listing 3 shows how ZTD$_{\text{SYS}}$ handles components that interact directly and asynchronously.

1) Handling Direct Dependency Interactions: A dependency can call functions from another dependency to access operating system resources. These dependencies will be contained in the executing thread's call stack. Hence, ZTD$_{\text{SYS}}$ grants access to a resource if all dependencies on the call stack have the necessary direct or transitive permissions to access the resource (Listing 3 lines 4-5, 20-28). This approach ensures that no dependency has permission to access any resource on its own and prevents malicious dependencies from exploiting privileged dependencies to access unauthorized resources.

2) Handling Indirect Dependency Interactions: In multi-threaded applications, a dependency can delegate a resource access operation to another dependency or the application (*delegates*). In a simpler case, the delegate may create a child thread to access the resource [27]. Then, the call stack will not contain the dependency that initiated the operation. ZTD$_{\text{SYS}}$ handles this interthread delegation scenario by using the permissions of dependencies in the parent and child thread's call stack to authorize access (Listing 3 lines 8-17). When a child thread is created, its parent's dependency policies are saved. The saved policies are retrieved and verified whenever the child thread accesses any resource. As reported in §V-B, this design does not handle more complex delegations involving two independent threads or processes.

3) Enforcing ZTD Policies: When access to a resource is denied, ZTD$_{\text{SYS}}$ provides two enforcement options that balance the reliability needs of an application against the risk appetite of the engineering team:

a) *Fatal Enforcement mode:* This option raises an exception to block the unauthorized action. Software engineers working on security-sensitive applications with low-risk tolerance may prefer this mode. They can implement exception-handling routines to prevent disruption to their applications.

b) *Non-fatal Enforcement mode:* This option does not interfere with the application. Instead, ZTD$_{\text{SYS}}$ sends an unauthorized access alert to the application maintainers, who will investigate the incident and begin any necessary remedial actions. Software engineers maintaining applications with high availability needs or in low security-sensitivity environments may prefer this mode.

E. ZTD-P$_2$: Discovery of Least Privileges

The huge number of dependencies in modern applications makes it infeasible to manually specify the least privilege policies for an application's dependencies. ZTD$_{\text{SYS}}$ infers the dependencies' least privileges from their observed execution behavior. This approach is motivated by findings from §III that vulnerabilities commonly expose resources that are not needed by the vulnerable dependencies during legitimate executions. Listing 4 shows the algorithm employed by ZTD$_{\text{SYS}}$ for discovering and generating the least privilege policies for an application's dependencies. ZTD$_{\text{SYS}}$ maintains a policy object for each dependency in the application. At runtime, it intercepts invocations of sensitive APIs that can access an operating system resource. It retrieves the classes and dependencies in the call stack and updates the policy of each

```
# In: resource type, specific item, and operation
# Out: True if access is approved, else False
def authorizeAccess(resType, resItem, resOp):
  callStackClasses = getClassesFromCallStack()
  depsPolicies = getPoliciesForDeps(callStackClasses)

  ## This part handles thread-based delegation
  if resType == "thread": # Child thread is created
     # Propagate parent's policy to child thread
     saveThreadParentPolicies(resItem, depsPolicies)
     return True

  ## For access from child threads,
  ## we check its contextPolicy and that of its parent
  parentContextPolicy = getParentPoliciesForThread()
  if parentContextPolicy != null:
    depsPolicies.extend(parentContextPolicy)

  ## Having obtained the appropriate policy, check it
  for policy in depsPolicies:
    if policy.checkAccess(resType, resItem, resOp):
       continue

    #Policy does not grant access
    if fatal_enforcement_mode:
      raiseException()
    else
      sendAlert()
```

Listing 3: Context-sensitive permission verification and permission inheritance. A dependency without defined permissions is authorized if its caller is authorized. An operation is authorized if all dependencies on the call stack are authorized.

```
# Function called when a resource access is requested
# In: resource type, specific item, and operation
def onPermissionRequested(resType, resOp, resItem):

  # We get the classes in the call stack
  classes = getClassesFromCallStack()

  # For each class, we get its parent dependency
  depPolicies = getPoliciesForDeps(classes)
  # The first dependency is the direct caller.
  depPolicies[0].addCoarsePerm(resType, resOp)
  depPolicies[0].addFinePerm(resType, resOp, resItem)

  # We assign transitive permissions to other deps
  for (i = 1; i < depPolicies.length(); i++) {
    depPolicies[i].addFinePerm(resType, resOp, resItem)
  }
```

Listing 4: Pseudocode for discovering and generating the least privilege policy from the observed execution behavior.

dependency with the necessary permission and the name of the specific resource object that is being accessed. At intervals, it writes the dependencies' policies to a policy specification file.

We also considered inferring a dependency's least privileges by analyzing the library's source code and identifying invocations of function calls that access different resources. A similar approach was applied by the dependency-level security defenses [20]. However, this approach does not enable fine-grained policy generation for dependencies as it is difficult to statically determine the specific resource objects that a dependency may dynamically access. Furthermore, the use of indirection in many languages (*e.g.,* C/C++ function pointers, Java reflection) will lead to an imprecise identification of the sensitive APIs that a dependency can access.

Table V: Java classes and instrumented methods that provide operating system resource access.

Operation	**Class**	**Instrumented Method**
File Read	FileInputStream	<Constructor>
File Write	FileOutputStream	<Constructor>
Network Connect	Socket	connect()
Runtime Execution	ProcessBuilder	start()

F. *ZTD-P$_3$: Risk Awareness through Continuous Monitoring*

As noted in §IV-B, ZTD requires engineers to continuously monitor their dependencies' resource access. ZTD$_{SYS}$ supports continuous monitoring using the approaches for least privilege discovery and policy enforcement given above.

When run throughout the application's lifetime, the least privilege discovery framework continuously informs software engineers of all resources their dependencies accessed. This improves engineers' risk awareness as they can audit any dependency that accessed a sensitive resource to ensure malicious actors cannot hijack such access.

In addition, when non-fatal enforcement mode is enabled, ZTD informs software engineers of any unexpected resource access by a dependency, facilitating an investigation. Further investigations that show the access was benign and necessary can lead to improvements of the provided policy.

V. ZTD$_{JAVA}$: A ZTD$_{SYS}$ IMPL. FOR JAVA APPLICATIONS

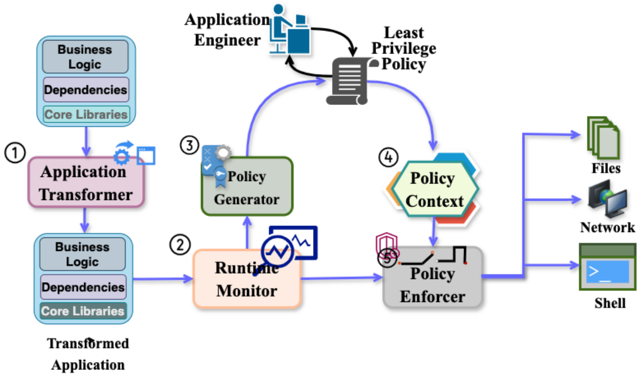

Fig. 2: The ZTD$_{SYS}$ design has five components. The application transformer instruments the application. The runtime monitor tracks dependencies' access to resources. The policy generator generates the least privilege policies for dependencies. The policy context loads the generated policies. The policy enforcer authorizes access.

We implemented a prototype of ZTD$_{SYS}$ for Java applications: ZTD$_{JAVA}$. Java applications comprise classes containing the application's business logic, classes from dependencies, and core classes provided by the Java Development Kit (JDK). The JDK core classes allow the application and its dependencies to access operating system resources and use other features provided by the language. Table V shows the core classes for accessing the file, network, and shell system.

Mapping Dependency Policies to Java Classes at Runtime: In Java applications, the classes in the call stack do not indicate their parent dependency. However, ZTD_{SYS} requires that the policies specified for dependencies apply to all classes within the dependency. Hence, we need to map the classes in the call stack to the specified dependency policy. ZTD_{JAVA} uses the heuristic that Java class names are formed from the directory tree structure containing the class. Classes in the same dependency share a common root directory path and their names share a *common prefix* representing their shared directory path. We refer to this common prefix as the dependency's *namespace*, as it contains all classes in the dependency. For dependencies from the Maven Central registry, this common prefix is usually obtained by combining the unique Group ID and Artifact ID of the library in the registry. Hence, the `com.app.bar` policy in Listing 2 will apply to all classes with names beginning with `com.app.bar`.

A. Implementation Details of ZTD_{JAVA}'s Components

ZTD_{JAVA} is implemented in 2,284 lines of Java code. As shown in Figure 2, ZTD_{JAVA} comprises five components. We discuss the implementation of each component below.

1) Application Transformer (AT) ①: The Application transformer modifies the bytecodes of an application at runtime. It is supplied as a command line argument for the *java* command used to execute the application. It takes in a list of classes and methods to modify. It uses the Java Instrumentation API [82] to intercept the specified target classes as they are loaded into the Java Virtual Machine and uses the ASM bytecode modification library [83] to insert a direct call to the runtime monitor at the start of the specified method. By default, the AT instruments the methods in Table V, but it can be configured to transform fewer or more methods, depending on the application's security needs.

2) Runtime Monitor ②: The runtime monitor is called whenever the application or a dependency attempts to access a protected resource using an instrumented method. Depending on ZTD_{JAVA}'s configuration, the runtime monitor invokes the policy generator and/or the policy enforcer.

3) Policy Generator ③: The policy generator generates a least-privilege policy for each dependency using the algorithm in Listing 4. When a resource is accessed, it adds the required permission to the policy objects of the dependencies on the call stack. The policy objects are written to the least-privilege policy file at specified intervals or during application shutdown. In addition, the generated policy file also informs software engineers of the resources their dependencies access and the potential risks they pose.

4) Policy Context ④: The policy context stores the specified policies for each dependency. It is implemented using a Patricia tree [84], where the nodes contain the dot-separated components of the dependency name, and the policies are contained in the leaf nodes. As classes do not contain unique identifiers of their parent dependencies, the context store is designed to get the policy of a class' parent dependency using only the class name. Hence, the policy for classes from the `com.foo.baz` dependency can be obtained by using the first three dot-separated components of their class name to transverse the `com`, `foo`, and `baz` nodes in the tree. When the application is started, ZTD_{JAVA} creates the policy context from the provided policy file. At runtime, the policy enforcer uses the context to retrieve the policies that should be applied to each class on the call stack.

5) Policy Enforcer ⑤: The policy enforcer uses the context-based authorization algorithm in Listing 3 to validate and authorize access to OS resources. First, we get the set of dependencies for the classes on the call stack, retrieve the policies for these dependencies, and verify that all policies have permission to access the resource.

B. Limitations of ZTD_{JAVA}

- *Asynchronous delegation:* ZTD_{SYS}'s handling of indirect dependency interactions (§IV-D) does not cover delegations between independent threads or processes created by different dependencies. This can occur if a dependency sends user-controlled data to a different thread where the data will influence resource access. A taint-tracking-based technique can associate permissions with communications between threads and processes, which will incur high runtime costs [85]. Alternatively, the permission model in ZTD (§IV-C1) could be expanded to recognize threads and processes as resources that require explicit access permissions.
- *Use of Native Execution:* An attacker can use native libraries to bypass ZTD's policy authorization. Our application transformer could protect the API that allows the use of native libraries (e.g. *System.loadLibrary()* in Java). Alternative designs execute native libraries in sandboxes [24], [26] but this introduces significant performance overhead.
- *False positives from incomplete generated policies:* We generate least privilege policies for each dependency based on their observed execution behaviors, similar to sandbox mining techniques [86]–[89]. These policies will block any unexplored behaviors. To address this, ZTD_{SYS} provides a non-fatal enforcement mode that alerts engineers rather than disrupting the application.
- *Namespace Pollution:* While rare, multiple dependencies can share the same namespace in a Java application. A malicious actor can create a dependency whose class names share the same namespace as another dependency to inherit the legitimate dependency's privileges. However, enforcing fine-grained policies will limit the attacker to only resources that the legitimate dependency can access.

VI. ZTD_{JAVA}: EVALUATION

This section evaluates ZTD_{JAVA}'s effectiveness, runtime costs, and policy configuration costs with the following questions.

RQ1 *Effectiveness* - Does ZTD_{JAVA} prevent SSC vulnerability exploits that access operating system resources?

RQ2 *Performance cost* - What is the performance cost of ZTD_{JAVA} on realistic applications?

RQ3 *Configuration Effort* - How much effort is required to audit policies generated for realistic applications?

A. Setup

1) Vulnerability Selection: We use the vulnerabilities from §III-A. We selected three vulnerabilities from each vulnerability class (Table II), prioritizing vulnerabilities with a publicly available exploit proof of concept (POC).

2) Application Selection: We used the DaCapo benchmark applications from §III-A. To obtain least-privilege policies for each application's dependencies, we used ZTD$_{\text{JAVA}}$'s policy generator and the DaCapo benchmark suite's test cases. These tests ran under ZTD$_{\text{JAVA}}$ without issue.

3) Baseline Selection: We use the Java Security Manager (JSM) [18] as a performance evaluation baseline as it shares similar requirements with ZTD$_{\text{JAVA}}$: authenticating access to OS resources for Java applications. We could not quantitatively compare to extant dependency-level security defenses (Table I), as none target the Java ecosystem.

B. Methodology

1) RQ1: Defense against exploits leveraging OS Resources: To evaluate RQ1, we took two steps. First, we tested ZTD$_{\text{JAVA}}$'s ability to mitigate SSC vulnerabilities. For each vulnerability, we created a sample application that depended on the vulnerable library and could be exploited using the available POC. We ran each application with ZTD$_{\text{JAVA}}$ to check if the exploits were blocked. Our results are in Table VI.

Secondly, we assessed whether the least-privilege policy generated by ZTD$_{\text{JAVA}}$ could prevent exploits in real applications. We injected 9 vulnerabilities into 4 applications: Biojava, Fop, Graphchi, and Zxing. We selected applications built with Maven so that we could easily add vulnerable libraries as dependencies. We could not exploit 2 vulnerabilities within BioJava because the exploits required Java 8 and Biojava required a minimum of Java 11 to run. We used ZTD$_{\text{JAVA}}$ and the provided workload to generate least-privilege policies for each application and evaluated the effectiveness of the generated policies in blocking exploits of the injected vulnerabilities. While we injected only nine vulnerabilities in this step, we expect the generated policies to also prevent exploits of remaining vulnerabilities as the vulnerabilities access resources not allowed by the policies. In addition, unlike previous security defenses [20], [21] that are only evaluated on vulnerabilities recreated in simple applications, we demonstrated ZTD$_{\text{JAVA}}$'s ability to generate and enforce policies for real applications.

We note that our evaluation uses only vulnerabilities analyzed in §III and, therefore, the result may not generalize to new vulnerabilities. However, the design of ZTD$_{\text{JAVA}}$ is based on the general ZTA framework. This should help mitigate all vulnerabilities covered by the threat model in §IV-A.

2) RQ2: ZTD$_{\text{JAVA}}$'s Performance Impact: First, we measured the cost of each instrumented operation with and without ZTD$_{\text{JAVA}}$'s instrumentation. We used the Java Microbenchmarking Harness (JMH) library [90] maintained by OpenJDK to avoid microbenchmarking pitfalls [91].

Table VI: ZTD$_{\text{JAVA}}$'s performance in preventing host-system-compromising exploits. It successfully blocks selected exploits. We injected 7 vulns. in 4 applications and 2 vulnerabilities in only 3 applications. Their exploits were blocked. A '–' means we did not inject this vulnerability.

CVE ID	Vuln. class	Impact	S. Apps	D. Apps
CVE-2020-9547	Des.	Code Ex.	✓	-
CVE-2020-8441	Des.	Code Ex.	✓	3/3
CVE-2022-36944	Des.	File Man.	✓	-
CVE-2021-44228	Code Inj.	Code Ex.	✓	3/3
CVE-2022-33980	Code Inj.	Code Ex.	✓	4/4
CVE-2022-22963	Code Inj.	Code Ex.	✓	-
CVE-2023-39021	Comm. Inj.	Code Ex.	✓	4/4
CVE-2023-39020	Comm. Inj.	Code Ex.	✓	4/4
CVE-2022-25914	Comm. Inj.	Code Ex.	✓	4/4
CVE-2022-0839	XXE	Data Exf.	✓	4/4
CVE-2021-23463	XXE	Data Exf.	✓	-
CVE-2019-10172	XXE	Data Exf.	✓	4/4
CVE-2022-4244	Path Trav.	File Man.	✓	4/4
CVE-2020-17518	Path Trav.	File Man.	✓	-
CVE-2020-17519	Path Trav.	File Exf.	✓	-

Second, we measured performance while increasing the number of dependencies (*expected: constant*) and the number of classes from different dependencies in the call stack during a method's invocation (*expected: linear*).

Third, we profiled the applications. We measured the execution time of each application without sandboxing, with ZTD$_{\text{JAVA}}$ (configured in four modes), and with JSM. We used the *converge* feature of the DaCapo benchmark harness [†].

3) RQ3: ZTD$_{\text{JAVA}}$'s Policy Configuration Effort: We measured the effort to audit the least-privilege policies generated by ZTD$_{\text{JAVA}}$ for DaCapo benchmark applications. As default policies are automatically generated, engineers should not need to master a new policy language. Therefore, the configuration effort will be spent mainly to audit and refine the generated policies.

To the best of our knowledge, there is no existing metric for configuration effort in this context. We therefore estimate that the configuration effort depends on two factors: (1) the number of dependencies for which ZTD$_{\text{JAVA}}$ generated a non-empty policy, and (2) the number of permissions provided in each such policy. The first metric captures the number of policies that engineers would need to study. The engineer may also need to assess if each provided permission is valid, so the second metric captures the level of analysis that they must perform for each dependency. For each application, we report the number of dependency policies generated and the average number of permissions per policy.

C. Results

1) RQ1: Defense against exploits leveraging OS Resources: Table VI shows details of the vulnerabilities we recreated. As shown, ZTD$_{\text{JAVA}}$ blocked the exploits of vulnerabilities in the sample applications, When applied to real applications, we find that the least privilege policies of all 4 applications were sufficient to prevent the 9 vulnerability exploits.

[†] An application is executed until three executions have times within 3%, then the next iteration's time is kept. We repeated the average of 3 trials.

Table VII: Microbenchmark of security-sensitive operations. Uncertainty is reported after the measurements converge.

Operation	**Vanilla** (μs)	**ZTD$_{\text{JAVA}}$** (μs)
File Read	7.96 ± 0.27	23.62 ± 0.20 (196.7%)
File Write	33.78 ± 1.73	33.23 ± 0.56 (-1.6%)
Socket Connect	71.23 ± 2.56	72.19 ± 2.34 (1.3%)
Shell Execution	320.69 ± 27.9	345.70 ± 3.98 (7.8%)

Table VIII: Execution time overheads and ZTD$_{\text{JAVA}}$ configuration effort on the DaCapo applications. The table shows dependency and authorization call counts, the overhead of ZTD$_{\text{JAVA}}$ and JSM, and the configuration effort for each app (x/y means x dependencies need policies and each policy provides an average of y permissions). '–': data unavailable because (column 2) application is not built with Maven/Gradle, or (column 6) JSM could not execute.

App Name	**Deps count**	**Auth calls**	**Vanilla**	**ZTD$_{\text{JAVA}}$**	**JSM**	**Config Effort**
Avrora	3	6	10,036	0.52%	0.18%	2/2
Batik	24	16	1,605	0.83%	–	3/2.7
Biojava	34	6	10,589	-0.56%	-0.75%	7/2
Eclipse	–	23k	16,464	2.91%	201.04%	9/3
Fop	30	94	734	0.23%	28.60%	8/2.5
Graphchi	40	97	5,245	-3.18%	-0.19%	3/2
H2	–	10	3,599	0.60%	-0.27%	2/2
Luindex	–	11	5,933	-0.36%	14.22%	4/2
Lusearch	–	6k	3,534	-0.48%	-0.50%	4/2
PMD	47	12	1,953	-1.02%	0.96%	3/2
Spring	111	–	5,800	-1.17%	–	17/1.6
Tomcat	–	1k	4,229	0.50%	10.99%	19/2.3
Tradebeans	–	42	23,401	0.07%	–	13/2.9
Tradesoap	–	37	13,129	0.10%	–	13/3
Xalan	–	9k	978	-2.04%	-0.58%	5/2
Zxing	5	2.5k	1,290	2.53%	-0.96%	3/1.7
Medians	–	–	–	-0.03%	21.06%	5/2

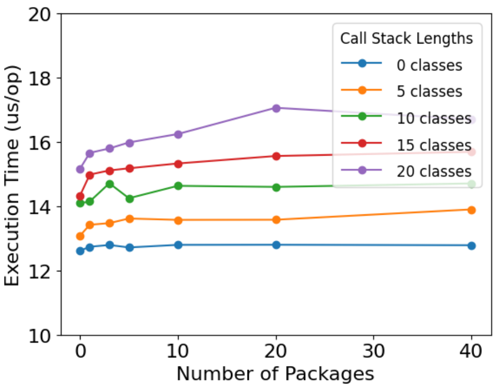

Fig. 3: Microbenchmarking results for the policy authorization operation, varying the number of dependencies in the application and the call stack sizes. As predicted, the execution time is constant with the dependency count and linear to the call stack lengths.

2) RQ2: ZTD$_{\text{JAVA}}$ Performance Impact: Table VII shows the results of microbenchmarks for 4 operations: file read, file write, socket connection, and shell execution. ZTD$_{\text{JAVA}}$ introduces modest overhead in file write, socket, and shell operations. For scaling, Figure 3 shows that ZTD$_{\text{JAVA}}$'s performance does not depend on the number of dependencies and is marginally affected by call stack depth.

Table VIII shows the profile results on applications from the DaCapo benchmark suite. ZTD$_{\text{JAVA}}$ introduces no noticeable overhead on 7 applications, with <1% overhead on 13 of the 16 applications. The JSM, despite operating at a higher abstraction level, introduces >10% overhead on four of sixteen applications, with ~200% overhead in Eclipse.

We attribute ZTD$_{\text{JAVA}}$'s minimal overhead, compared to the JSM, to two factors. First, ZTD$_{\text{JAVA}}$ performs policy authorization checks at the dependency granularity, *i.e.,* once per dependency in the call stack (Listing 3). In contrast, the JSM checks permissions at the *class* granularity in the call stack. Second, ZTD$_{\text{JAVA}}$ performs less frequent policy authorization checks. The default implementation enforces only 4 permissions while the JSM enforces 28 permissions and protects 100+ methods [81].

3) RQ3: ZTD$_{\text{JAVA}}$'s Configuration Effort: The last column of Table VIII shows the number of dependency policies generated for each application and the average number of permissions provided in each policy. 9 of the 16 applications have less than 5 dependencies that require policy specification. Policies provide only 1-3 permissions to the dependency. All policies were generated by ZTD$_{\text{JAVA}}$ without manual effort and executed on applications without failure.

Note that the count of policies and permissions in Table VIII includes both direct and transitive permissions. Hence, compared to the data in Table VIII, only fewer dependencies directly accessed any operating system resources.

VII. DISCUSSION

This section discusses ZTD$_{\text{JAVA}}$'s application to software applications (§VII-A) and other future directions (§VII-B).

A. Applying ZTD$_{\text{JAVA}}$ to Software Applications

1) Potential Usage Scenarios of ZTD$_{\text{JAVA}}$: ZTD aims to protect software applications from SSC vulnerabilities. We foresee three potential usage scenarios for ZTD$_{\text{SYS}}$ based on the risk tolerance and reliability needs of an application.

Proactive Security: Applications run in fatal enforcement mode (§IV-D3). The engineering team observes dependency resource access, generates policies, and enforces them to block abnormal resource access. This mode provides high security, but enforcing incomplete policies can lead to false positives and disrupt the application. This mode would have prevented the Equifax breach (CVE-2023-50164) [6] and attacks due to the Log4J vulnerability (CVE-2021-44228) [56].

Reactive Security: Applications generate policies and enforce them nonfatally (§IV-D3). This alerts engineers to unauthorized access without disrupting the application. This mode would have detected the Equifax breach earlier, preventing the reported 76-day dwell time [92] and limited its impact.

Emergency Security: Applications have policy discovery enabled (§IV-E) but no enforcement. When new vulnerabilities are reported in a dependency, the policies discovered for that dependency can be modified and enforced to mitigate the vulnerability by blocking access to any exposed resource. This mode can act as *emergency first-aid* and would have prevented software engineers from shutting down applications after the discovery of CVE-2021-44228 [93].

2) Test cases for generating least-privilege policies: In the usage scenarios described, the engineering team responsible for the application is tasked with selecting the test cases for policy generation. They can use available workloads for end-to-end tests or use fuzzing to generate new high-coverage test cases [94], [95]. Alternatively, they can run their application in ZTD_{SYS}'s policy discovery mode until they feel confident that the necessary functionalities have been exercised. This process is performed once for the entire application and needs only to be repeated when new functionalities or dependencies are introduced. Furthermore, discovered least-privilege policies are unlikely to be affected by dependency version updates, as such updates rarely introduce new permission requirements [20]. In the future, we intend to implement lightweight coverage metrics that help engineers decide when sufficient behaviors have been covered.

3) Barriers to ZTD_{JAVA} Adoption: While ZTD_{JAVA} addresses SSC vulnerabilities, we foresee two barriers to adoption.

a) Engineers may fear that runtime modifications might introduce errors [96]. ZTD_{JAVA} mitigates this concern by modifying only a few (4) classes. All are in JDK core.
b) False positives may occur if ZTD_{JAVA} encounters legitimate resource accesses that were not previously observed. We mitigate this by adding the non-fatal enforcement mode where any false positives will not disrupt the application. A possible ZTD_{JAVA} utility would be a code coverage metric to assess if sufficient program behavior has been covered.

4) ZTD_{JAVA} vs. the Java Security Manager (JSM): Both ZTD_{JAVA} and the (deprecated) JSM address Java application vulnerabilities via resource access authorization. As noted in §II-C, JSM had high runtime costs (up to 200%) as well as high policy complexity: JSM protected 100+ operations with 28 different permission types [81]. Based on common SSC vulnerabilities, ZTD_{JAVA} protects 3 sensitive resources by default, allowing lower performance costs and policy complexity. However, ZTD_{JAVA}'s application transformer (§V) can also be configured to protect a different set of resources according to the security and performance needs of the application. Furthermore, ZTD_{JAVA} overcomes the usability flaws that led to the lack of use of JSM and subsequent deprecation [51]. It features a flexible permission model (§IV-C) that allows coarse-grained policies for easy-to-develop policies and fine-grained policies for stronger security, easier programming with automated policy discovery (§IV-E), and low performance cost (§VI-C2). While ZTD_{JAVA} is designed to address vulnerabilities in an application's dependencies, it can also be extended to mitigate the application's own vulnerabilities, serving as an effective replacement for the deprecated JSM.

5) Design Choice: Protect Resources or Operations: In line with ZTA, ZTD_{SYS} protects functions that access resources. This approach differs from JSM [18] and some other dependency-level defenses [21], [26], which protect functions performing sensitive operations. Protecting sensitive operations provides finer control over dependencies' behavior. However, the size, complexity, and diversity of applications and programming languages mean there are many 'sensitive' operations, each requiring a different permission. As seen with JSM (§VII-A4), this leads to increased policy complexity and runtime performance overhead. To provide a balance, ZTD_{SYS} offers a customizable application transformer for adding or disabling permissions.

B. Future Directions

1) Improving the visibility of third-party library capabilities: Recent supply chain vulnerabilities like Log4J/Log4Shell [56] highlight the need for third-party library engineers to communicate the functionalities and resource usage of third-party libraries. This practice would help application engineers assess their risks, and discourage the provision of unnecessary OS resource capabilities to these libraries. Initiatives like Software Bill of Materials [34] and Open-source Insights [97] could make this task machine-checkable if these approaches were extended to document the capabilities of third-party libraries. ZTD_{JAVA}'s policy generation component can assist engineers in documenting their libraries' capabilities and permissions.

2) Application to Other Ecosystems and Programming Langs: Supply chain attacks in the Java ecosystem exploited accidental vulnerabilities [55], [98], unlike attacks in the JavaScript and Python ecosystems which are often caused by malicious vulnerabilities [66]. This emphasizes the need for runtime defenses that protect apps from vulnerable dependencies in addition to preventing the use of malicious libraries.

The Zero-Trust Dependencies concept is language-agnostic and protects applications relying on third-party libraries (§IV). ZTD_{JAVA} instrument functions in a language's core libraries that interact with the operating system's resources (§IV). With minimal adjustments, the tool can be used with other JVM-based languages such as Kotlin [99] and Groovy [100]. Application to other languages is future work.

VIII. CONCLUSION

Software supply chain vulnerabilities and attacks are increasing. Using NIST's Zero-Trust Architecture as a framework, we show that existing security defenses are incomplete. We propose, design, implement, and evaluate the *Zero-Trust Dependencies* approach. Zero-Trust Dependencies places zero implicit trust in third-party software libraries, and can mitigate risks and protect operating system resources without impeding the application's normal behavior. We design a low-cost, low-effort system, ZTD_{SYS}, enabling Zero-Trust Dependencies's application to software. Our prototype for Java applications, ZTD_{JAVA}, effectively mitigates supply chain vulnerabilities with minimal runtime cost and ease of configuration. In summary, Zero-Trust Dependencies provides a robust defense against risks from vulnerable dependencies.

DATA AVAILABILITY

Our artifact is at: https://doi.org/10.5281/zenodo.14436182. It has vulnerability data, ZTD_{JAVA} source, and experiments.

REFERENCES

[1] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, August 1984.

[2] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[3] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *USENIX Security Symposium*, 2019.

[4] Dapeng Yan, Yuqing Niu, Kui Liu, Zhe Liu, Zhiming Liu, and Tegawendé F. Bissyandé. Estimating the attack surface from residual vulnerabilities in open source software supply chain. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 493–502. ISSN: 2693-9177.

[5] Sameed Ali, Prashant Anantharaman, and Sean W. Smith. Armor Within: Defending Against Vulnerabilities in Third-Party Libraries. In *IEEE Security and Privacy Workshops (SPW)*, 2020.

[6] The equifax breach was entirely preventable | WIRED. https://www.wired.com/story/equifax-breach-no-excuse/.

[7] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. Zero Trust Architecture. Technical Report NIST Special Publication (SP) 800-207, National Institute of Standards and Technology, August 2020.

[8] What is Zero Trust? https://cloud.google.com/learn/what-is-zero-trust.

[9] What Is Zero Trust Architecture? | Microsoft Security. https://www.microsoft.com/en-us/security/business/security-101/what-is-zero-trust-architecture.

[10] What is Zero Trust? | IBM. https://www.ibm.com/topics/zero-trust.

[11] Daniel D'Silva and Dayanand D. Ambawade. Building A Zero Trust Architecture Using Kubernetes. In *2021 6th International Conference for Convergence in Technology (I2CT)*, pages 1–8, April 2021.

[12] Elisa Bertino. Zero Trust Architecture: Does It Help? *IEEE Security & Privacy*, 19(05):95–96, September 2021.

[13] Pacharee Phiayura and Songpon Teerakanok. A Comprehensive Framework for Migrating to Zero Trust Architecture. *IEEE Access*, 11:19487–19511, 2023. Conference Name: IEEE Access.

[14] Eduardo B. Fernandez and Andrei Brazhuk. A critical analysis of Zero Trust Architecture (ZTA). *Computer Standards & Interfaces*, 89:103832, April 2024.

[15] SELinux Wiki. http://selinuxproject.org/page/Main_Page.

[16] AppArmor. https://apparmor.net/.

[17] Docker: Accelerated Container Application Development, May 2022. https://www.docker.com/.

[18] The Security Manager (The Java™ Tutorials > Essential Java Classes > The Platform Environment). https://docs.oracle.com/javase/tutorial/essential/environment/security.html.

[19] Willem De Groef, Fabio Massacci, and Frank Piessens. NodeSentry: least-privilege library integration for server-side JavaScript. In *Annual Computer Security Applications Conference (ACSAC)*, December 2014.

[20] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Containing Malicious Package Updates in npm with a Lightweight Permission System. In *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021.

[21] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. Preventing Dynamic Library Compromise on Node.js via RWX-Based Privilege Reduction. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[22] Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C. Rinard. Supply-Chain Vulnerability Elimination via Active Learning and Regeneration. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[23] Pardis Pashakhanloo, Aravind Machiry, et al. PacJam: Securing Dependencies Continuously via Package-Oriented Debloating. In *ACM-Asia Conf. on Computer and Communications Security (AsiaCCS)*, 2022.

[24] Mengtao Sun and Gang Tan. NativeGuard: protecting android applications from third-party native libraries. In *ACM conference on Security and privacy in wireless & mobile networks (WiSec)*, 2014.

[25] Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. Cage4Deno: A Fine-Grained Sandbox for Deno Subprocesses. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2023.

[26] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, Andre DeHon, and Jonathan M. Smith. BreakApp: Automated, Flexible Application Compartmentalization. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2018.

[27] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. FLEXDROID: Enforcing In-App Privilege Separation in Android. In *Proceedings 2016 Network and Distributed System Security Symposium*, San Diego, CA, 2016. Internet Society.

[28] Chinenye Okafor, Taylor R. Schorlemmer, Santiago Torres-Arias, and James C. Davis. SoK: Analysis of Software Supply Chain Security by Establishing Secure Design Properties. In *ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. ACM, 2022.

[29] The Linux Foundation. Supply chain levels for software artifacts(slsa): Safeguarding artifact integrity across any software supply chain. https://slsa.dev/, 2021. Accessed: 2024-03-22.

[30] Ryan Naraine. Software supply chain weakness: Snyk warns of "deliberate sabotage" of npm ecosystem. https://tinyurl.com/4p6x2dwc, March 2022. Accessed: 2024-03-22, url: "https://www.securityweek.com/software-supply-chain-weakness-snyk-warns-deliberate-sabotage-npm-ecosystem/".

[31] Jack Wallen. University of minnesota researchers tried to poison the linux kernel. https://tinyurl.com/5ckrdpv6, April 2021. Accessed: 2024-03-22, url:"https://thenewstack.io/university-of-minnesota-researchers-tried-to-poison-the-linux-kernel-for-a-research-project/".

[32] Zhu Frank. Automatically assess and remediate the solarwinds hack. https://tinyurl.com/4maapt9t, April 2021. Accessed: 2024-03-22, url: "https://jfrog.com/blog/automatically-assess-and-remediate-the-solarwinds-hack/".

[33] Solarwinds. Setting the New Standard in Secure Software Development The SolarWinds Next-Generation Build System. Technical report, solarwinds, December 2021.

[34] Software Bill of Materials (SBOM) | CISA. https://www.cisa.gov/sbom.

[35] OSV - Open Source Vulnerabilities. https://osv.dev/.

[36] Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. Sigstore: Software Signing for Everybody. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[37] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1393–1410, 2019.

[38] Badis Hammi and Sherali Zeadally. Software supply-chain security: Issues and countermeasures. In *Computer*, volume 56, pages 54–66.

[39] Spring cloud function. https://docs.spring.io/spring-cloud-function/docs/current/reference/html/spring-cloud-function.html.

[40] John Kindervag. Build Security Into Your Network's DNA: The Zero Trust Network Architecture. 2010.

[41] Naeem Firdous Syed, Syed W. Shah, Arash Shaghaghi, Adnan Anwar, Zubair Baig, and Robin Doss. Zero Trust Architecture (ZTA): A Comprehensive Survey. *IEEE Access*, 10:57143–57179, 2022.

[42] Keyvan Ramezanpour and Jithin Jagannath. Intelligent zero trust architecture for 5G/6G networks: Principles, challenges, and the role of machine learning in the context of O-RAN. *Computer Networks*, 217:109358, November 2022.

[43] Fabio Federici, Davide Martintoni, and Valerio Senni. A Zero-Trust Architecture for Remote Access in Industrial IoT Infrastructures. *Electronics*, 12(3):566, January 2023.

[44] Seccomp BPF (SECure COMPuting with filters) — The Linux Kernel documentation.

[45] Nik Sultana, Henry Zhu, Ke Zhong, Zhilei Zheng, Ruijie Mao, Digvijaysinh Chauhan, Stephen Carrasquillo, Junyong Zhao, Lei Shi, et al. Towards Practical Application-level Support for Privilege Separation. In *Annual Computer Security Applications Conference (ACSAC)*, 2022.

[46] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: splitting applications into reduced-privilege compartments. In *USENIX Symposium on Networked Systems Design and Impl. (NSDI)*, 2008.

[47] Yongzheng Wu, Jun Sun, Yang Liu, and Jin Song Dong. Automatically partition software into least privilege components using dynamic data dependency analysis. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013.

[48] Marc Ohm, Timo Pohl, and Felix Boes. You Can Run But You Can't Hide: Runtime Protection Against Malicious Package Updates For Node.js, May 2023. arXiv:2305.19760 [cs].

[49] Z. Cliffe Schreuders, Tanya McGill, and Christian Payne. The state of the art of application restrictions and sandboxes: A survey of application-oriented access controls and their shortfalls. *Computers & Security*, 32:219–241, February 2013.

[50] Michael Maass. A Theory and Tools for Applying Sandboxes Effectively. PhD Dissertation. Carnegie Mellon University. 2016.

[51] Sean Mullan. JEP 411: Deprecate the Security Manager for Removal. https://openjdk.org/jeps/411.

[52] P. Ladisa, H. Plate, M. Martinez, and O. Barais. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1509–1526, Los Alamitos, CA, USA, May 2023. IEEE Computer Society.

[53] Github Language Stats. https://madnight.github.io/githut/#/pull_requests/2023/4.

[54] TIOBE Index. https://www.tiobe.com/tiobe-index/.

[55] Cybersecurity and Infrastructure Security Agency. Csrb report on log4j. Technical report, Cybersecurity and Infrastructure Security Agency, July 2022. Accessed: 2024-03-21.

[56] Menghan Xiao. Digging into the numbers: One year after log4shell. https://www.scmagazine.com/feature/digging-into-the-numbers-one-year-after-log4shell, December 2022. Accessed: 2024-03-21.

[57] Amnon Even-Zohar. Eslint: Compromising the build using supply chain attack. https://cycode.com/blog/eslint-compromising-the-build-using-supply-chain-attack/, 2021. Accessed: 2024-03-21.

[58] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. Empirical analysis of security vulnerabilities in python packages. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 446–457, 2021.

[59] Lyuye Zhang, Chengwei Liu, Sen Chen, Zhengzi Xu, Lingling Fan, Lida Zhao, Yiran Zhang, and Yang Liu. Mitigating persistence of open-source vulnerabilities in maven ecosystem, 2023.

[60] Berkay Kaplan and Jingyu Qian. A survey on common threats in npm and pypi registries. In Gang Wang, Arridhana Ciptadi, and Ali Ahmadzadeh, editors, *Deployable Machine Learning for Security Defense*, pages 132–156, Cham, 2021. Springer International Publishing.

[61] Amir M. Mir, Mehdi Keshani, and Sebastian Proksch. On the effect of transitivity and granularity on vulnerability propagation in the maven ecosystem. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 201–211, 2023.

[62] Open Data Releases | Ecosyste.ms: Packages. https://packages.ecosyste.ms/open-data.

[63] Paschal C. Amusuo, Ricardo Andrés Calvo Méndez, Zhongwei Xu, Aravind Machiry, and James C. Davis. Systematically Detecting Packet Validation Vulnerabilities in Embedded Network Stacks. In *IEEE/ACM International Conference on Automated Software Engineering (ASE 2023)*, August 2023. arXiv:2308.10965 [cs].

[64] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–88, June 2012.

[65] Wafaa Al-Kahla, Ahmed S. Shatnawi, and Eyad Taqieddin. A Taxonomy of Web Security Vulnerabilities. In *2021 12th International Conference on Information and Communication Systems (ICICS)*, pages 424–429, May 2021. ISSN: 2573-3346.

[66] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43, Cham, 2020. Springer International Publishing.

[67] CodeQL. https://codeql.github.com/.

[68] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, et al. The DaCapo benchmarks: java benchmarking development and analysis. *ACM SIGPLAN Notices*, 41(10), 2006.

[69] CWE-502: Deserialization of Untrusted Data (4.14). https://cwe.mitre.org/data/definitions/502.html.

[70] CWE-863: Incorrect Authorization (4.14). https://cwe.mitre.org/data/definitions/863.html.

[71] CWE-770: Allocation of Resources Without Limits or Throttling (4.14). https://cwe.mitre.org/data/definitions/770.html.

[72] CWE-94: Improper Control of Generation of Code ('Code Injection') (4.14). https://cwe.mitre.org/data/definitions/94.html.

[73] CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') (4.14). https://cwe.mitre.org/data/definitions/22.html.

[74] CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') (4.14). https://cwe.mitre.org/data/definitions/79.html.

[75] CWE-611: Improper Restriction of XML External Entity Reference (4.14). https://cwe.mitre.org/data/definitions/611.html.

[76] CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') (4.14). https://cwe.mitre.org/data/definitions/78.html.

[77] CWE-248: Uncaught Exception (4.14). https://cwe.mitre.org/data/definitions/248.html.

[78] Hossein Zare, Mojgan Azadi, and Peter Olsen. Techniques for detecting and preventing denial of service attacks (a systematic review approach). In Shahram Latifi, editor, *Information Technology - New Generations*, pages 151–157. Springer International Publishing.

[79] Spyridon Samonas and David Coss. The CIA strikes back: Redefining confidentiality, integrity and availability in security. *Journal of Information Systems Security*, 10(3):21–45.

[80] R.S. Sandhu and P. Samarati. Access control: principle and practice. *IEEE Communications Magazine*, 32(9):40–48, September 1994. Conference Name: IEEE Communications Magazine.

[81] Oracle Corporation. Permissions in the java development kit, 2014. https://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.html.

[82] java.lang.instrument (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html.

[83] ASM. https://asm.ow2.io/.

[84] Patricia tree. https://xlinux.nist.gov/dads/HTML/patriciatree.html.

[85] Christopher Brant, Prakash Shrestha, Benjamin Mixon-Baca, Kejun Chen, Said Varlioglu, Nelly Elsayed, Yier Jin, Jedidiah Crandall, and Daniela Oliveira. Challenges and Opportunities for Practical and Effective Dynamic Information Flow Tracking. *ACM Computing Surveys*, 55(1):17:1–17:33, November 2021.

[86] Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. Mining sandboxes. In *International Conference on Software Engineering (ICSE)*, 2016.

[87] Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. Mining Sandboxes for Linux Containers. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

[88] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating Seccomp Filter Generation for Linux Applications. In *Cloud Computing Security Workshop (CCSW)*, 2021.

[89] Matthew W Sanders and Chuan Yue. Mining least privilege attribute based access control policies. In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[90] OpenJDK: jmh. https://openjdk.org/projects/code-tools/jmh/.

[91] Diego Costa, Cor-Paul Bezemer, Philipp Leitner, and Artur Andrzejak. What's Wrong with My Benchmark Results? Studying Bad Practices in JMH Benchmarks. *IEEE Transactions on Software Engineering (TSE)*, 47(7), 2021.

[92] Equifax data breach FAQ. https://www.csoonline.com/article/567833.

[93] SAS response/recommendations for zero-day log4j2 CVE-2021-44228 vulnerabilities. https://communities.sas.com/t5/Administration-and-Deployment/SAS-response-recommendations-for-zero-day-log4j2-CVE-2021-44228/td-p/785489.

[94] Mitchell Olsthoorn, Dimitri Stallenberg, Arie van Deursen, and Annibale Panichella. SynTest-solidity: automated test case generation and fuzzing for smart contracts. In *International Conference on Software Engineering: Companion*, pages 202–206. ACM.

[95] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. PMFuzz: test case generation for persistent memory programs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 487–502.

[96] Yuwei Liu, Yanhao Wang, Purui Su, Yuanping Yu, and Xiangkun Jia. InstruGuard: Find and fix instrumentation errors for coverage-based greybox fuzzing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[97] Open Source Insights. https://deps.dev/.

[98] Equifax confirms Apache Struts flaw it failed to patch was to blame for data breach. https://tinyurl.com/mr38cya7.

[99] Kotlin programming language. https://kotlinlang.org/.

[100] The apache groovy programming language. https://groovy-lang.org/.