Dynamic On Demand Decoy Deployment Using MicroVMs

Mohammad Shariq Farooqui
University of Essex
Colchester, United Kingdom
shariq.farooqui@essex.ac.uk

Amit Kumar Singh
University of Essex
Colchester, United Kingdom
a.k.singh@essex.ac.uk

Pushpinder Kaur Chouhan
British Telecom
United Kingdom
pushpinder.kaur.chouhan@bt.com

Zhan Cui
British Telecom
United Kingdom
zhan.cui@bt.com

***Abstract*—Honeypots remain a key defensive technique for engaging intruders and gathering intelligence, yet existing designs struggle to balance interaction realism, resource efficiency, and security isolation. Low-interaction systems, which emulate services rather than running them, are lightweight but easily fingerprinted, while container-based approaches offer convenience but expose kernel-sharing risks. Recent deception frameworks such as CATCH have proposed dynamic decoy deployment, where high-interaction environments are instantiated only when attacker behaviour warrants it. However, CATCH does not prescribe a concrete mechanism capable of delivering safe, high-fidelity decoys with sub-second responsiveness. This paper provides that missing mechanism by introducing a microVM-based on-demand isolation architecture for SSH-initiated deception. When suspicious activity is detected—via honeytokens in our prototype—sessions are transparently redirected into dedicated Firecracker microVMs restored from snapshots. This approach operationalises the dynamic deployment concept proposed by CATCH by offering a lightweight execution substrate capable of spawning realistic, strongly isolated, per-attacker environments at the moment of detection. We implement a PostgreSQL protocol handler as a case study and demonstrate median microVM startup latency of 242 ms—comparable to Docker container startup (300 ms) but with full VM-level isolation rather than shared-kernel containment—with 106 MB memory usage per attacker and zero overhead for legitimate users. Timing analysis shows that fingerprinting is limited to a brief first-query window; subsequent interactions are indistinguishable from real systems. These results indicate that microVMs provide a practical foundation for scalable, high-interaction deception and represent a viable dynamic deployment backend for CATCH-style active-defence architectures.**

***Index Terms*—deception technology, honeypot, microVM, Firecracker, active defence**

I. Introduction

Credential theft remains the primary attack vector in web application breaches, with stolen credentials involved in 88% of basic web application attack breaches according to the 2025 Verizon Data Breach Investigations Report [1]. Once attackers obtain valid credentials, authentication offers no further resistance. Zero Trust Architecture (ZTA) [2] addresses this through continuous verification, but most ZTA implementations focus on access control rather than active engagement with suspicious users.

Honeypots offer a different approach: rather than blocking suspicious connections, they engage attackers within controlled environments to gather intelligence and waste attacker resources. Any traffic a honeypot receives is inherently suspicious, since legitimate users have no reason to access it [3]. However, honeypot design involves a trade-off among three properties.

Interaction depth determines how realistically the honeypot responds to attacker actions. Prior work demonstrated that low- and medium-interaction honeypots could be systematically fingerprinted through protocol inconsistencies, constituting a class break in honeypot security [4]. Deep systems that execute real commands eliminate most fingerprinting vectors but require substantial infrastructure.

Resource efficiency constrains deployment scale. High-interaction honeypots traditionally required dedicated hardware per instance, limiting deployments to a handful of monitored systems.

Security isolation protects operators from compromise. Any system executing attacker commands risks being taken over. Container-based approaches share the host kernel, creating vulnerabilities that sophisticated attackers can exploit [5].

Table I illustrates how existing approaches sacrifice at least one property. This paper addresses the trade-off using lightweight virtualisation. Recent developments in microVM technology, driven by serverless computing requirements, have produced hypervisors capable of booting isolated environments in hundreds of milliseconds [6]. By spawning such environments on demand when suspicious activity is detected, the architecture achieves genuine operating system interaction, minimal per-instance overhead, and hardware-enforced isolation simultaneously.

TABLE I
Honeypot deployment approaches and their trade-offs.

Approach	Depth	Efficiency	Isolation
Low-interaction	No	Yes	Yes
High-interaction (dedicated VM)	Yes	No	Yes
Container-based	Yes	Yes	No

Our prior work introduced CATCH [7], a system that automatically deploys decoys and breadcrumbs based on network analysis and attacker movement patterns. CATCH motivates the need for a dynamic, on-demand deployment mechanism capable of creating realistic, high-interaction environments at attacker-triggered moments. However, CATCH

did not prescribe a concrete mechanism for safely instantiating high-interaction decoys with strong isolation and low startup latency. This work addresses that missing component by introducing a microVM-based isolation layer that enables CATCH-style dynamic decoy deployment. By spawning Firecracker microVMs on demand, the system provides the realism and isolation required for high-interaction deception while achieving the responsiveness needed for dynamic deployment. It specifically addresses Secure Shell (SSH) based terminal access scenarios. Advanced attackers who have obtained valid credentials—whether through insider threat, credential theft, or phishing—frequently conduct post-compromise operations through interactive terminal sessions, using command-line tools for lateral movement, data exfiltration, and persistence establishment. The focus is on what happens after an attacker gains access and begins executing commands in this terminal environment.

This paper makes three contributions:

1) An architecture for on-demand honeypot deception that spawns dedicated microVMs when threat detection triggers, performs asynchronous VM preparation, and provides transparent session redirection.
2) A prototype implementation demonstrating the architecture with PostgreSQL, chosen because database access represents a high-value target and requires handling both single-command execution and interactive session modes.
3) Quantitative benchmarks measuring VM startup phases, memory overhead, legitimate user latency, and timing-based detectability.

II. Related Work

A. Deception and Dynamic Decoy Deployment

Classical honeypots fall into low-, medium-, and high-interaction categories, each offering different trade-offs between realism, complexity, and resource cost. Low- and medium-interaction systems emulate services or protocol behaviours, but extensive empirical analyses show they are easily fingerprinted at Internet scale [4], making them unsuitable for sophisticated intruders. High-interaction honeypots provide realism by running genuine operating systems and services, but traditional VM-based deployments require substantial resources and cannot be instantiated on demand. CATCH [7] introduced the concept of dynamic, behaviour-triggered decoy and breadcrumb deployment, driven by network analysis and attacker-path prediction. However, CATCH leaves unsolved the key systems question of how to safely instantiate high-interaction decoys at attacker-triggered moments. This work provides such a mechanism by realising dynamic decoy instantiation via microVM snapshots.

B. Honeypot Fingerprinting

A large-scale scan of 3.3 billion IP addresses identified 7,605 honeypot instances across nine implementations with high accuracy [4]. The techniques exploited differences between honeypot libraries and the servers they emulate. A subsequent study detected 21,855 honeypot instances using a multistage fingerprinting framework [8]. These studies indicate that protocol emulation creates detectable inconsistencies regardless of implementation care.

C. Container Security Limitations

Analysis of Docker-based honeypot implementations identified fundamental isolation weaknesses [5]. Containers share the host kernel, creating attack vectors. Information leakage through `/proc` exposes filesystem structure, processor information, and kernel version. The authors conclude that "isolation at the hypervisor level" represents best practice for honeypot infrastructure targeting sophisticated attackers.

D. MicroVM Technology

Firecracker is a virtual machine monitor (VMM) developed by Amazon Web Services (AWS) for serverless computing [6]. Unlike general-purpose hypervisors, Firecracker implements a minimal device model, booting Linux guests in under 125 ms with memory overhead below 5 MB. The system uses the Kernel-based Virtual Machine (KVM) for hardware virtualisation, inheriting the security properties of hardware-enforced isolation. Amazon deploys Firecracker in Lambda and Fargate, supporting millions of production workloads.

MicroVM technology combines the strong isolation of hardware virtualisation with the low overhead previously associated only with containers, making it a natural candidate for honeypot workloads where both properties are essential.

E. On-Demand Honeypot Provisioning

The concept of dynamically provisioning honeypots per attacker was introduced by HoneyCloud [9], which assigned each attacker a new VM instance. However, traditional virtualisation imposed significant overhead: HoneyCloud could only restore home directories between sessions, and system modifications and uptime consistency remained limitations.

F. Research Gap

MicroVM technology has not been applied in the honeypot context. Existing high-interaction architectures rely on pre-provisioned virtual machines rather than on-demand spawning triggered by attacker behaviour. Recent work has explored container-based honeypots for cloud environments [10] and Large Language Model (LLM) powered responses [11], but neither addresses the isolation-efficiency trade-off through lightweight virtualisation. This paper bridges these gaps by implementing and evaluating behaviour-triggered microVM isolation.

III. System Architecture

A. Scope and Threat Model

This work focuses on the isolation architecture that executes after threat detection, not the detection mechanism itself. The system assumes an external threat detection capability exists, whether honeytokens, behavioural analytics, or other indicators of compromise. Our prototype uses honeytokens

as a concrete trigger mechanism because they provide deterministic, low-false-positive signals suitable for demonstrating the architecture. In a production deployment, the isolation layer would integrate with frameworks such as CATCH [7], which provides the network-analysis-driven detection and breadcrumb placement that triggers decoy instantiation.

All commands executed within isolated sessions are recorded with their execution mode and timing metadata, supporting analysis and auditing.

B. Overview

The system operates as an SSH server that accepts connections, monitors command execution for suspicious patterns, and transparently redirects detected attackers into isolated microVM environments. Each session progresses through one of three execution modes:

Real mode: Commands execute directly on the host system. This is the default for all sessions. Legitimate users remain in real mode throughout their session, experiencing no overhead from the deception infrastructure.

Intercepted mode: When a command matches a honeytoken pattern, the system returns hardcoded fake output immediately without executing the command. Simultaneously, a background task begins preparing an isolated environment.

Isolated mode: Commands execute inside a Firecracker microVM. Once a session enters isolated mode, it remains isolated. The isolated environment contains deceptive data designed to engage attackers while protecting the real system.

Fig. 1 shows the overall architecture: the SSH proxy mediates all connections, forwarding legitimate traffic to the host shell (real mode) while redirecting detected attackers to per-attacker microVMs (isolated mode). Fig. 2 traces the per-command decision logic, showing how the system checks whether a session is already isolated, evaluates honeytoken patterns for new sessions, and branches into the appropriate mode.

The remaining subsections focus on the isolated execution path because it constitutes the architectural contribution. Real mode requires no additional infrastructure, and intercepted mode is a transient state that serves only to bridge the gap while a microVM is being prepared.

C. Honeytoken-Triggered Isolation

Honeytokens provide a deterministic trigger mechanism. Unlike behavioural analysis, which requires interpreting command sequences to infer intent, honeytokens offer a clear signal: legitimate users have no reason to access them. The implementation defines honeytokens as regular expression patterns matched against incoming commands, targeting credential theft activities (MITRE ATTACK[1] technique T1552.001: Unsecured Credentials in Files).

Table II lists the five honeytoken categories implemented in the prototype, each targeting a different credential-discovery vector within T1552.001 and T1083.

[1]MITRE Adversarial Tactics, Techniques, and Common Knowledge framework.

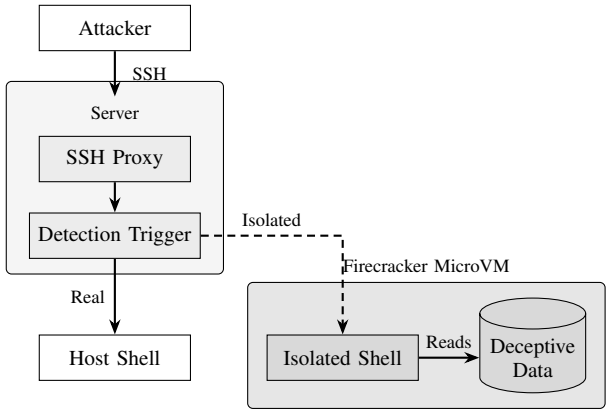

Fig. 1. System architecture. The SSH proxy mediates all connections. Legitimate users are forwarded to real infrastructure. When an attacker triggers a honeytoken, a dedicated Firecracker microVM is restored from a snapshot and used for subsequent commands.

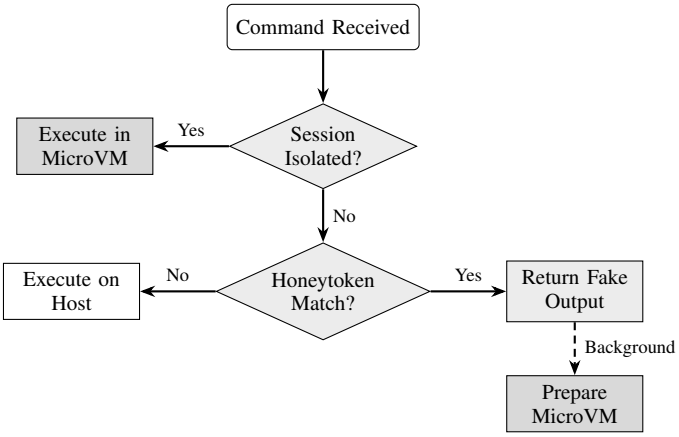

Fig. 2. Command execution flow. Each command is checked against honeytoken patterns. On first match, the system returns fake output and begins asynchronous microVM preparation. Subsequent commands from the attacker execute inside the isolated microVM.

TABLE II
HONEYTOKEN TRIGGER CATEGORIES AND REPRESENTATIVE COMMAND PATTERNS.

Trigger	Example Command	Technique
Database credentials	`cat ~/.pgpass`	T1552.001
Password hashes	`cat /etc/shadow`	T1552.001
SSH private keys	`cat ~/.ssh/id_rsa`	T1552.001
Recursive secret search	`grep -r password /`	T1552.001
Sensitive file discovery	`find / -name *.pem`	T1083

Each trigger returns contextually realistic fake output: truncated SSH private keys, bcrypt password hashes, `.pgpass` entries with credentials pointing to the isolated VM's database, and directory listings referencing further honeytokens. This layered design encourages attackers to follow the planted breadcrumbs deeper into the deceptive environment.

To illustrate the full session lifecycle, consider the evaluated attack scenario. An attacker begins with reconnaissance commands (`whoami`, `uname -a`, `ls -la ~`),

which execute in real mode against the genuine host. The attacker then reads `~/.pgpass`, triggering intercepted mode: the system immediately returns fake credentials (`hospital_user:Hospital98323`) while asynchronously preparing a microVM in the background. The attacker connects to PostgreSQL using the discovered credentials, at which point the system verifies that the connection parameters match the planted values and routes the session into the now-ready isolated VM. Subsequent queries execute against the VM's PostgreSQL instance, which contains fabricated patient records indistinguishable in schema and format from the real database.

D. Asynchronous VM Preparation

VM preparation can begin as soon as isolation is triggered, not when the next command arrives. When a honeytoken match occurs, the system schedules VM preparation to run concurrently with the current request handling. The honeytoken command returns its fake output immediately while the background task proceeds through network setup, snapshot restoration, and SSH establishment.

Subsequent commands check whether the preparation task has completed. If the VM is ready, execution proceeds immediately. If preparation is ongoing, the command waits. In practice, attackers often pause between commands, allowing preparation to complete before the next command is entered.

E. Snapshot Restoration

Cold-booting a virtual machine takes seconds, which is unacceptable for on-demand isolation. The implementation uses snapshot restoration instead. A snapshot captures the complete execution state of a running VM. Restoring a snapshot resumes execution from exactly where it was paused, bypassing the boot sequence entirely. Firecracker exposes a RESTful Application Programming Interface (API) for VM management, accepting snapshot paths and network configuration for restoration.

F. State Replay

When a session transitions from real to isolated mode, the isolated environment starts from a clean snapshot with no knowledge of prior commands. To maintain consistency, the system can track commands executed in real mode and replay them in the isolated environment. If an attacker ran `cd /var/log` before triggering isolation, subsequent commands execute in that directory. Replay output is discarded; it exists only to establish state.

State replay is optional and configurable: deployments can filter commands by program name, limit replay depth, or disable it entirely. This feature is included for deployments requiring session consistency but is not evaluated in this work; the core contribution focuses on the isolation mechanism itself.

Beyond command history, an attacker may observe other system properties during a session. Some—such as hostname, username, shell, timezone, and locale—can be configured during snapshot preparation to match the host environment. Others are constrained by the virtualisation layer: Firecracker uses its own guest kernel rather than the host kernel, so `uname -r` will differ, and guest memory is set at VM creation time (256 MB in our prototype) rather than reflecting the host's physical memory. Boot time also changes, since snapshot restoration effectively resets it. These differences are consistent with any virtualised environment and do not specifically indicate a honeypot, but deployments requiring tighter consistency can adjust the snapshot configuration accordingly.

G. Network Setup

Each microVM requires a virtual network tap (TAP) device for network connectivity. The host kernel manages each TAP device, exchanging Ethernet frames with the guest VM as a software-defined network port. Each isolated session receives a dedicated TAP device with a unique IP address, ensuring network isolation between concurrent attacker sessions. SSH daemon availability dominates VM startup time, accounting for approximately 85% of total latency. Network setup—TAP device creation and IP configuration—completes in under 20 ms.

IV. Evaluation

The evaluation addresses two research questions:

- **RQ1**: What overhead does on-demand isolation introduce, and are legitimate users affected?
- **RQ2**: Can attackers detect isolation through timing analysis?

Benchmarks use the PostgreSQL case study to evaluate the architecture. The core startup latency (network setup, snapshot restoration, SSH establishment) is application-independent; PostgreSQL-specific overhead is limited to protocol handling.

A. Experimental Setup

Benchmarks ran on a Dell XPS 15 with Intel Core i9-13900H and 32 GB RAM, running Arch Linux. The system uses Firecracker 1.14.0-dev with guest VMs allocated 256 MB RAM. A PostgreSQL container serves as the production database, while Firecracker microVMs contain isolated PostgreSQL instances with deceptive data.

We evaluate two primary scenarios. The *legitimate user* scenario executes eight commands per session—file listing, identity checks, and five PostgreSQL queries—none of which access honeytokens. The *attacker full chain* scenario follows MITRE ATTACK Atomic Red Team techniques for credential discovery (T1552.001): three reconnaissance commands, a honeytoken trigger (`cat ~/.pgpass`), and five database queries using the discovered credentials. Each scenario is run for 200 independent iterations. One iteration consists of a complete end-to-end session: establishing a fresh SSH connection, executing the scenario's full command sequence, and closing the session. Repeating each scenario 200 times provides sufficient data for robust statistical analysis. We report medians for typical performance and 95th percentiles for tail latency (the value below which 95% of all observed runs fall). Client-side timing captures observable latency, while

server-side instrumentation records VM startup phase durations.

B. RQ1: Isolation Overhead

TABLE III
VM STARTUP PHASE TIMING OVER 200 ITERATIONS.

Phase	Median	95th Percentile	Notes
Network Setup	17 ms	18 ms	TAP device creation
Firecracker Start	1 ms	1 ms	Process spawn
Snapshot Restore	18 ms	22 ms	Memory state load
SSH Ready	205 ms	215 ms	Daemon availability
Shell Init	1 ms	1 ms	Terminal allocation
Total	**242 ms**	**253 ms**	

1) VM Startup Latency: Table III shows timing statistics for each startup phase. SSH daemon availability dominates startup time at 85% of total latency. Firecracker process spawning and snapshot restoration are fast: the hypervisor starts in under 1 ms, and memory state loads in approximately 18 ms. Network setup—TAP device creation and IP configuration—completes in 17 ms. The resulting distribution is unimodal with low variance (95th percentile within 5% of median).

2) Memory Consumption: Per-VM memory consumption averages 106 MB, 41% of the 256 MB guest allocation. A host with 8 GB available could support approximately 75 concurrent isolated attackers.

3) Legitimate User Impact: Legitimate user scenarios execute commands that never access honeytokens. Across 1,600 commands (8 per session $\times$ 200 iterations), median latency is 33 ms with a 95th percentile of 41 ms. Isolation is never triggered. These results confirm zero isolation overhead for non-attackers: no VMs are started and no snapshots are restored. The proxy layer introduces baseline latency for command forwarding, but this is constant regardless of whether the deception infrastructure is active.

4) Scalability: The system was tested with up to 100 concurrent attacker sessions, each started 100 ms after the previous one to simulate realistic arrival patterns. Table IV summarises the results.

TABLE IV
CONCURRENT SESSION SCALABILITY. SESSIONS STARTED 100 MS APART.

Sessions	Success	Mean (s)	95th Percentile (s)
10	100%	6.6	27.4
25	100%	6.2	31.8
50	100%	2.6	2.9
100	100%	3.8	3.7

All concurrency levels achieve 100% success with no connection, isolation, or timeout failures. Since each VM takes approximately 242 ms to start and sessions arrive every 100 ms, a small batch of 10 sessions results in several VMs being prepared simultaneously, competing for CPU and I/O. With larger batches, early sessions finish before later ones arrive, so fewer VMs are active at any given moment. This explains the lower mean and 95th percentile latency at 50 and 100 sessions compared to 10 and 25. Memory is the practical limit: at 106 MB per VM, a host with 8 GB available supports approximately 75 concurrent isolated sessions.

C. RQ2: Timing-Based Detectability

Table V compares query latency between the legitimate user scenario (real database) and the attacker full chain scenario (isolated VM, post-isolation queries).

TABLE V
QUERY LATENCY COMPARISON: REAL DB VS ISOLATED VM

Query	Real DB	Isolated	Delta
`\dt` (post-isolation)	38 ms	476 ms	+438 ms
`SELECT 1;`	35 ms	31 ms	−4 ms
`SELECT COUNT(*) FROM patients;`	36 ms	31 ms	−5 ms
`SELECT * FROM patients LIMIT 10;`	35 ms	31 ms	−4 ms
`SELECT diagnosis, COUNT(*)...`	36 ms	31 ms	−5 ms

The first query after isolation shows a latency increase (+438 ms) because the benchmark issues commands in rapid succession, arriving before VM preparation completes. This latency includes VM startup (242 ms) and initial PostgreSQL connection establishment inside the VM. Subsequent queries show the isolated VM performing 3–5 ms faster than the real database due to localhost networking inside the VM versus inter-container networking for the production database.

Importantly, the trigger command itself introduces no timing anomaly. When a honeytoken match occurs, the system serves pre-prepared fake output directly from memory without executing the command or waiting for VM preparation. The attacker receives an immediate, realistic-looking response—no slower than any previous command in the session. VM preparation proceeds in the background while the attacker inspects the returned output. The only observable latency increase occurs if the attacker issues a subsequent command before VM preparation finishes, as shown by the first-query delta in Table V.

For an attacker to detect isolation, they would need to issue a command immediately after the trigger (within approximately 250 ms), measure its latency precisely, and have prior knowledge of expected baseline latency. After the first post-isolation command, timing provides no detection signal. Any pause between the trigger and the next command allows VM preparation to complete undetected.

V. PERFORMANCE, EFFORT, AND SCOPE ANALYSIS

A. Performance Comparison

Table VI presents startup latency figures from prior work as reference points. Firecracker's 125 ms measures only kernel initialisation with no services. Our 242 ms includes snapshot restoration, SSH daemon readiness, and shell initialisation: a complete startup sequence ready to handle attacker commands. This is faster than Docker container startup (300 ms) while

TABLE VI
STARTUP LATENCY REFERENCE POINTS FROM LITERATURE. VALUES ARE FROM DIFFERENT HARDWARE AND METHODOLOGIES; DIRECT COMPARISON IS INDICATIVE ONLY.

System	Startup	Isolation
Docker container [12]	300 ms	Shared kernel
Firecracker (bare boot) [6]	125 ms	Full VM
Container (cold start) [13]	7500 ms	Shared kernel
Container (checkpoint/restore) [13]	300–400 ms	Shared kernel
This system	242 ms	Full VM

providing full VM isolation rather than shared-kernel containers.

B. Implementation Effort

The architecture requires protocol-specific handlers for applications with complex I/O patterns. Simple command-line programs that follow standard input/output conventions work without special handling. PostgreSQL was chosen as the case study precisely because it represents a challenging case: it supports both single-command execution (`psql -c "SELECT..."`) and interactive sessions with stateful prompts, requiring the handler to manage both modes. Programs with complex terminal interactions (e.g. `top`, `vim`) would require additional handlers, but the PostgreSQL case study demonstrates that the architecture accommodates demanding I/O patterns.

C. Evaluation Scope

All benchmarks ran on a single host, with the SSH proxy, database, and microVMs sharing the same physical machine. Production deployments with network-separated services would introduce additional latency not captured here. The query workload used simple operations; complex queries or transaction-heavy workloads may behave differently. All measurements used automated benchmark clients executing commands in rapid succession. Human attackers with variable typing speeds and pauses between commands would provide additional time for VM preparation to complete undetected.

VI. CONCLUSION AND FUTURE WORK

This paper presented an on-demand microVM isolation architecture for dynamic decoy deployment, demonstrating that Firecracker-based deception achieves sub-second startup latency while maintaining VM-level security isolation. By providing a lightweight mechanism for spawning realistic, per-attacker decoys that run genuine operating systems and services, this work delivers the dynamic deployment capability required for adaptive deception systems such as CATCH. Future work includes extending protocol support to additional interactive applications and distributed deployment across multiple hosts.

REFERENCES

[1] Verizon, "2025 Data Breach Investigations Report," 2025. [Online]. Available: https://www.verizon.com/business/resources/reports/2025-dbir-data-breach-investigations-report.pdf

[2] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero Trust Architecture," National Institute of Standards and Technology, NIST Special Publication 800-207, Aug. 2020. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf

[3] J. Franco, A. Aris, B. Canberk, and A. S. Uluagac, "A survey of honeypots and honeynets for Internet of Things, Industrial Internet of Things, and Cyber-Physical Systems," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 4, pp. 2351–2383, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9520645

[4] A. Vetterl and R. Clayton, "Bitter Harvest: Systematically fingerprinting low- and medium-interaction honeypots at Internet scale," in *Proc. USENIX Workshop on Offensive Technologies (WOOT)*, Aug. 2018. [Online]. Available: https://www.repository.cam.ac.uk/handle/1810/280555

[5] D. Sever and T. Kišasondi, "Efficiency and security of Docker based honeypot systems," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2018, pp. 1167–1173. [Online]. Available: https://ieeexplore.ieee.org/document/8400212

[6] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *Proc. 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020, pp. 419–434. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/agache

[7] P. K. Chouhan, M. Colombo, R. Asal, and Z. Cui, "CATCH: A tool to automatic deploy decoys and breadcrumbs based on network analysis," in *2025 5th Intelligent Cybersecurity Conference (ICSC)*. IEEE, 2025.

[8] S. Srinivasa, J. M. Pedersen, and E. Vasilomanolakis, "Gotta catch 'em all: A multistage framework for honeypot fingerprinting," *Digital Threats: Research and Practice*, vol. 4, no. 3, pp. 1–28, Sep. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3584976

[9] P. Clemente, J.-F. Lalande, and J. Rouzaud-Cornabas, "HoneyCloud: Elastic honeypots – on-attack provisioning of high-interaction honeypots," in *Proc. International Conference on Security and Cryptography (SECRYPT)*, 2012, pp. 434–439.

[10] V. S. Devi Priya and S. Sibi Chakkaravarthy, "Containerized cloud-based honeypot deception for tracking attackers," *Scientific Reports*, vol. 13, 2023. [Online]. Available: https://doi.org/10.1038/s41598-023-28613-0

[11] Z. Wang, J. You, H. Wang, T. Yuan, Y. Wang, S. Lv, and L. Sun, "HoneyGPT: Breaking the trilemma in honeypots with large language models," *arXiv preprint arXiv:2406.01882*, 2024. [Online]. Available: https://arxiv.org/abs/2406.01882

[12] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale: A comparative study," in *Proc. 17th International Middleware Conference*, ser. Middleware '16. New York, NY, USA: Association for Computing Machinery, Nov. 2016, pp. 1–13. [Online]. Available: https://doi.org/10.1145/2988336.2988337

[13] S. Nadgowda, S. Suneja, and A. Kanso, "Comparing scaling methods for Linux containers," in *2017 IEEE International Conference on Cloud Engineering (IC2E)*, Apr. 2017, pp. 266–272. [Online]. Available: https://ieeexplore.ieee.org/document/7923811