# Back to the future: N-Versioning of Microservices

Antonio M. Espinoza*, Riley Wood †, Stephanie Forrest*, Mohit Tiwari†
*Biodesign Center for Biocomputing, Security and Society
Arizona State University, Tempe, AZ
†Dep. of Electrical and Computer Engineering
The University of Texas at Austin, Austin, TX
Email: {amespi22, steph}@asu.edu, riley.wood@utexas.edu, tiwari@austin.utexas.edu

***Abstract*—Microservices are the dominant architecture used to build internet-scale applications today. Being internet-facing, their most critical attack surfaces are the OWASP top 10 Web Application Security Risks. Many of the top 10 OWASP attack types—injection, cross site scripting, broken access control and security misconfigurations—have persisted for many years despite major investments in code analysis and secure development patterns. Because microservices decompose monolithic applications into components using clean APIs, they lend themselves to practical application of a classic security/resilience principle, $N$-versioning. The paper introduces RDDR, a principled approach for applying N-versioning to microservices to improve resilience to data leaks. RDDR applies N-versioning to vulnerable microservices, requiring minimal code changes and with low performance impact beyond the cost of replicating microservices. Our evaluation demonstrates RDDR mitigating vulnerabilities of the top 5 of the top 10 OWASP types by applying diversity and redundancy to individual microservices.**

## I. Introduction

N-versioning is a classic technique for increasing the resiliency of programs to various defects by running multiple versions of a program simultaneously and comparing the outputs [19]. Unlike popular diversity methods such as Address Space Layout Randomization (ASLR), N-versioning can protect against logic errors because different versions may use different program logic. However, N-versioning incurs the memory and computation overhead of running N instances, which can be prohibitive for modern, large-scale software. Moreover, N-versioning has historically been applied only to monolithic architectures [12], [22], [24] or at the binary level with limited practical adoption [17]. For large-scale systems, these monolithic architectures have largely been supplanted by microservice architectures that use containerization.

Microservice architectures are the backbone of today's most popular websites/applications. For example, Lyft, Netflix, Facebook, and Spotify all use microservice architectures, each comprised of hundreds to thousands of microservices. Each microservice runs in a container [29] and performs a simple task (or service) provided through an API that is available to the system, referred to as a *deployment*. This distributed architecture supports large applications by making software modular and scalable (horizontally and vertically). In this paper, we leverage the API, modularity, and scalability of microservices to strengthen their security through N-versioning in a system called RDDR. The API provides tractable insertion points for deploying RDDR as a proxy for particular microservices, while the modularity and scalability allow the seamless integration of RDDR with the deployment.

RDDR is a system that: **R**eplicates a request to N instances of a microservice, **D**e-noises non-deterministic behavior, identifies **D**ifferences in the instances' responses, and **R**esponds appropriately. We evaluate RDDR on ten indicative Common Weakness Enumerations (CWEs), finding that RDDR mitigates each of them successfully. These CWEs collectively cover vulnerabilities from the top five of the most important Open Web Application Security Project (OWASP) vulnerability classes [54]—the OWASP Top Ten list is comprised of the most common and important vulnerabilities faced by web applications.

Although we cannot eliminate the overhead incurred by N-versioning an individual microservice, we can reduce the overall overhead by N-versioning only a subset of the microservices in a deployment. Many exploits and bugs manifest as manipulated data, visible when exiting a microservice. By targeting the microservices that handle unmodified user data (*e.g.*, API servers, parsers, and input sanitizers), the number of N-versioned microservices can be minimized.

To the best of our knowledge, RDDR is the first practical implementation of an N-versioning defense for distributed cloud applications. The orchestration layer provided by microservices creates the opportunity to bring classic security concepts like automated detection-response and information flow tracking into production environments. RDDR provides a straightforward implementation path for N-versioned systems for platforms that use container orchestration frameworks, such as Kubernetes [52].

The paper's main contributions are:

- RDDR, a novel N-versioning architecture for microservices, and techniques addressing design/implementation challenges: orchestrating diversity in the microservice-architecture, maintaining consistency, coping with non-determinism and ephemeral state, managing variance such as version numbers, and handling communication protocols. (§IV)
- An open-source implementation of RDDR for Kubernetes, a popular container orchestration platform, which is available for broader adoption [63], [64] (§V).
- An evaluation of RDDR on the most relevant Top 10 OWASP Web Application Security Risks using ten real-

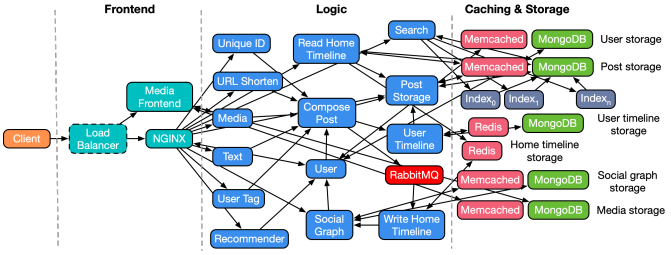



Fig. 1: A small-scale social network deployment by Gan *et al.* [31]. Part of the DeathstarBench open-source benchmark suite.

world CWEs. We find that RDDR mitigates each tested CWE with performance overhead that is near-linear in the number of redundant microservices. (§V-G)

## II. Motivation

To motivate RDDR, we consider two examples of web application scenarios that would benefit from N-versioning.

In the first scenario, suppose that a security bug is discovered in a production web application. The development team quickly releases a patch to mitigate the bug. Unfortunately, the patch introduces a new bug, an event commonly observed in practice. For example, Crameri *et al.* investigated impediments to deploying updates in real systems [25], and found that bugs in updates are one of the most common reasons that software updates fail. Similarly, Yin *et al.* [67] estimate that between 14.8% and 24.4% of OS updates are implemented incorrectly. Using RDDR, these update-based vulnerabilities could be mitigated by running the old and new versions in parallel while checking for consistency to verify correct behavior. *Correct* in this context means that both versions produce the same output for a given input. This strategy reduces the attack surface to just those vulnerabilities shared by both instances.

In the second scenario, developers proactively minimize the attack surface by deploying multiple implementations of the same function. For example, if a deployment uses both HAProxy and nginx as a reverse proxy, it can avoid threats such as CVE-2019-18277 which allows request smuggling [36] in HAProxy. Many other application components have diverse, compatible implementations that are available, *e.g.*, Dropbear and OpenSSH for SSH; PostgresSQL, CockroachDB, and EnterpriseDB for SQL; and Apache and nginx for HTTP servers. Similarly, equivalent libraries could be used as components to derive diversity in applications, such as pairs of Python libraries: `rsa` and `Crytpo`, or `markdown` and `markdown2`. RDDR can easily accommodate such cases by running the implementations in parallel to detect divergent/-malicious behaviour.

These scenarios illustrate how microservice architectures can benefit from N-versioning. However, the classical approach of N-versioning the entire system is prohibitive. Even if diversity were applied to a single microservice, one would need to create N instances of the entire deployment. RDDR provides a solution by supporting N-versioned deployments for only the subset of microservices most likely to suffer failures or attacks. Consider the microservice application depicted in Figure 1. If all microservice containers in Figure 1 were equally costly, then N-versioning the "Search" and "Compose Post" services (using the techniques described in scenarios one and two respectively) with RDDR would incur an overhead of $\sim 20\%$ instead of $300\%$ (assuming a 3-versioned system).

Although Knight *et al.* [43] showed long ago that this form of implementation diversity does not always generate completely independent failure modes, it can significantly mitigate many bugs and vulnerabilities. Our tests of RDDR using different libraries and database implementations confirm that, in many cases, independent implementations do diverge during exploitation (*e.g.*, Table I, rows 1 and 4-8).

## III. Related Work

Several areas of related work are relevant to RDDR: Service Oriented Architectures, N-versioning, Moving Target Defense (MTD), and deployment strategies. The Service-Oriented Architecture (SOA) community typically focuses on improving quality of service (*e.g.*, [10], [27], [35]), while N-versioning aims to improve software robustness (*e.g.*, [33], [49], [68]) and MTD focuses on reducing attack surfaces [14], [39], [56]. We share MTD's goal of improving security through diversity, but our approach borrows from N-versioning, and we use the distributed architecture developed in the SOA community to create a novel N-versioned system that runs in real time in a containerized environment via Docker.

### A. Service Oriented Architecture (SOA)

Distributed applications composed of microservices are the most relevant subset of SOAs. Theoretical models have been proposed that leverage containerization for redundancy and diversity. Gorbenko *et al.* describe "reliable concurrent execution" [35], which is similar to RDDR's front-end proxy. As a theoretical model, however, this work avoids many significant implementation challenges (§ IV-B).

Diffy, a regression testing tool developed by Twitter, also resembles RDDR as it runs two versions of a single microservice (old and new) in parallel [58]. Although RDDR adopted ideas from Diffy to support seamless integration into distributed application architectures, there are important differences because Diffy was developed for finding bugs and is not intended for production use. Although Diffy replicates traffic to the microservice replicas, it does not merge requests to downstream microservices. RDDR addresses this issue with an outgoing proxy to merge traffic streams. Similarly, RDDR's non-deterministic noise filter was inspired by Diffy but extends it to handle ephemeral state such as Cross-Site Request Forgery (CSRF) tokens. This extension allows RDDR to handle the many real-world applications that rely on CSRF tokens for security by automatically identifying, saving, and later restoring ephemeral state output by a microservice.

### B. N-versioning

The central idea of N-versioning entails executing multiple copies of a program/system (usually diverse) in parallel and

checking outputs for consistency. Early examples include systems that run redundant components in hardware to produce reliable computation in computers [46] and aviation [62], [65]. When divergence is detected N-versioned systems typically vote to decide which output to accept. Software implementations of N-versioning—also known as Multi-Variant and N-Variant Execution— typically halt execution when divergence is detected, either restarting the system from scratch or from a "refresh state" before continued computation [22], [24], [60], [61].

Gholami *et al.* used N-versioning to reduce server load and improve performance of distributed applications [33]. They varied service instances by changing the number of available features, with light-weight instances handling requests that do not require the full feature set. Their defense is similar to a MTD (not redundant) and because their architecture focuses on efficiency, their diversity methods are unlikely to provide a security benefit.

Otterstad *et al.* describe a theoretical system for deploying diverse microservices behind a controller [49]. Their proposal resembles RDDR's use of N-versioned microservices with divergence triggering blocked responses. Because this system is a theoretical model, it does not address the challenges faced by real-world implementations discussed in Section I.

Österlund *et al.* propose N-versioning (called multi-variant execution) to improve robustness of the Linux kernel [68]. Similar to RDDR, this work aims to protect against information leaks by deploying two diverse kernels in parallel. They use a *variant generator* to manually create variations among the kernels, which may diverge in the presence of information leaks. Our approach relies on diversity that is derived organically, by multiple teams implementing the same software specification in isolation [13], [41], [42], from different versions of software, and automatically from the OS.

RDDR's two version approach is similar to that of Hosek *et al.* [37] who studied a similar deployment strategy in non-microservice environments. Their method compares system call traces of the two programs, which is both computationally expensive, and can lead to false positives, *e.g.*, when refactored code is labeled as "divergent" even if the output of the programs is identical on all inputs. RDDR does not attempt to determine which version is correct, treating all divergence as problematic, since divergence could indicate a bug or vulnerability in either version.

### C. Moving Target Defense (MTD)

MTD systems diversify applications over time rather than running multiple variants concurrently. Some MTD systems consider distributed server applications [14], [39], [56], and Torkura *et al.* use MTD to randomize the attack surface of cloud applications [56]. Their approach is data-driven and prioritizes high-risk vulnerabilities by aggregating known microservice vulnerabilities and comparing their CVSS scores [7] and OWASP ratings [2]. By paying the price of N-versioning we are able to leverage the protection provided by all versions simultaneously, unlike an MTD system which only reaps the benefits of the version it is currently running (*i.e.*, an attacker can persist until the MTD system is running a version it can exploit).

Some MTD diversity strategies can be adopted in the RDDR framework, such as that proposed by Taguinod *et al* [55], which uses two different methods to create diversity. The first, translates PHP applications to Python automatically. The second takes advantage of the translation layer between MySQL and PostgreSQL databases, allowing them to switch the backend database periodically to prevent attacks. The translated application could be run (in parallel with the original) by RDDR and the database translation layer could be imported using RDDR's protocol support module described in Section IV-B1. Diversity created in this way is powerful, but requires the developer to expend resources creating translations from one application to another. We elected to use readily available diversity (§ IV-C) to alleviate the burden on developers.

Although some strategies for generating MTD application diversity can be adopted by RDDR, MTD *network shifting* defenses cannot. These defenses include port hopping and network address shifting [50]. Network-based defenses are not applicable to RDDR's threat model because they defend against the reconnaissance phase of network attacks by periodically shifting network configurations. RDDR assumes an adversary that is exploiting an application through a front-facing interface that is always available, even in the face of shifting network configurations. Further, the Kubernetes environment typically has a single publicly accessible IP address, with inner containers using unrouteable IP addresses that are inaccessible from outside the deployment.

### D. Other Deployment Strategies

Proxies such as Envoy [20] and nginx [48] offer a feature called "traffic shadowing." In traffic shadowing, traffic to a microservice in a production deployment is replicated and forwarded to a "shadow" copy, enabling developers to perform regression tests on real-world traffic. However, the proxy ignores all traffic returned by the shadowed instance and does not actively protect the microservice. By contrast, RDDR sits on the critical path of traffic and monitors communication to and from the protected microservices. Unlike a shadow proxy, RDDR also copes with applications that exhibit random noise or instance-specific state.

## IV. RDDR Design and Implementation

This section describes our threat model, presents our design, and recounts implementation details.

### A. Threat Model

We assume that the attacker's goal is to leak data from a distributed application by exploiting one or more vulnerabilities in the application's constituent microservices, *e.g.*, using cross-site scripting in components with known vulnerabilities. We assume that the attacker extracts data through one of the microservice endpoints, rather than a side channel. A

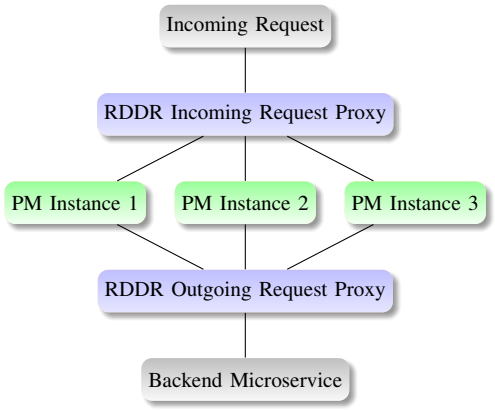



Fig. 2: RDDR schematic: PM denotes **P**rotected **M**icroservice. Colors represent: external traffic (gray), RDDR (blue), and the replicated protected service (green).

vulnerability is considered mitigated if the information leak is detected and blocked. Lastly, the attacker is assumed to be unprivileged in the context of host system administration and in the context of the web application. Although we exclude side channel leakage, prior work addresses such leakage[s] (*e.g.*, [11], [15], [47]).

RDDR intercepts attacks that cause insecure data to be returned or passed between microservices in the deployment, which allows RDDR to detect relevant attacks from five of the Top Ten OWASP Application Security Risks [54]: broken access control, cryptographic failures, injection, insecure design, and security misconfiguration (A1-A5). The majority of attacks in OWASP A6-A10 do not explicitly cause data leaks and thus are not within our threat model.

### *B. RDDR*

There are four main phases of a RDDR proxy for a protected microservice: *R*eplicate, *D*e-noise, *D*iff, and *R*espond. Traffic is first **replicated** by the incoming request proxy and sent to each instance. The response from the instances are **de-noised** to eliminate known variance, before RDDR **diffs** the responses to detect divergent behavior. Finally, RDDR **responds** to the client. The response is either the unanimously response from the N versions or, in the event of divergence, a web page indicating that RDDR intervened. If the protected microservice requires communication with a microservice inside the deployment, all outgoing requests are verified for consistency by an outgoing request proxy.

RDDR enhances application robustness with low overhead by exploiting the distributed nature of cloud applications. By creating multiple instances of only critical microservices, RDDR avoids the cost of diversifying the entire application, which can be significant [21]. Modern container orchestration tools such as Kubernetes [52] and Docker Swarm [9] simplify the process of replicating microservices from a base image. This pre-existing infrastructure, and the fact that RDDR was designed to be run as a container using a Docker image, make RDDR straightforward to deploy.

RDDR is written in Python 3.8 with source code freely available online under the MIT License. Architecturally, RDDR can be visualized as a set of proxies which sit on either side of the N instances of the protected microservice (Figure 2). Both proxies operate at the transport/socket layer, bind to an IP and one or more ports to await incoming connections.

The **RDDR Incoming Request Proxy** handles request traffic sent to the protected microservices. Both the incoming requests and their responses pass through the Incoming Request Proxy which maintains the state required to handle SSL/TLS connections. The proxy replicates the request, makes any necessary modifications (§ IV-B3), then forwards it to all instances. The Incoming Request Proxy compares the response from each instance for consistency. Traffic is tokenized and compared for divergence according to the application layer protocol (*e.g.*, HTTP, SQL, *etc.*). If divergence is detected, excluding non-deterministic noise (§ IV-B2), the proxy closes the connection to the client and halts communication. This approach avoids information leaks or other problems but disrupts only the diverging instances.

Deployed "behind" the application instances are zero or more **RDDR Outgoing Request Proxies**, depending on the number of microservices the protected microservice communicates with. This proxy monitors traffic flowing between the instances and another microservice in the deployment, with one proxy assigned for each distinct microservice. The proxy detects information leaks on connections *initiated* by the protected service. The Outgoing Request Proxy is a dual of the Incoming Request Proxy, monitoring messages produced by the N instances for consistency.

*1) Protocol Support:* RDDR supports multiple transport and application layer protocols. It currently supports unencrypted TCP and encrypted SSL/TLS (via Python's SSL library) at the transport layer. It also supports some application layer protocols, including PostgreSQL, HTTP, and JSON. Support for application layer protocols is implemented by Python modules that comply with a standard interface, allowing developers to extend RDDR to support other protocols. These modules handle all protocol-specific tasks such as tokenizing, differencing traffic, and traffic modification.

The application protocol modules tokenize responses from the instances and monitor for divergence according to the semantics of the protocol. For example, the HTTP module tokenizes at the newline boundary and compares lines. If necessary, it also interprets the HTTP header and decompresses the message before differencing, and it saves CSRF tokens (§IV-B3). The PostgreSQL module tokenizes traffic into separate messages according to the PostgreSQL message format [1] and differences messages of known critical types.

*2) Handling Nondeterminism:* A crucial feature missing from earlier theoretical models is a method for distinguishing random noise from relevant divergent behavior. This occurs any time the N microservice instances send a random string to the client, *e.g.*, when the instances send PHP session IDs that web applications use to uniquely identify users. Because

each of the N instances generates a different random string, traffic will diverge even in the absence of a bug. RDDR addresses this similarly to Diffy [58] by deploying two identical instances, the *filter pair*, of the microservice (no diversity applied) and using them to determine which divergences are problematic and which arise from nondeterminism. RDDR's filtering assumes any divergences in the filter pair are benign. With filtering enabled, RDDR identifies a divergence if any instances *except* the "filter pair" produce non-identical output. The filter pair must detect any non-deterministic behavior to avoid false-positive divergences. Therefore, we assume a cryptographically-secure source of randomness to avoid duplicate values.

*3) Handling Ephemeral State:* Client-server handshakes that use ephemeral microservice state complicate RDDR N-versioning. For example, CSRF tokens are strings randomly generated by the server and embedded in HTML forms requested by clients to prevent CSRF attacks. Such tokens protect against a user who is tricked into submitting POST data to the server via a malicious link. Without special handling, such tokens would appear to the N-version monitor as non-deterministic noise, triggering a divergence. RDDR addresses this by modifying the message sent to each instance to ensure that each instance receives the correct token.

Specifically, RDDR's HTTP handler plugin stores the values that it identifies as CSRF tokens and reinserts them when needed. The plugin scans the traffic sent to the client, noting lines that differ across all instances. Within those lines, it looks for the character ranges that differ, and if they are alphanumeric and at least ten characters long, the plugin saves them. The criteria for saving—alphanumeric and at least ten characters—was determined empirically and correctly identified all CSRF tokens in our evaluation. The plugin maintains a mapping between tokens and microservices. It then substitutes the correct token for each microservice before forwarding. Because they are ephemeral, tokens are deleted after forwarding. In our current system, only the HTTP extension implements this feature.

*4) Handling Known Variance:* Although RDDR anticipates many sources of benign divergence and handles them automatically, some are out of scope for RDDR. For example, if different versions of the same application are deployed behind RDDR, and a user requests the software version, RDDR will identify deterministic divergence. To mitigate this and similar issues, we support *manual* configuration of RDDR to ignore application-specific benign divergence (through RDDR's configuration file). This is currently implemented for the PostgreSQL plugin.

### C. Acquiring Program Variants

Because RDDR requires unanimous agreement among the N microservice variants, the attack surface of the system is the intersection of the attack surfaces of all instances. The success of this strategy depends on the diversity and independence of the constituent microservice instances, which must also preserve desired behavior. There are several sources of effective diversity in this setting, including: independent implementations, different versions of the same implementation, and system/process level diversity. The RDDR architecture supports each of these.

All of our diversity methods assume the threat model described in Section IV-A, except version diversity which also assumes that there are no feature changes between versions, *i.e.*, the only code edits are those that affect patch vulnerabilities. Consequently, if the protected microservice is undergoing rapid functionality changes (*e.g.*, adding API calls), we suggest that developers adopt another source, such as library diversity. We tested a wide variety of diversity sources in our experiments to illustrate RDDR's flexibility, but we do not assign them a hierarchical ordering of importance. For general applications such as proxies and databases, we recommend adopting diverse implementations (§ V-C); and for highly specialized containers, we recommend diverse libraries (§ V-A). As mentioned above, if the only updates to the mircroservice are those that address vulnerabilities, then we recommend version diversity (§ V-D).

**Diverse implementations** of a particular microservice function may be available from different vendors. By 'function,' we mean a logical component of a system such as a database or proxy that performs a specific task. Databases like PostgreSQL, *e.g.*, implement a well-specified query language and network protocol, as do CockroachDB [23] and EnterpriseDB [28]. Because these three applications use the same network protocol and have the same functionality, it is straightforward to deploy them as diverse instances of the same logical database.

A related approach uses **different versions** of an application from a shared codebase. This technique can help close vulnerabilities in an older version of the microservice while simultaneously preventing exploitation of new bugs that might be introduced by a software update. We assume that each version shares at least the set of features required by the larger application. This technique was explored by Hosek *et al.* in [37], who found that new vulnerabilities introduced by software updates could be successfully mitigated by deploying them in parallel with older versions. Malicious inputs may trigger divergence in the patched code causing RDDR to intervene, however, the desired effect of stopping information leakage is obtained.

There are also automated ways of generating diversity that do not incur software development overhead (*e.g.*, Forrest *et al.* [16] and Cox *et al.* [24]). The OS can diversify software at the process level by randomizing the address space, inserting randomized stack guards, and using disjoint code layouts. Prior works such as [32], [44], [59] have explored N-versioning implementations relying on these techniques. Developers can also leverage compilers to introduce diversity. Larsen *et al.* [45] provide a useful overview of state-of-the-art techniques for automating software diversity, including mutating and reordering basic blocks, randomizing the stack layout, and inserting garbage code, among many other techniques. Rebaudengo *et al.* [51] use compiler-based diversification to increase soft-

ware robustness to transient hardware errors. AVATARs, a project from NEC Laboratories, uses a combination of diverse compilers and source-to-source translators to yield diverse executables from a single codebase [53]. These methods can be readily accommodated by RDDR. We illustrate this in Section V-E with ASLR derived diversity.

### D. Limitations

RDDR is vulnerable to Denial-of-Service (DoS) attacks. Consider the case where three different versions of an application are deployed and one has a bug that causes runaway CPU utilization creating a DoS. The N-versioned system will be vulnerable to this DoS unless it causes a divergence. While out of scope for this paper, this DoS issue could be mitigated with a timeout counter. Other DoS attacks can be mitigated by using automated signature generation (*e.g.*, [40]) to defeat an attacker who repetitively triggers divergence by entering the diverging input repeatedly.

RDDR potentially broadens the timing attack surface. Consider application A deployed alongside application B. A contains no timing bugs, whereas B contains one timing bug that can leak sensitive information: Suppose B takes longer to respond to a login request with a valid username than one with an invalid username. If A and B are deployed together and RDDR waits for them both to respond, the timing channel in B remains exploitable. Prior work from Yin *et al.* explores how to deter attackers from exploiting the weakest instance in an N-versioned deployment to learn information via a side channel [66].

N-versioning is not applicable to services that generate instance-specific secrets that expect a unique user response, such as multi-factor authentication. If a microservice protected with RDDR does generate secrets (*i.e.*, unique values) that flow directly or indirectly to the client, then the de-noising step would cause RDDR to deny all traffic. For example, if one were to 2-version a multi-factor authentication microservice, each version would expect a different responses from the user. Because the user would only be supplying the correct response to one of the microservices, the other would fail to authenticate causing RDDR to intervene in the connection.

Microservices containing time-varying information in their outputs could generate a false positive in RDDR. Consider a microservice that reports a coarse-grained timestamp. If a request is made on a time boundary to a two-version deployment with non-deterministic filtering, it is possible that the filter pair both receive timestamp $t$ whereas the third instance receives time $t + 1$ generating a false alarm. This issue is easily addressed using the method discussed in Section IV-B4 for handling known variance, although we have not found it to be an issue in testing.

## V. Evaluation

We evaluate RDDR's performance using several different deployments, diversity generation methods, and vulnerability types, summarized in (Table I). Each deployment was hosted on real hardware, and the relevant Kubernetes files are available [63], [64]. Our evaluation covers five OWASP categories and three kinds of diversity: independent implementations, multiple software versions, and automatically-generated diversity (address space randomization). Subsections V-A through V-E describe our setups, the vulnerabilities, and results. We also consider a large-scale distributed application, GitLab (§ V-F), which incorporates many industry best practices for security. Finally, we conduct a detailed performance evaluation to study the overhead incurred by N-versioning with RDDR (§ V-G).

### A. Library diversity in RESTful APIs

We use the term 'RESTful' to refer to APIs that call functions on stateless servers with deterministic output. We evaluated four different CVEs involving RESTful architectures: 2020-13757, 2020-11888, 2020-10799, and 2014-3146 (Table I). We generated diversity using two implementations of otherwise identical libraries, and RDDR mitigated each of the four CVEs. In each test case we compared two instances: the vulnerable library corresponding to the CVE and a library with similar functionality but a different code base.

CVE-2020-13757 was mitigated using the `rsa` and `crypto` Python libraries as variants to decrypt RSA encrypted data. CVE-2020-11888 was mitigated using the `markdown2` and `markdown` Python libraries as variants to sanitize user input markdown. CVE-2020-10799 was mitigated using the `svglib` and `cairosvg` Python libraries as variants to transform svg files into .png files. Finally, CVE-2014-3146 was mitigated using the `lxml` Python and the `sanitize-html` Node.js libraries to sanitize user input XML. To create RESTful servers with access to Python libraries, the function calls were accessed using flask servers, and for the Node.js sever we used the `http` library.

These examples demonstrate that RDDR is compatible with RESTful architectures, that diversity can be created by running servers with identical APIs but different libraries, and that generating diversity using libraries written in different languages is effective against this class of CVEs.

### B. SQL Injection

The Damn Vulnerable Web App (DVWA) is a website that illustrates many common web vulnerabilities [26]. It is an educational tool that can be configured for different security levels. DVWA contains an SQL injection in which an attacker modifies a benign query to inject malicious queries. A common defense is query sanitization, which detects characters of the user's input, such as apostrophes and quotation marks, and prevents them from being interpreted as SQL syntax. Different DVWA security levels sanitize user input to varying degrees.

We used RDDR to harden DVWA against SQL injections, which tested the outgoing request proxy and its ability to handle ephemeral state. We modified DVWA slightly to use an external database and deployed three instances of the DVWA frontend, which communicated with a single backend database through RDDR's outgoing proxy (*e.g.*, Figure 2 illustrates this, with the DVWA protected microservices labeled 1-3 and

| CVE | Microservice/program | Exploit | CWE | Mitigated | OWASP # | Diversity |
|---|---|---|---|---|---|---|
| 2017-7484 | PostgreSQL | Exposure of sensitive information to an unauthorized actor | 200,285 | ✓ | 1 | Identical API, different program. |
| 2017-7529 | Nginx | Integer overflow | 190 | ✓ | N/A | Version number. |
| 2019-10130 | PostgreSQL | Improper access control | 284 | ✓ | 1 | Version number. |
| 2019-18277 | HAProxy | HTTP Request Smuggling | 444 | ✓ | 4 | Multi-program. |
| 2014-3146 | lxml lib/RESTful | Cross site scripting | Other | ✓ | 3* | Library in different language. |
| 2020-10799 | svglib lib/RESTful | Improper restriction of XML external entity reference | 611 | ✓ | 5 | Compatible libraries. |
| 2020-13757 | rsa lib/RESTful | Use of risky crypto | 327 | ✓ | 2 | Compatible libraries. |
| 2020-11888 | markdown2 lib/RESTful | Cross site scripting | 79 | ✓ | 3 | Compatible libraries. |
| N/A | DVWA | SQL injection | 89* | ✓ | 3 | Multi-programming. |
| N/A | ASLR POC | Heap overflow | 122* | ✓ | N/A | Random memory layout. |

TABLE I: RDDR vulnerability mitigations. Items marked with an asterisk are unofficial *i.e.*, assigned by the authors and not a governing body such as NIST. CVE-2017-7484 has two CWEs, one assigned by NIST the other by Redhat. For CVE-2014-3146, the CWE assignment of "Other" was designated by NIST. Common Weakness Enumerations (CWEs) were assigned to OWASP numbers using the official mapping [38].

the database labeled 'backend microservice'). The DVWA instances were configured with different levels of security and sanitized user input differently. One instance was configured for high input sanitization, and the other two instances, forming the filter pair, performed no input sanitization. As expected, RDDR diverged when the SQL attack was launched. Although slightly contrived, this example emphasizes the role of the outgoing proxy, which enabled the N instances to communicate with a single backend database and was instrumental in detecting the divergence. It also highlights the way RDDR handles known ephemeral state of web applications.

RDDR prevents the SQL injection, while processing benign traffic normally. When the user requests the SQL Injection demo page, the request is forwarded by RDDR's incoming proxy to every instance of DVWA's frontend. Each instance returns the web page containing an input form and CSRF token. As described in Section IV-B3, RDDR automatically identifies the CSRF tokens and saves them, forwarding the page sent by the first instance.

### C. Diverse microservice implementations

In some cases, multiple vendors offer products that comply with the same interface but have different implementations. To be applicable, each implementation must implement the same application layer protocol (*e.g.*, HTTP), or a translation layer must be available so all benign traffic looks identical across all instances. Here we examine two CVEs and diverse microservice implementations as the source of diversity.

*1) Reverse Proxies:* In the first example, RDDR protects a reverse proxy, and we document the work required to add RDDR to an existing deployment. RDDR successfully mitigates CVE-2019-18277 [8], a vulnerability in HAProxy (version 1.5.3) that allows HTTP request smuggling [36], by using nginx as a diverse implementation of a reverse proxy. To test the CVE we created a simple service (S1) which contains an API call that should not be invoked directly from outside the deployment. To enforce this requirement both HAProxy and nginx were configured to deny the API call when proxying, When the client sends a malicious request string, the request is not filtered by HAProxy and passed on to S1. Nginx however, is not susceptible to request smuggling and does not pass the request to S1. This causes a divergence in the values returned by the proxy's calls to S1, and RDDR prevents the response from reaching the client. This exploit causes two divergences—in the information returned to the client and in data sent from the protected microservice. Thus, the exploit could also have been mitigated by monitoring microservice output via the outgoing request proxy.

The addition of nginx to the deployment was straightforward, requiring a total of 174 lines of configuration split between six configuration files, many of which were slight modifications of easily available default configuration files. The total time to add RDDR to the deployment was approximately one hour.

*2) Databases:* CockroachDB and PostgreSQL are two examples of databases that can be deployed together behind RDDR without a translation layer. We used two Postgres instances and one CockroachDB instance to mitigate CVE-2017-7484, an information leak in Postgres versions up to 9.2.20 [4].

```
CREATE FUNCTION leak2(integer,integer) RETURNS
    boolean
    AS $$BEGIN RAISE NOTICE 'leak % %', $1, $2;
        RETURN $1 > $2; END$$
    LANGUAGE plpgsql immutable;
CREATE OPERATOR >>> (procedure=leak2, leftarg=integer
    , rightarg=integer, restrict=scalargtsel);
SET client_min_messages TO 'notice';
EXPLAIN (COSTS OFF) SELECT * FROM some_table WHERE
    col_to_leak >>> 0;
```

Listing 1: Exploit for CVE-2017-7484

The vulnerability is a bug in how Postgres enforces access control during query planning. An attacker can leverage a custom-defined function and operator to leak information with a `SELECT` query. Although protected data is not leaked in the query results themselves, it is leaked when the database creates the query plan, which passes sensitive information to the attacker's custom operator. We assume that a protected table exists within the database. An unprivileged user can use the exploit shown in Listing 1 to create a custom operator that leaks this table during query planning.

Because the vulnerability affects Postgres but not CockroachDB (CockroachDB does not support user-defined functions and operators), RDDR mitigates it successfully. If these features were required by the application, this particular diversification would not be appropriate. That is, all instances of the N-versioned deployment must share the minimum set of required features to implement the target application. In addition, the diverse implementations must be configured to behave identically under normal operation. For example, in our deployment we configured Postgres' transaction isolation level to match CockroachDB, which forces serializable isolation (the strictest setting). In our evaluation, RDDR was configured to anticipate the different software version strings. There may be other application complexities which that developer has to specify for RDDR. For example, the PostgreSQL query language does not require any particular row order unless specified by the `ORDER BY` keyword, and each implementation is allowed to order rows arbitrarily. If they differ, then RDDR will block the benign traffic.

Once the N instances were configured, CVE-2017-7484 was mitigated. The exploit shown in Listing 1 fails at the first step. The Postgres instance indicates successful creation of the custom function, whereas CockroachDB raises an error indicating that the feature is not supported. RDDR identifies the divergence and breaks the connection before either response can reach the client. If the attacker tries to reconnect and proceed with subsequent steps of the attack, the final `EXPLAIN` query which causes the leak is always blocked. This example demonstrates that RDDR successfully protects the system even when an attacker is aware of RDDR and the N-version deployment. One might object that the user should always choose the more secure implementation, but in many cases, this is not known at deployment time. The example also demonstrates the feasibility of deploying very diverse implementations of the same logical function behind RDDR, and it illustrates some of the challenges we encountered, such as unspecified row order behavior.

As with any diversity method, this example is susceptible to common mode failures. However, the Project on Diverse Software found that common mode failures in diverse software implementations are rare, and the few that do appear generally arise from a flaw in the specification itself rather than a recurrent bug present in every codebase [18].

### D. Varying microservice versions

Next, we consider different versions of the same microservice as a source of diversity. Nginx is a widely used proxy for web applications, and CVE-2017-7529 [5] exploits versions up to 1.13.2 with an integer overflow that can leak sensitive server information. The vulnerability arises from a bug in how nginx processes a content range request. The attacker sends a specially crafted Range header in a HTTP request, nginx fails to check its bounds which leads to an integer overflow when calculating the size of the payload to return, causing it to return data past the end of the requested document to the client.

To mitigate this vulnerability, we deployed a three-version configuration of nginx, with the two instances comprising the filter pair running version 1.13.2, and the third instance running 1.13.4 which is not vulnerable. This example models a hypothetical deployment strategy where developers deploy old and new versions of the same application in parallel behind RDDR to patch bugs in the old version while protecting against any new bugs accidentally introduced by the patch. We ran the exploit against the N-versioned nginx and RDDR successfully aborted the connection. RDDR observes that one instance's response is much longer than the other's and treats this as divergent behavior.

N-versioned deployments of multiple versions are straightforward to deploy because of the way that containerized platforms like Docker handle versioning. These platforms have built in support for specifying image versions by tag. Thus, the deployed version can be changed by simply changing the specified version tag.

### E. Generating Diversity with ASLR

RDDR can prevent pointer leaks in the presence of ASLR which, *e.g.*, were used in the heartbleed [3] exploit. We demonstrate this capability with a proof-of-concept C program— a simple echo server that stores the requester's message in a buffer and returns it without checking for overflow. If the requester overwrites the null terminator at the end of the buffer, the program leaks a pointer adjacent to the buffer in the stack. When the program is deployed with ASLR, this helps the attacker learn the location of an exploitable gadget. The attacker's exploit proceeds as follows: (1) send a large payload to cause the program to leak a pointer; (2) calculate the address of the gadget as an offset from the leaked pointer; (3) send an even larger payload to overwrite the return address with the calculated address of the gadget. RDDR defeats this exploit at step (1) by detecting and preventing the pointer leak. When two instances of the same binary with ASLR are N-versioned, each has a unique address space. When the attacker tries to leak a pointer, each instance reports a different address, triggering a divergence and termination of the connection to the attacker. This deployment does not require non-deterministic filtering, since RDDR's filter ignores addresses that differ in the filter pair.

This case study demonstrates how RDDR can leverage OS-generated diversity like ASLR. Although other N-versioning systems have similar capabilities (§ III) in this space, RDDR brings this capability to the microservice setting.

### F. N-versioning components of GitLab

Here we apply RDDR to GitLab [34], a popular web-based platform for hosting and collaborating on source code repositories. Because some components of the deployment are more important than others, N-versioning with RDDR is applied only to the most relevant subset of the containers.

*1) GitLab Architecture:* The GitLab application is constructed from a number of smaller microservices, some of which were developed in-house by the GitLab team and others

that are independent open-source projects (*e.g.*, Postgres and nginx).

Figure 3 presents a simplified view of the GitLab architecture (with RDDR guarding the Postgres module). At the top of the diagram are client interfaces for SSH and HTTP(S). This traffic is routed to the GitLab shell and nginx ingress proxy respectively. From there, a request may be passed to a number of services depending on its type. For this study, we considered vulnerabilities in a particular microservice that could affect GitLab, *e.g.*, the attacks discussed in sections V-C and V-F.

*2) N-versioning Postgres within GitLab:* We consider the information leak described in CVE-2019-10130, and use RDDR to diversify the Postgres database microservice in a three-instance configuration.

```
-- Create leaky function, and operator to call it
CREATE FUNCTION op_leak(int, int) RETURNS bool
    AS 'BEGIN
        RAISE NOTICE ''leak %, %'', $1, $2;
        RETURN $1 < $2; END'
    LANGUAGE plpgsql;
CREATE OPERATOR <<< (procedure=op_leak, leftarg=int,
     rightarg=int, restrict=scalarltsel);
-- Will call leaky function, printing rows to console
    .
SELECT * FROM some_table WHERE col_to_leak <<< 1000;
```

Listing 2: Exploit for CVE-2019-10130

CVE-2019-10130 affects Postgres 10.0 through 10.7, allowing a user-defined operator to leak privileged information from a table that enforces per-row security policies. We assume the presence of an SQL injection vulnerability in the frontend of the application which enables the attacker to send arbitrary SQL queries to the backend database to affect the exploit. To be vulnerable, a privileged database user must first create a table with row-level security, grant `SELECT` privileges to another database user, but deny access to one or more rows. The unprivileged user can then execute the exploit shown in Listing 2 to leak the protected rows. First, she creates a function that prints its arguments to the console. Then, she creates a custom operator to call this function. Finally, she executes a `SELECT` query that invokes her custom operator. The `SELECT` query itself will not return the protected rows, since access control is properly implemented for `SELECT`. However, the function will still be passed the value of column 'a' from every row in the table, and the function can leak them out.

We compose the N-versioned Postgres deployment from three instances of Postgres, two at version 10.7 (buggy filter pair) and a third at version 10.9 (fixed). During the exploit, RDDR detects the difference in behavior between 10.7 and 10.9 and aborts the connection. The modifications to the GitLab architecture are shown in Figure 3. An empty database is initialized with the schema for GitLab and instantiated in each instance. GitLab is configured to use an external Postgres database and RDDR's incoming proxy, which forwards all queries to every Postgres instance. All benign GitLab functions remain fully operational: users can log in, create projects, view projects and more, and RDDR does not interfere. Only

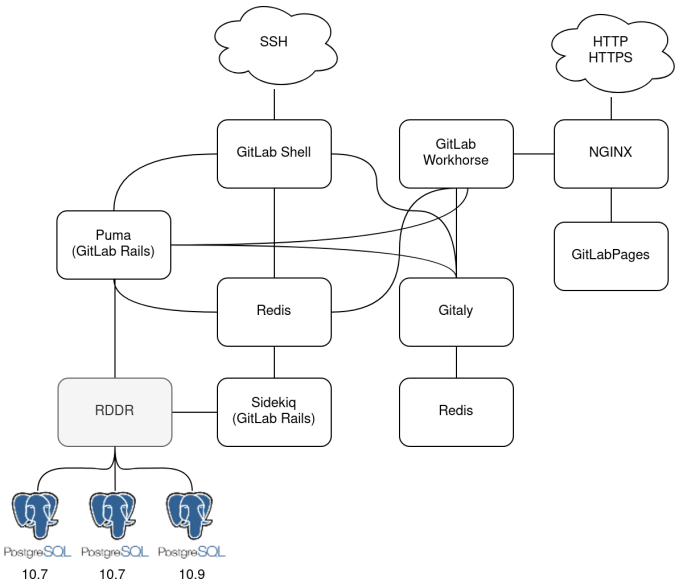



Fig. 3: Modified GitLab architecture with Postgres replicated behind RDDR(bottom left).

when a neighboring container launches the described exploit does RDDR react, closing the connection to the client before protected rows from the table are leaked.

This deployment shows that RDDR functions robustly when deployed in a complex system with high levels of benign traffic. It also illustrates how RDDR can be deployed to protect individual services, which is much more scalable than monolithic approaches. This enables developers to pay the cost of protecting only the most critical services in their application, maximizing reliability and minimizing overhead.

### G. Performance

*1) TPC-H Benchmark Performance:* By design N-versioned systems incur the overhead of executing multiple versions in parallel. RDDR minimizes this cost through micro-versioning—replicating and diversifying only the most vulnerable microservices, *e.g.*, those that handle unmodified user data. The extent of this savings, however, depends on the particular configuration of microservices in any given deployment. Therefore, we quantify RDDR's overhead by focusing on a single representative component—a Postgres database.

First, we compare the performance of a single instance of a Postgres database (without RDDR) to a 3-version deployment behind RDDR, where all Postgres instances are identical, using TPC-H [6], a widely used benchmark in industry for studying database performance. The benchmark specifies a database schema and 22 test queries. We initialized each instance with a TPC-H database using a factor of 10, creating a 10GB database. We then executed all the queries (except one that could not be executed in parallel) against each of our two deployments, when they were running 1, 2, 4, 8 or 16 clients in parallel. For each query, we measured time to execute, memory usage, and CPU usage of each deployment for each number of

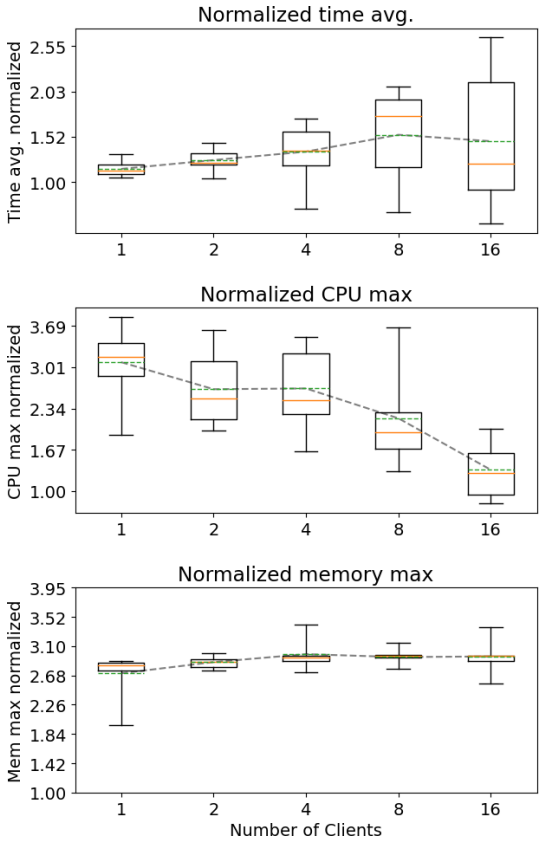



Fig. 4: Performance of RDDR normalized to the baseline and run with concurrent clients. Boxes span the $5^{th}$ through $95^{th}$ percentile. The mean for each sub-graph is depicted as a dashed line, and medians in orange.

clients. We measured only the memory and CPU usage of the process tree that comprises each deployment. All tests were run on an AWS virtual machine with 32 vCPUs and 128 GB of memory. Figure 4 reports our results, which are normalized to the performance of the baseline single-instance deployment (without RDDR).

As expected, the memory overhead of 3-versioning is approximately $3\times$ that of a single instance as seen in the bottom plot of Figure 4. CPU utilization overhead is roughly $3\times$ with one client, but drops quickly when run with more clients. We attribute this to the fact that Postgres quickly used all available cores to service requests in parallel for both the single-instance and 3-version deployments. Finally, the average slowdown incurred by RDDR (Figure 4 top plot, dashed line) approaches a constant value and does not increase exponentially even when the number of clients does.

*2) Throughput and Latency:* To evaluate throughput and latency, we deployed RDDR on three identical Postgres instances and evaluated using the `pgbench` benchmark. Its performance is compared to a baseline deployment consisting of a single Postgres instance, both with and without an Envoy front proxy. By comparing to a database proxied by an Envoy instance we can evaluate how RDDR compares to

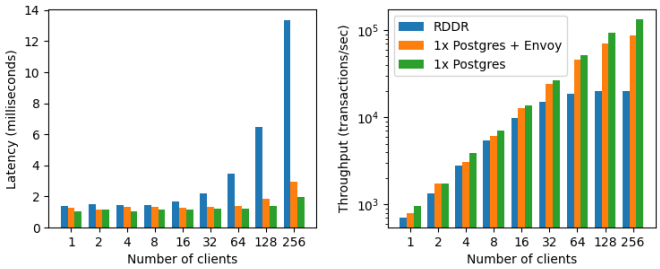



Fig. 5: Throughput and latency for 10,000 transactions per client.

an optimized and widely used proxy designed to be cloud native. Each deployment was hosted on an `m5a.8xlarge` AWS machine with 32 virtual CPUs and 128 GB of memory, which we will refer to as the "server machine." The `pgbench` benchmark was executed from a separate `m5a.4xlarge` AWS machine with 16 virtual CPUs and 64 GB of memory, which we will refer to the "client machine."

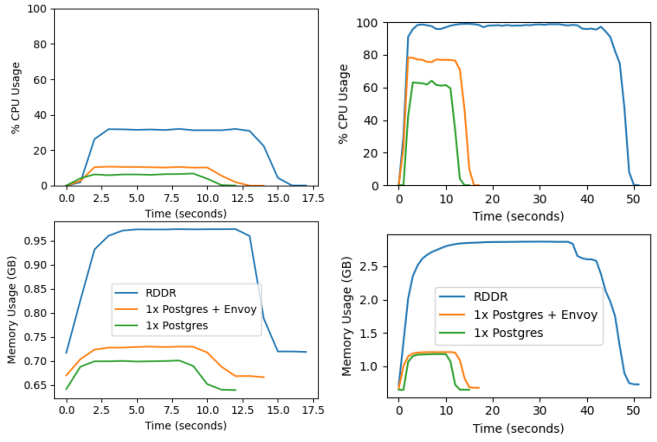
Each deployment was initialized with a database of scale factor 100, creating a total of 10,001,100 table rows. We ran `pgbench` for different numbers of simultaneous clients, ranging from 1 to 256 in powers of two. Each client is executed in a separate thread and makes 10,000 SELECT transactions against each deployment. RDDR's performance is compared to two different baseline deployments: a single instance of Postgres with an Envoy front proxy, and a single instance without a front proxy. These experiments allow us to measure RDDR's overhead and how it compares to the cost of adding a front proxy (common in many cloud application deployments). Figure 5 shows RDDR's throughput and latency compared to each baseline. At 8 clients, RDDR incurs a 10% reduction in throughput and an 11% increase in latency compared to the single instance of Postgres behind Envoy. Above that level, the host machine became overloaded, which explains RDDR's relative performance decline.

In Figure 6, we quantify the relative CPU and memory usage of each deployment when serving 16 or 128 simultaneous clients. At 16 clients, RDDR exhibits roughly 3x memory and compute overhead, showing that RDDR incurs only a small additional overhead beyond the expected 3X cost for a 3-instance deployment. As we increase to 128 clients, however, the compute resources of the server become constrained and RDDR experiences near 100% CPU utilization.

In summary, the latency incurred is generally constant and acceptably low (Figure 5), while the memory and CPU overhead is approximately 3x for the container being protected, provided the system resources have not been exhausted (Figure 6).

## VI. Discussion

In large microservice deployments, it may not be obvious which containers should be prioritized for N-versioning. Generally, any microservice that processes user input data directly is at risk for leaking data and should be given high priority




(a) 16 clients (b) 128 clients

Fig. 6: Aggregate CPU and memory usage for each deployment with 16 and 128 clients.

for protection. This includes containers that contain encryption functions, parsers, tokenizers, proxies, and sanitizers (§ V). Similarly, containers that receive external input via API calls (*e.g.*, [30], [57]) are good candidates for N-versioning. Although these recommendations do not cover every possible deployment type, they include categories that are overall both highest risk and most likely to benefit from N-Versioning.

Each case study involved a different example deployment, and in each case RDDR protected the system against the tested vulnerability. Although we cannot guarantee that our results generalize beyond these examples, we are optimistic that the methods described here generalize and will defeat many similar forms of attack, and RDDR is designed to support the importation of other sources of diversity.

We evaluated RDDR's run-time performance using two benchmarks. On the TPC-H benchmark, memory and CPU overhead for one client behaved as expected with about a 3x increase for a 3-instance deployment of the micro-service. On the `pgbench` benchmark, we profiled throughput and latency, comparing RDDR to a single instance of a database with and without the frontend proxy. We found that, on our 32-core server, RDDR's throughput tapers off above 16 simultaneous clients, when RDDR running with three instances exhausts the parallelism of the server more quickly than the baselines running with one. Such degradation can be mitigated by upgrading to servers with more cores, or deploying each instance of the N-versioned set on a different machine; RDDR can easily be reconfigured to run distributed across multiple hosts.

RDDR is advantageous when relatively few microservices within a deployment require N-versioning, such as the GitLab example (Figure 3) where one service out of nine was replicated, adding three new containers. Assuming that all containers consume equal resources, the expected overhead would be 33% instead of the 3X required by traditional monolithic N-versioning. In deployments such as Netflix or Uber, which are typically composed of 100's of microservices, duplicating a few select microservices (the API sever for example) would have negligible effect on overhead.

RDDR can be used to mitigate many types of attacks, and our work highlights several important examples. However, assessing mitigations using standards such as the OWASP is problematic because specific details of an attack can vary widely. For example, (5) on the list, which encompasses *XML External Entities* (XXE), can be mitigated in some cases (as we showed for CVE-2020-10799), but not in others, *e.g.*, DOS attacks triggered by other XXE attacks. A similar issue arises if we map Common Weakness Enumeration (CWE) numbers to specific RDDR use cases since the effect of the exploit must be observable at either RDDR's incoming or outgoing proxy to be mitigated.

RDDR is not applicable to social engineering attacks where a malicious actor uses legitimate credentials to access system components. Such attacks will not cause divergence, regardless of the source of diversity.

Important areas for future work are threat models that include hardware vulnerabilities. RDDR could potentially diversify microservices at the hardware level, *e.g.*, using Kubernetes, which can orchestrate a deployment across multiple physically-diverse machines. Diversity could include the ISA (x86, ARM, MIPS, *etc.*) or the chip manufacturer (Intel, AMD, Snapdragon, *etc.*).

## VII. Conclusion

Microservice architectures are the dominant approach for building today's large-scale, public-facing, distributed cloud applications. By revisiting and reframing the classic security strategy of N-versioning, RDDR provides robust protection for the most vulnerable components of these systems at a fraction of the cost of replicating the entire application. This approach, together with the implementation challenges that RDDR overcomes, addresses one of the major drawbacks of N-versioning—the high cost of running the versions. RDDR can mitigate important web application vulnerabilities with minimal overhead beyond the cost of running multiple instances of vulnerable components and provides a scalable approach for protecting large complex micro-service architectures. We hope that the work presented here will bring the benefits of N-version systems to microservice architectures and contribute to improving their security.

## Acknowledgments

We would like to thank our anonymous reviewers for their insightful feedback. We gratefully acknowledge the partial support of the NSF (CCF 1908633, OAC 2115075, CNS 1817020, CNS 1704778), DARPA (FA8750-19C-0003, N6600120C4020), AFRL (FA8750-19-1-0501), Intel (SCAP and SRC 2965.001), and the Santa Fe Institute.

## REFERENCES


[1] Message formats. In *PostgreSQL: Documentation*, chapter 52.7. The PostgreSQL Global Development Group, 12 edition.

[2] OWASP risk rating methodology. https://owasp.org/www-community/OWASP_Risk_Rating_Methodology.

[3] CVE-2014-0160. CVE Details, Apr 2014.

[4] CVE-2017-7484. CVE Details, Jul 2017.

[5] CVE-2017-7529. CVE Details, Jul 2017.

[6] TPC Benchmark H. Technical Report 2.18.0, Transaction Processing Performance Council, Dec 2018.

[7] Common Vulnerability Scoring System v3.1: Specification Document. Technical Report 3.1, FIRST.org, Inc., June 2019.

[8] CVE-2019-18277. CVE Details, Oct 2019.

[9] Deploy to swarm. https://docs.docker.com/get-started/swarm-deploy/, Mar 2020.

[10] Hanane Abdeldjelil, Noura Faci, Zakaria Maamar, and Djamal Benslimane. A diversity-based approach for managing faults in web services. In *2012 IEEE 26th International Conference on Advanced Information Networking and Applications*, page 81–88, Mar 2012.

[11] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana Maria Popa. Firecracker: Lightweight virtualization for serverless applications.

[12] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[13] A. Avizienis and Ling Chen. On the implementation of n-version programming for software fault tolerance during program execution. 1977.

[14] M. Azab, B. Mokhtar, A. S. Abed, and M. Eltoweissy. Toward smart moving target defense for linux container resiliency. In *2016 IEEE 41st Conference on Local Computer Networks (LCN)*, pages 619–622, 2016.

[15] Michael Backes, Goran Doychev, and Boris Köpf. Preventing side-channel leaks in web traffic: A formal approach. Feb 2013.

[16] G. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefnaovic, and D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *10th ACM Conf. on Computer and Communications Security*, 2003.

[17] Emery D Berger and Benjamin G Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. page 11.

[18] P. G. Bishop, D. G. Esp, M. Barnes, P. Humphreys, G. Dahll, and J. Lahti. Pods — a project on diverse software. *IEEE Transactions on Software Engineering*, SE-12(9):929–940, 1986.

[19] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, volume 1, pages 3–9, 1978.

[20] Cloud Native Computing Foundation. Envoy. https://www.envoyproxy.io.

[21] M. Co, J. W. Davidson, J. D. Hiser, A. Nguyen-Tuong J. C. Knight, W. Weimer, J. Burket, G. L. Frazier, B. Dutertre T. M. Frazier, I. Mason, N. Shankar, and S. Forrest. Double Helix and RAVEN: A system for Cyber Fault Tolerance and Recovery. In *Proc. of the 11th Cyber and Information Security Research Conf.* Oak Ridge National Laboratory, 2016. Runner up, best paper.

[22] Michele Co, Bruno Dutertre, Ian Mason, Natarajan Shankar, Stephanie Forrest, Jack W. Davidson, Jason D. Hiser, John C. Knight, Anh Nguyen-Tuong, Westley Weimer, Jonathan Burket, Gregory L. Frazier, and Tiffany M. Frazier. Double Helix and RAVEN: A System for Cyber Fault Tolerance and Recovery. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference on - CISRC '16*, pages 1–4, Oak Ridge, TN, USA, 2016. ACM Press.

[23] Cockroach Labs. CockroachDB. https://www.cockroachlabs.com.

[24] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *In Proceedings of the 15th USENIX Security Symposium*, pages 105–120, 2006.

[25] Olivier Crameri, Nikola Knezevic, Dejan Kostic, Ricardo Bianchini, and Willy Zwaenepoel. Staged deployment in mirage, an integrated software upgrade testing and distribution system. *ACM SIGOPS Operating Systems Review*, 41, 10 2007.

[26] Dewhurst Security. Damn Vulnerable Web App. https://www.cockroachlabs.com.

[27] Glen Dobson, Stephen Hall, and Ian Sommerville. A container-based approach to fault tolerance in service-oriented architectures. In *International Conference of Software Engeneering. Citeseer*, 2005.

[28] EnterpriseDB Corporation. EnterpriseDB. https://www.enterprisedb.com.

[29] Dominik Ernst, David Bermbach, and Stefan Tai. Understanding the container ecosystem: A taxonomy of building blocks for container lifecycle and cluster management. In *Proceedings of the 2nd International Workshop on Container Technologies and Container Clouds, IEEE*, 2016.

[30] facebook.com. Graph api - documentation - facebook for developers. https://developers.facebook.com/docs/graph-api/.

[31] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[32] Robert Gawlik, Philipp Koppe, Benjamin Kollenda, Andre Pawlowski, Behrad Garmany, and Thorsten Holz. Detile: Fine-grained information leak detection in script engines. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, page 322–342, Berlin, Heidelberg, 2016. Springer-Verlag.

[33] Sara Gholami, Alireza Goli, Cor-Paul Bezemer, and Hamzeh Khazaei. A framework for satisfying the performance requirements of containerized software systems through multi-versioning. ACM, Dec 2019.

[34] GitLab Inc. GitLab. https://about.gitlab.com/.

[35] Anatoliy Gorbenko, Vyacheslav Kharchenko, and Alexander Romanovsky. Using inherent service redundancy and diversity to ensure web services dependability. In *Methods, Models and Tools for Fault Tolerance*, pages 324–341. Springer, 2009.

[36] Ronen Heled. Http request smuggling. 2005.

[37] P. Hosek and C. Cadar. Safe software updates via multi-version execution. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 612–621, 2013.

[38] https://cwe.mitre.org. Cwe-1026: Weaknesses in owasp top ten (2017), 2019.

[39] Rui Huang, Hongqi Zhang, Yi Liu, and Shie Zhou. Relocate: a container based moving target defense approach. In *The 7th International Conference on Computer Engineering and Networks*, volume 299, page 008. SISSA Medialab, 2017.

[40] Jessica Jones, Jason D. Hiser, Jack W. Davidson, and Stephanie Forrest. Defeating Denial-of-Service Attacks in a Self-Managing N-Variant System. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 126–138, Montreal, QC, Canada, May 2019. IEEE.

[41] J. P. J. Kelly, A. Avižienis, B. T. Ulery, B. J. Swain, R.-T. Lyu, A. Tai, and K.-S. Tso. Multi-version software development. *IFAC Proceedings Volumes*, 19(11):43–49, 1986.

[42] John Patrick Joseph Kelly. Specification of fault-tolerant multi-version software: Experimental studies of a design diversity approach. 1983.

[43] John C Knight and Nancy G Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on software engineering*, (1):96–109, 1986.

[44] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. *DSN*, pages 431–442, 2016.

[45] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy*, pages 276–291, 2014.

[46] Robert E Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM journal of research and development*, 6(2):200–209, 1962.

[47] Soo Jin Moon, Vyas Sekar, and Michael K. Reiter. Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1595–1606. Association for Computing Machinery, Oct 2015.

[48] Nginx, Inc. Nginx. https://www.nginx.com.

[49] Christian Otterstad and Tetiana Yarygina. Low-level exploitation mitigation by diverse microservices. In Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen, editors, *Service-Oriented and Cloud Computing*, Lecture Notes in Computer Science, page 49–56. Springer International Publishing, 2017.

[50] Richard Poschinger, Nils Rodday, Raphael Labaca-Castro, and Gabi Dreo Rodosek. Openmtd: A framework for efficient network-level mtd evaluation. In *Proceedings of the 7th ACM Workshop on Moving Target Defense*, pages 31–41, 2020.

[51] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano. A source-to-source compiler for generating dependable software. In *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 33–42, 2001.

[52] David K. Rensin. *Kubernetes: Scheduling the Future at Cloud Scale*. O'Reilly Media, 2015.

[53] Atul Singh, Nishant Sinha, and Nitin Agrawal. Avatars for pennies: Cheap n-version programming for replication. In *6th Workshop on Hot Topics in System Dependability*, 2010.

[54] Neil Smithline, Brian Glas, Torsten Gigler, and Andrew van der Stock. OWASP Top 10, 2021.

[55] Marthony Taguinod, Adam Doupé, Ziming Zhao, and Gail-Joon Ahn. Toward a Moving Target Defense for Web Applications. In *Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI)*, August 2015.

[56] Kennedy A. Torkura, Muhammad I.H. Sukmana, Anne V.D.M. Kayem, Feng Cheng, and Christoph Meinel. A cyber risk based moving target defense mechanism for microservice architectures. In *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, page 932–939, Dec 2018.

[57] Inc. Twitter. Twitter api documentation — docs — twitter developer platform. view-source:https://developer.twitter.com/en/docs/twitter-api.

[58] Inc Twitter. Diffy. https://github.com/opendiffy/diffy, Mar 2020.

[59] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. Secure and efficient application monitoring and replication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 167–179, Denver, CO, June 2016. USENIX Association.

[60] Alexios Voulimeneas, Dokyung Song, Per Larsen, Michael Franz, and Stijn Volckaert. dmvx: Secure and efficient multi-variant execution in a distributed setting. In *Proceedings of the 14th European Workshop on Systems Security*, pages 41–47, 2021.

[61] Alexios Voulimeneas, Dokyung Song, Fabian Parzefall, Yeoul Na, Per Larsen, Michael Franz, and Stijn Volckaert. Distributed heterogeneous n-variant execution. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 217–237. Springer, 2020.

[62] John H Wensley, Leslie Lamport, Jack Goldberg, Milton W Green, Karl N Levitt, Po Mo Melliar-Smith, Robert E Shostak, and Charles B Weinstock. Sift: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, 1978.

[63] Riley Wood and Antonio Espinoza. Rddr documentation. https://rddr.readthedocs.io/en/latest/.

[64] Riley Wood and Antonio Espinoza. Rddr source code. https://bitbucket.org/rddr-team/rddr/src/master/.

[65] Ying C Yeh. Triple-triple redundant 777 primary flight computer. In *1996 IEEE Aerospace Applications Conference. Proceedings*, volume 1, pages 293–307. IEEE, 1996.

[66] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. *SIGOPS Oper. Syst. Rev.*, 37(5):253–267, October 2003.

[67] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 26–36, New York, NY, USA, 2011. Association for Computing Machinery.

[68] Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. kMVX: Detecting Kernel Information Leaks with Multi-variant Execution. In *ASPLOS*, April 2019.