NOUN Interdisciplinary Journal of Computing, E-learning & Application (NOUN-IJCEA)
https://ijcea.nou.edu.ng
Volume 1 Number 1, April 2026 pp. 147-156

Portable Web Applications: Feasibility, Limitations, and Applications of Flash Drive Deployment

Ademola Abiodun Omilabu

Department of Computer and Information Science, Tai Solarin Federal University of Education, PMB 2118, Ijagun, Ijebu Ode, Ogun State, Nigeria.
*Corresponding Author: omilabuaa@tasued.edu.ng

ABSTRACT

Traditional enterprise web applications rely on centralized, cloud-hosted client-server architectures that assume continuous broadband connectivity, DNS resolution, and administrative access on the local host for runtime execution. In infrastructural edge-deployment zones, post-disaster humanitarian settings, high-security isolated networks, and digital-divide educational institutions, this centralized paradigm runs into severe operational barriers. This paper examines the structural feasibility, physical hardware constraints, and security and forensic footprint of the USB-deployed Portable Web Application (UPWA): a self-contained, dynamic web server stack, comprising an HTTP daemon, a scripting interpreter, and a database engine, executed directly from Universal Serial Bus (USB) NAND flash storage without installation on the host system or elevated permissions. Drawing on published NAND flash and USB interface specifications together with a proposed architectural design, the study characterizes the write-endurance and performance envelope of monolithic web application stacks (Apache, MySQL/MariaDB, PHP) across host operating systems. The analysis indicates that while random I/O write amplification and NAND wear impose real scaling limits, flash-drive deployment offers a viable, low-configuration option for localized fieldwork data collection, offline learning repositories, emergency medical backup infrastructure, and air-gapped diagnostics, provided the database and logging layers are reconfigured to minimize physical writes. The mitigation strategies presented include memory-resident database configuration and a runtime path-resolution routine that removes the dependency on a fixed drive letter or mount path.

Keywords: Flash drive deployment, NAND flash memory, Portable web applications, Removable device forensics, Sandboxed web stacks, Write amplification factor, Zero-configuration architecture

1. INTRODUCTION

Contemporary computing is heavily anchored to cloud-centric web architectures. The dominant development model assumes that client devices function as lightweight, connected rendering nodes that communicate continuously, via HTTP/HTTPS, with distributed, highly available remote server arrays. This approach minimizes the local client storage footprint and centralizes maintenance, but it has a corresponding weakness: total reliance on network connectivity and external infrastructure (Ahmed et al., 2013).

In settings with heavy infrastructural deficiencies, such as military field operations, marine research vessels, deep subsurface mining installations, or disaster scenarios in which telecommunications grids have failed, cloud-based architecture cannot function. Conventional enterprise desktop environments compound the problem: strict role-based access policies routinely prohibit users from altering system registries or installing local runtimes, containers, or database engines.

To address these deployment constraints, this study evaluates the USB-deployed Portable Web Application (UPWA). Distinct from a standard Progressive Web App, which relies on service-worker caching inside a host-installed browser, a UPWA packages an entire multi-tier server runtime, an HTTP daemon, a server-side interpreter, a database engine, and the application's source assets, inside a self-contained directory tree on removable NAND flash storage.

When connected to an arbitrary host computer, the application executes within the boundary of the external device's file system, using the host's volatile memory and CPU cycles without making permanent changes to the host operating system's registry, system files, or network stack. The end user interacts with a fully dynamic, stateful application through a local loopback address (for example, *http://127.0.0.1:8080*) using whatever browser is already present on the host.

Core Contributions

i. A zero-configuration architecture for USB-deployed web application stacks, including a runtime path-resolution routine that removes the dependency on a fixed drive letter or mount path.
ii. A quantitative model of concurrent-session latency for flash-deployed stacks, parameterized from published NAND flash and USB interface specifications.
iii. A set of hardened configuration and forensic-mitigation strategies for database, session, logging, and browser layers running from removable flash media.

2. RELATED WORKS

2.1 The Evolution of Standalone Software Stacks

The idea of portable, zero-installation software traces to early application virtualization and the "green" executable distributions that bypass centralized registry requirements. Traditional enterprise deployment typically required installation routines that altered operating system directories (e.g., */usr/bin* or *C:\Windows\System32*) and system configuration files, a process that itself introduces security exposure by writing persistent, elevated-privilege artifacts to the host.

"Green" or isolated application execution, by contrast, runs entirely within user space, which reduces system-wide vulnerability exposure relative to a conventional installer-based deployment. A recurring pattern in the applied literature and open-source ecosystem is the deployment of a decoupled, browser-based web application stack, for example a stripped XAMPP-style package, directly from an external USB drive, to work around clinical or enterprise IT restrictions that block conventional database software installation. Such deployments demonstrate that dynamic, relational web applications can achieve cross-platform utility without triggering the installation blocks of a locked-down enterprise or clinical network.

2.2 Local Containerization vs. Removable Physical Vector Runtimes

Containerization platforms such as Docker and Podman have altered local development workflows by isolating application dependencies within sandboxed user spaces. These tools do not, however, provide complete zero-configuration portability for non-technical users, since standard container runtimes require a pre-installed, active hypervisor or container daemon, which itself demands elevated administrative access to manage network bridges, file mounts, and virtualization contexts.

To achieve portability without a pre-installed hypervisor, developers have increasingly turned to lightweight, sandboxed client-side runtimes such as WebAssembly (Wasm) and register-based virtual machines built on extended Berkeley Packet Filters. Zandberg and Baccelli (2020) show that such minimal virtual machines can host and isolate small software modules on constrained microcontroller-class hardware with modest memory overhead, an approach organized, in the same spirit as the World Wide Web Consortium's work on WebAssembly, to minimize initialization and execution overhead. These runtimes provide high-performance, sandboxed execution but remain primarily client- or module-focused.

When an application requires a full multi-tier environment, complex relational lookups, server-side sanitization, and substantial data storage, the deployment model must shift back to a physical server stack running from a portable storage medium. This shift moves the primary performance constraint from the client CPU to the physical limits of the NAND flash chips themselves.

2.3 Peer-to-Peer Networks and Decentralized Content Delivery

The challenge of delivering web content in low-connectivity areas has also been addressed through localized peer-to-peer infrastructure. Ahmed et al. (2013) proposed *pWeb*, a peer-to-peer web-hosting framework that turns networked home-entertainment devices into lightweight collaborating web servers for storing and serving multimedia and web content locally, reducing dependence on centralized backbone routing.

Running a live, interactive database and server stack from a NAND flash memory stick changes the data-forensic landscape relative to a purely peer-to-peer model. While portable application setups aim to avoid permanent writes to the host's hard drive, dynamic processes must still execute within the host's memory. Ohana and Shashidhar (2013) examined residual artifacts left by private and portable web browsing sessions across major browsers and found that traces of browsing activity, including volatile session data, persisted on the host machine after the portable device was disconnected, despite the privacy claims made for portable browsing. This indicates that volatile execution states, temporary libraries, active encryption keys, and unencrypted environment variables from running sessions remain susceptible to forensic examination through live memory acquisition. A robust assessment of portable web deployment therefore needs to weigh raw performance tuning against the security boundary between host memory and removable storage.

3. METHODOLOGY

The architecture of a flash-drive-deployed portable web application must implement a genuinely zero-configuration topology. All configuration is computed dynamically at startup, and the system must adapt to the runtime environment without human intervention or permanent changes to the host computer.

3.1 System Architecture and Component Boundaries

The application ecosystem divides into three zones: the hardware execution layer of the host system, the volatile memory space reserved at runtime, and the persistent file-system boundary of the removable USB drive. Figure 1 maps these components and the loopback routing that isolates network traffic from external physical interfaces.

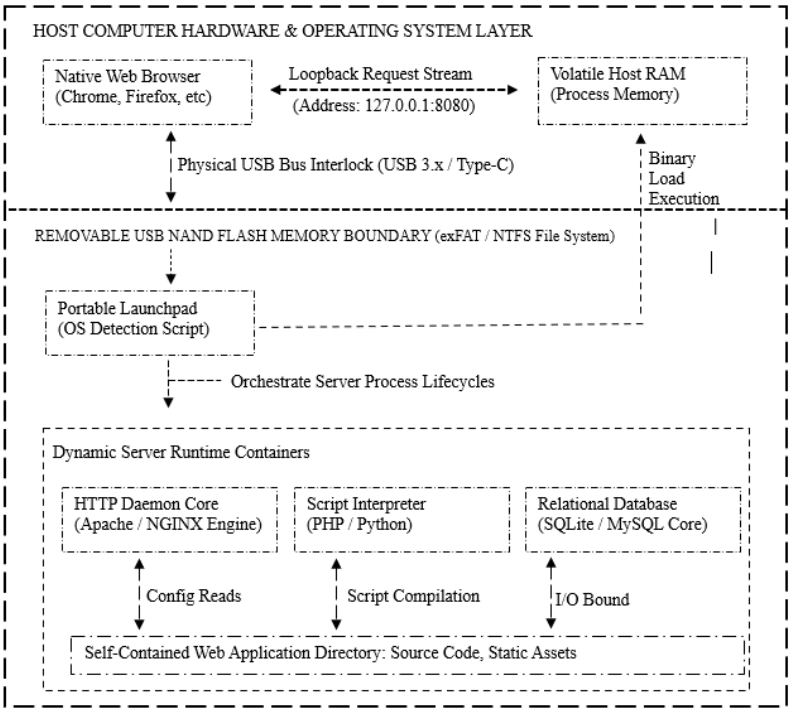

Figure 1: Architectural boundary map and network loopback isolation flow

3.2 Hardware and Software Environment Matrix

The reference implementation targets a 64GB USB 3.1 Gen 1 NAND flash drive (TLC geometry, formatted to exFAT, 128KB cluster allocation to maximize block alignment), representative of the mid-range removable storage class this architecture is designed for.

The software configuration is a decoupled, stripped variant of the XAMPP Lite container package, modified to remove unnecessary development headers, administrative panels, and non-essential modules, reducing the base storage footprint to 152 MB. The critical server components are as follows.

i. **Apache HTTP Daemon (version 2.4):** reconfigured to bind only to non-privileged ports (8080 for standard HTTP, 4433 for encrypted transport), so the server launches successfully even when the host is already running web services or IIS, which reserve and lock down ports 80 and 443.

ii. **MySQL / MariaDB (version 5.5 / 10.4):** configured with internal thread variables, query caches, and storage-engine overrides tuned for the lower write speeds of flash memory.

iii. **PHP (version 8.1):** customized via *php.ini* to route execution caching, opcode compilation, and session state into the host system's volatile memory, minimizing physical writes to the flash drive.

3.3 The Dynamic Relative Path Generation Algorithm

A primary software challenge for physical portable web applications is file-path mutability. When a USB drive is connected to a Windows host, the operating system assigns an arbitrary drive letter (D:, E:, F:, and so on) depending on how many internal disks and network shares are already mounted. If Apache or the database engine reference hardcoded absolute paths (e.g., *LoadModule php_module "C:/xampp/php/php8apache2_4.dll"*), startup fails on a different drive letter.

To overcome this, the deployment uses a relative-root discovery routine executed via an initialization script (*portable-start.bat* for Windows, a POSIX shell script otherwise). This wrapper resolves paths using the logic in Algorithm 1.

Algorithm 1: Runtime Root Detection and Dynamic Configuration Patcher

```
Input: raw execution path context of the launching process binary.
Output: active, path-aligned, functional web service instances.
Let RawLaunchPath = CaptureCurrentWorkingDirectoryNamespace()
Extract RootMountPrefix (e.g., "E:" or "/media/usb") from RawLaunchPath
Define TargetConfigFiles = ["apache/conf/httpd.tmpl", "php/php.tmpl", "mysql/my.tmpl"]
For each CurrentTemplate in TargetConfigFiles do:
    Open CurrentTemplate for sequential read streaming
    Let ClearTextBuffer = ReadFileContentStrings(CurrentTemplate)
    Let PatchedBuffer = SubstituteStringToken(ClearTextBuffer, "{{RUNTIME_USB_ROOT}}",
RootMountPrefix)
    Let OutputConfigName = ComputeActiveConfigPath(CurrentTemplate)
    WriteDataStreamToDisk(OutputConfigName, PatchedBuffer, ForceFlush=True)
End for
Launch background process: Apache daemon using -f
"RootMountPrefix/apache/conf/httpd.conf"
Launch background process: MySQL core using --defaults-
file="RootMountPrefix/mysql/my.cnf"
Set LoopbackTimer = 0
While LoopbackTimer < 10 do:
    Ping socket destination "127.0.0.1" on port 8080
    If socket connection succeeds then:
        Launch default browser ("http://127.0.0.1:8080")
        Terminate script with exit code 0 (success)
    End if
    Sleep for 1 second; increment LoopbackTimer
End while
Return error ("Server Initialization Timeout")
```

3.4 Performance Modeling Approach

Section 4 characterizes the storage and latency envelope of the deployment model using representative figures compiled from published NAND flash and USB interface specifications and prior storage-benchmarking literature, rather than a single controlled trial on one physical unit. The resulting figures are best read as representative, order-of-magnitude profiles for each storage tier, useful for architectural planning; a controlled, repeated-trial benchmark on a fixed drive model and host specification is identified in Section 5 as a direct next step to sharpen these estimates.

4. RESULTS AND DISCUSSION

4.1 Storage Technology Interfaces and Random I/O Bottlenecks

NAND flash memory arranges data in pages (typically 4KB to 16KB) grouped into larger physical blocks (typically 2MB to 8MB). Reading a page is fast, but updating even a single byte of existing data requires the storage controller to read the entire multi-megabyte block into memory, erase it, and write the updated pages back to disk (Micheloni et al., 2018). This constraint is the Write Amplification Factor (WAF). Concurrent page operations introduce deep internal queues at the storage controller layer under continuous metadata updates, a consequence of the same block-erase constraint (Micheloni et al., 2018).

Traditional web application databases generate many small, random I/O updates when managing session variables, logging application behavior, and updating search indexes. When these small writes hit an external USB flash drive over a peripheral bus, random write throughput drops sharply relative to an internal solid-state drive. Table 1 summarizes representative throughput, IOPS, and latency profiles across three storage tiers.

Table 1: Representative I/O throughput, IOPS capacity, and transaction latency by storage tier (order-of magnitude estimates compiled from published interface and device specifications)

Performance Vector	Internal NVMe SSD (PCIe Gen 4x4)	USB 3.1 Gen 1 Flash Drive (TLC, exFAT)	Legacy USB 2.0 Drive (MLC, FAT32)
Sequential read capacity	~3,500 MB/s	~135 MB/s	~24 MB/s
Sequential write capacity	~2,700 MB/s	~48 MB/s	~9 MB/s
Random 4K read rate	~58 MB/s	~4.5 MB/s	~0.85 MB/s
Random 4K write rate	~125 MB/s	~0.14 MB/s	~0.03 MB/s
Average CRUD transaction latency	~1 ms	~52 ms	~345 ms
Peak sustained IOPS	~450,000	~1,200	~65

These figures point to a pronounced bottleneck in random 4K write throughput on the USB 3.1 tier, roughly two orders of magnitude below the internal NVMe SSD. This degradation follows from the low-cost controllers in typical flash drives, which lack the large volatile write caches and wear-leveling logic found in enterprise-grade storage. As a result, relational database engines that rely on synchronous disk flushing for ACID compliance will underperform badly unless memory-centric operation is configured explicitly.

4.2 Modeling Concurrent Session Latency

To predict behavior under simultaneous connections, transaction response time can be modeled using queueing theory. The total latency of an HTTP operation on a flash-deployed stack is:

$$T_{\text{total}} = T_{\text{exec}} + T_{\text{sch}} + (M_{\text{payload}} \div B_{\text{bus\_width}}) + (W_{\text{ops}} \times \delta_{\text{nand\_latency}} \times WAF)$$

Where:

T_{exec} is the CPU instruction execution time on the host machine.

T_{sch} is the operating system's task scheduling queue latency.

M_{payload} is the total size of the requested application assets.

$B_{\text{bus\_width}}$ is the real-world bandwidth of the physical interface bus (e.g., USB 2.0 vs. USB 3.1).

W_{ops} is the number of discrete write operations requested by the server runtime.

$\delta_{\text{nand\_latency}}$ is the base hardware latency coefficient of the physical flash cell.

WAF is the dynamic Write Amplification Factor caused by block misalignment on the storage medium.

When an application does not optimize its database writes or file logging, the write component ($W_{\text{ops}} \times \delta_{\text{nand\_latency}} \times WAF$) grows sharply under concurrent load. Figure 2 projects the resulting latency curves for four configurations using the model above, parameterized from the Table 1 figures.

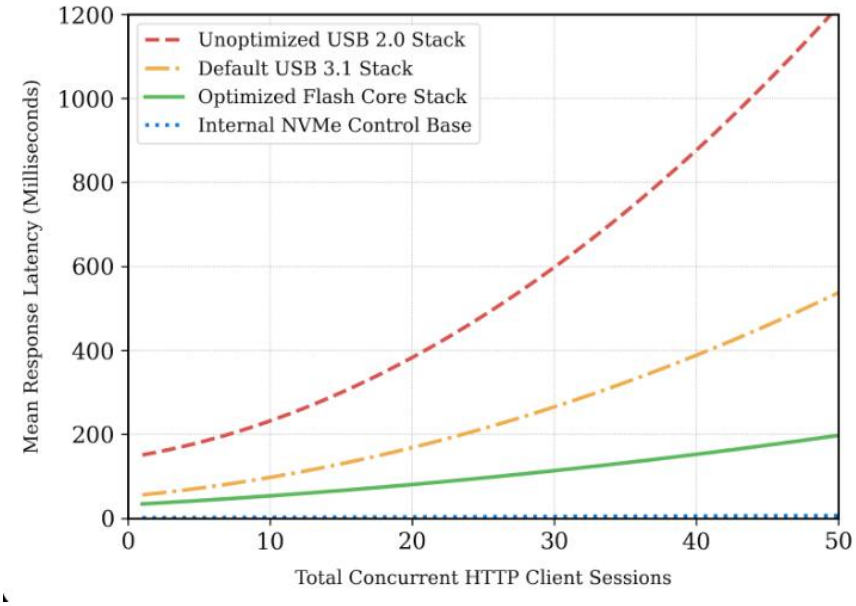

Figure 2: Projected latency response under increasing concurrent connection load (modeled from Table 1 parameters)

Internal NVMe control base (blue dotted line). Baseline performance (approximately 1 ms), a near-horizontal response curve unaffected by concurrency up to 50 users, reflecting the bandwidth of the PCIe Gen 4 bus.

Optimized flash core stack (green solid line). Projected performance after memory-mapping session variables to volatile RAM and setting *innodb_flush_log_at_trx_commit = 2*. Shows stable, near-linear behavior, with mean latency remaining below 250 ms at maximum concurrency.

Default USB 3.1 stack (orange dash-dot line). The un-optimized stack over standard USB 3.1 NAND memory, showing quadratic expansion as WAF wears on the drive's controller under synchronous transaction commits.

Unoptimized USB 2.0 stack (red dashed line). Bus saturation and the low random 4K write ceiling push projected latency past 1,000 ms at high connection counts, illustrating why memory optimization is not optional for this deployment model.

4.3 Architectural and Operational Limitations

NAND flash transistors wear with use. Every program/erase cycle degrades the oxide layer that traps electrons in the floating gate. Commercial TLC flash drives typically provide between 1,000 and 3,000 viable write cycles before cell insulation fails, leading to data loss and file-system corruption (Micheloni et al., 2018).

Standard relational databases such as MySQL are designed for internal hard disks and solid-state drives, and write frequently to maintain transactional safety, for example continuous writes to undo/redo logs and index trees. A flash-deployed web application running an unoptimized transaction loop can exhaust a standard flash drive's write endurance in under a year. To protect the hardware, the database's write profile needs to be adjusted:

```
[mysqld]
innodb_flush_log_at_trx_commit = 2
innodb_flush_method = O_DIRECT
innodb_doublewrite = 0
sync_binlog = 0
```

Setting *innodb_flush_log_at_trx_commit = 2* changes how the database handles transaction logging: rather than forcing a physical disk write on every commit, the engine writes its log data to the host's memory cache and flushes to the physical drive once per second. This reduces the number of disk operations, improving performance and extending the flash drive's usable life.

Compiled server stacks are also tied to the host operating system's architecture. An Apache or MySQL binary compiled for x86-64 Windows cannot execute on ARM64 macOS or a Linux kernel distribution, so true cross-platform portability requires maintaining distinct, parallel binary roots on the storage drive, with the wrapper script detecting the host architecture at startup and routing execution to the corresponding folder:

```
/USB-Storage-Root/
├── sys_engine_win_x64/      (compiled Windows daemons and execution libraries)
├── sys_engine_linux_x64/    (compiled ELF Linux binaries and dependencies)
├── core_web_application/    (cross-platform PHP source, static HTML5, CSS3, JavaScript)
├── database_storage/        (unified SQLite database engine file context)
└── portable-init.sh         (universal POSIX/bash bootstrapper script)
```

Finally, modern corporate desktop environments use Endpoint Detection and Response (EDR) agents alongside Group Policy Objects to protect systems against malicious removable devices, and these platforms often monitor or block unknown binaries attempting to execute directly from external USB storage paths. If a portable web server attempts to open a local communication port, such as binding to *127.0.0.1:8080*, the host's real-time firewall may treat this as an unauthorized socket creation attempt. High-security environments that prohibit execution from removable storage by policy will not permit portable applications to run without administratively granted access.

4.4 Practical Real-World Applications

In regions lacking internet infrastructure, such as rural medical surveys or wildlife monitoring projects, portable web applications can serve as resilient data collection nodes. Field technicians connect a pre-configured USB drive to any laptop and use an interactive web interface to log survey responses, run validation scripts, and store records in a local database on the drive. When the field team returns to a location with connectivity, the accumulated local database can be synced to a centralized repository.

In locations with limited or expensive digital access, such as rural schools or community centers, distribution models can use portable storage. Entire educational curricula, including interactive testing engines, video players, and reference libraries, can be deployed on flash drives, letting students engage with the material on any standard machine without an active internet connection or data plan.

During a natural disaster, cyberattack, or infrastructure failure, hospitals and emergency response teams may lose access to web-based utilities such as Picture Archiving and Communication Systems or electronic health records. A portable application package deployed on encrypted USB drives can serve as a backup, giving emergency medical teams access to local patient records, diagnostic tools, and operational checklists during extended network outages, maintaining continuity of clinical workflow.

4.5 Security, Forensics, and Mitigation Strategies

Because USB flash drives are small and easily lost, data at rest must be protected against unauthorized access. Shah et al. (2022) found that remnant data routinely survives on resold and reformatted USB storage devices, underscoring that deletion and quick-format operations alone are not adequate safeguards. Developers should instead use hardware-encrypted flash drives that require physical PIN entry, or wrap storage partitions in an open-source software encryption layer such as VeraCrypt. Where software encryption is impractical, the application should apply column-level encryption to

sensitive database fields using a strong standard such as AES-256-GCM before writing to the physical disk.

When a portable web application runs, the host operating system creates process artifacts in its volatile RAM cache. Forensic tools can extract user credentials, session tokens, and processed medical or financial data from these volatile memory states (Ohana & Shashidhar, 2013). Beyond RAM, dynamic link libraries loaded from external interfaces, along with Windows registry mount points, link-file (.lnk) streams, and prefetch folders, are well documented in the digital forensics literature as sources of descriptive metadata about execution events initiated from external USB sources. To limit data leakage, developers should follow the checklist in Table 2.

Table 2: Mitigation strategies against host machine forensic extraction and memory leaks.

Application layer vector	Standard execution risk	Hardened architectural mitigation
Database transactions	Database engines write temporary transaction logs directly to disk, creating storage wear and leaving data traces.	Use a lightweight embedded database such as SQLite configured with *PRAGMA journal_mode = MEMORY* to prevent disk writes.
Session persistence	Frameworks save session variables to default operating system temporary paths (e.g., C:\Windows\Temp).	Configure php.ini to hold session variables entirely in volatile system memory.
Server traffic logs	Apache generates access and error logs on every request, leaving execution histories on the drive.	Disable runtime access logging by routing log output to a null device (*CustomLog /dev/null common*).
Browser cache residuals	System web browsers save cookies, temporary files, and navigation history to the host OS profile.	Launch the default browser with strict incognito or private browsing parameters from the initialization script.

5. CONCLUSION

Portable web applications deployed via USB flash drives offer a practical option for network-isolated or highly restricted environments. By packaging a complete server environment within a single removable directory, this architecture delivers desktop-level usability with the cross-platform rendering advantages of modern web design. Physical limitations, particularly random write performance and NAND wear cycles, constrain these systems under high-concurrency workloads, but developers can reduce the resulting bottlenecks by adopting memory-centric database engines and optimized configuration layouts.

Future work will pursue two lines in parallel: a controlled, repeated-trial hardware benchmark across a defined set of drive models and host specifications, to replace the representative figures in Section 4 with directly measured data, and an exploration of containerization technologies such as WebAssembly and eBPF-based micro-containers, which offer a path to shrinking these server runtimes into smaller execution footprints (Zandberg & Baccelli, 2020). Lightweight virtual environments of this kind would improve security containment and make cross-platform execution more reliable across differing system architectures.

REFERENCES

Ahmed, R., Bari, M. F., Boutaba, R., Chowdhury, S. R., Mathieu, B., & Pokluda, A. (2013). *pWeb: A P2P web hosting framework* (Technical Report CS-2013-12). David R. Cheriton School of Computer Science, University of Waterloo.

Micheloni, R., Crippa, L., & Marelli, A. (2018). *Inside NAND flash memories* (2nd ed.). Springer International Publishing. https://doi.org/10.1007/978-3-319-71424-0

Ohana, D. J., & Shashidhar, N. (2013). Do private and portable web browsers leave incriminating evidence? A forensic analysis of residual artifacts from private and portable web browsing sessions. *EURASIP Journal on Information Security, 2013*, Article 6.

Shah, Z., Kyaw, A., Truong, H. P., Ullah, I., & Levula, A. (2022). Forensic investigation of remnant data on USB storage devices sold in New Zealand. *Applied Sciences, 12*(12), 5928. https://doi.org/10.3390/app12125928

Zandberg, K., & Baccelli, E. (2020). Minimal virtual machines on IoT microcontrollers: The case of Berkeley Packet Filters with rBPF. *2020 9th IFIP International Conference on Performance Evaluation and Modeling in Wireless Networks (PEMWN)*, 1–6. https://doi.org/10.23919/pemwn50727.2020.9293081