ChatGPT

Synthetic Digital Immunity (SDI) Architecture Diagrams

Synthetic Digital Immunity (SDI) is a proactive cybersecurity architecture inspired by biological immune systems – *“an AI-driven defense mechanism that mimics the human immune system to detect anomalies, neutralize threats, and ensure operational continuity”* [1]. In the SDI approach, cloud microservices are equipped with polymorphic “hunter-killer” capabilities that detect attacks and autonomously morph to eliminate threats. Below, four figures detail the SDI architecture and its key components: an overview of the system (Figure 1), the Polymorphic Response Engine (PRE) pipeline (Figure 2), a genetic algorithm (GA) convergence graph (Figure 3), and a mutation topology comparison (Figure 4). Each diagram is formatted for IEEE-style embedding (black-and-white print optimized, with optional color for digital viewing).

SDI Architecture Overview (Figure 1)

Figure 1: Synthetic Digital Immunity (SDI) architecture overview. This diagram shows an end-to-end view of the SDI system within a federal cloud Kubernetes cluster. Ingress traffic from users/clients flows through a Load Balancer into microservices deployed in the cluster (two sample services are shown: *Service 1* and *Service 2*), each running an Envoy sidecar proxy for enhanced observability and control [2]. During normal operation, Envoy proxies forward client requests (HTTP/S) to the services. When an anomaly or suspected attack is detected (① **Detection**), the Envoy sidecar generates an **anomaly event** and publishes it to the **“Immune Bus”** – an event streaming bus implemented with Apache Kafka (dashed arrows to Kafka in the figure). The Kafka-based **immune event bus** asynchronously feeds these events to an external **AI Analyzer** service for further inspection. Simultaneously, the affected service’s sidecar can divert suspicious traffic to a dedicated **Honeypot container** (③ **Isolation** phase, shown via the dotted red path) to safely capture the malicious payload without impacting the production service. The AI Analyzer performs *antigen extraction* on the incoming events (identifying the attack signature/pattern from the anomaly data) and determines an appropriate countermeasure. It then triggers a CI/CD pipeline (using Jenkins or GitLab runners) for **mutation synthesis** (④), launching an automated build/test process to create a **mutated “antibody” microservice** variant. In the **Propagation** stage (⑤), this new **antibody microservice** – a patched or morphed version of the original service code – is redeployed into the Kubernetes cluster, replacing or augmenting the vulnerable service instance. The entire cycle (Detection → Isolation → Extraction → Mutation → Propagation) happens continuously and autonomously, enabling the system to *“adapt instantly and self-heal before users even notice a glitch”* [3] [1]. *(Note: Envoy sidecars run alongside each microservice to provide consistent network security and telemetry, serving as the “sensors” and first responders in the SDI architecture* [2] *.)*

PRE Pipeline Flow (Figure 2)

Figure 2: Polymorphic Response Engine (PRE) five-phase pipeline. The SDI’s adaptive defense is organized into a five-stage pipeline that mirrors an immune response: **Detection → Isolation → Antigen Extraction → Mutation Synthesis → Propagation**. In this flow, each phase takes specific inputs and

produces outputs (artifacts) that feed into the next stage. As shown in the diagram, an anomaly detected by the system triggers the **Detection** phase, yielding an **Anomaly Token (AT)** – a concise signature or "fingerprint" of the threat event. Next, during **Isolation**, the suspicious activity is confined (e.g. by redirecting it to a honeypot or quarantining the affected service), producing an artifact denoted **ETB** (for example, an *Exploit Test Bundle* containing captured malicious payload and context). The **Antigen Extraction** phase processes the ETB to distill a defensive "antigen" – represented by **VS**, which could stand for a *Variant Specification* or *Vaccine Signature* that characterizes the exploit's key traits. This VS artifact then enters **Mutation Synthesis**, where a genetic algorithm or program transformation engine generates a mutated microservice variant (e.g. injecting validation checks or rearranging code). The output of this stage is **MP**, a *Mutated Package/Program* – essentially a new microservice build candidate. Finally, in **Propagation**, the mutated service is deployed to the live environment, resulting in an **IFS** artifact – an *Immune Fortified Service* instance now running in the cluster. The figure uses color-coded arrows to illustrate the progression and hand-off of these artifacts between stages: for example, the blue arrow from Detection to Isolation carries the Anomaly Token, orange from Isolation to Extraction carries ETB, and so on. By the end of the pipeline, the threat is neutralized and the system has evolved, with the new service instance exhibiting polymorphic defenses against the previously encountered attack. This closed-loop pipeline ensures that **each attack stimulates an automated immune response**, progressively hardening the microservice ecosystem with minimal human intervention.

GA Convergence Graph (Figure 3)

Figure 3: Genetic algorithm convergence for automated mutation synthesis. This chart depicts the convergence of a genetic algorithm (GA) used in the Mutation Synthesis phase to evolve microservice code variants. The x-axis represents successive generations of the GA (0 to 50), and the y-axis shows the **average fitness** of the population in each generation (on a 0–1 scale). The fitness function is defined to measure how effective a mutated service is at blocking or mitigating the target attack without regressing functionality. Two example run profiles ("Run 1" – solid line, and "Run 2" – dashed line) are plotted, illustrating how the algorithm improves candidate solutions over generations. Starting from a lower initial fitness (~0.2–0.3 at generation 0), the average fitness in both runs increases monotonically as beneficial mutations are selected and propagated. A horizontal red dotted line marks a **convergence threshold** at 0.95 (95% of the theoretical maximum fitness). In Run 1, the population crosses this threshold at **generation 19**, whereas in Run 2 it occurs at **generation 28** (indicated by the annotated markers on the curves). After reaching the threshold, the improvements level off, signifying that the GA has found a highly fit mutated service design (further generations yield only marginal gains). This demonstrates reliable convergence: despite different random initial populations, both runs evolve towards an effective solution that meets the desired fitness criterion. The convergence graph validates that the **polymorphic mutation strategy can automatically produce a robust "antibody" service variant within a few dozen generations**, supporting rapid adaptation in the SDI framework. (In practice, convergence criteria like reaching fitness $\geq$0.95 or no significant improvement over N generations would trigger the pipeline to proceed to deployment of the new microservice.)

Mutation Topology Diagram (Figure 4)

Figure 4: Microservice code mutation – before vs. after. This diagram compares a microservice's code structure **before** and **after** the polymorphic transformation applied in the Mutation Synthesis stage. On the **left**, the original service code is illustrated in simplified form (pseudo-code steps 1–4). In this example, the service might perform a sequence of operations: e.g., (1) validateUser(); (2) fetchData(); (3) processData(); (4)

returnResult(). The **right** side shows the **mutated service code** after an AST (Abstract Syntax Tree) transformation and recompilation. Several changes are highlighted: **a guard check is inserted up front** (step 1: guardCheck()) to pre-empt the specific exploit vector identified by the anomaly token (this is a new input validation or sanity-check that was not in the original code). The business logic steps have been **reshaped in control-flow** – here we show that `processData()` and `fetchData()` have been re-ordered (steps 2 and 3) or otherwise modified, without altering the overall functional outcome. This reshuffling, along with potential no-op instruction inserts or refactoring, ensures the resulting binary's structure is significantly different from the original (the diagram notes a **bytecode Δ > 30%**, meaning over 30% of the compiled code has changed). Such polymorphic alterations *"mutate while keeping the original algorithm intact"* [4], preserving the service's external behavior (correct outputs) but in a new internal form. The net effect is a moving target for attackers: the mutated "antibody" service instance is immune to the specific exploit used in the attack (thanks to the guard and other changes) and its altered code footprint thwarts malware signature recognition or repeat attacks. After mutation, the new service is deployed in place of the old one, and normal operations resume, now with improved resilience. This **before-and-after code topology** illustrates how SDI's polymorphic engine systematically generates functionally equivalent yet internally diversified microservice implementations to stay ahead of evolving threats.

---

[1] [3] The Role of Digital Immune System in Cyber Resilience
https://appinventiv.com/blog/digital-immune-system/

[2] Envoy proxy - home
https://www.envoyproxy.io/

[4] Polymorphic code - Wikipedia
https://en.wikipedia.org/wiki/Polymorphic_code