Netcluster: A Lightweight LAN-Based Distributed Cybersecurity Toolkit With PIN-Based Cluster and Resource Management Using Ray Framework

Shailesh G. Thakare[1], Sanika R. Patil[2], Ankush L. Patil[3], Dr. Pushpalata Aher[4]
[1,2,3]*Student, Department of Computer Science and Engineering, Sandip University, Nashik, India*
[4]*Professor, Department of Computer Science and Engineering, Sandip University, Nashik, India*

***Abstract*—Cybersecurity operations such as password cracking and brute-force attacks demand large amounts of computational power, especially when working with massive password wordlists and complex key spaces. Single-machine approaches simply cannot keep up with this scale in a reasonable amount of time. This paper presents NetCluster, a lightweight distributed cybersecurity toolkit built for local area network (LAN) environments using the Ray framework. The system introduces a PIN-based cluster connection mechanism that allows controlled and secure participation of nodes, and also supports user-defined CPU allocation for flexible resource management. Unlike traditional broadcast-based systems, NetCluster can support multiple independent clusters running within the same network without interference. We evaluated the system using the rockyou.txt dataset and found that distributing the workload across nodes significantly cuts down execution time, with near-linear speedup observed for small cluster configurations. Overall, NetCluster is easy to deploy, scalable, and well-suited for academic and research use cases.**

***Index Terms*—Cluster Management, Cybersecurity Toolkit, Distributed Computing, Hash Cracking, Parallel Processing, Ray Framework, Resource Allocation.**

I. INTRODUCTION

The growing complexity of modern passwords and the sheer size of password datasets have made cybersecurity tasks like password cracking and brute-force attacks extremely resource-intensive. A typical attack might involve iterating through tens of millions of possible combinations, and doing that on a single machine just takes too long to be practical. This is a common challenge in educational and research cybersecurity environments.

Distributed computing is one way to tackle this problem. The idea is simple—split the task into smaller pieces and run them in parallel across multiple machines. Frameworks like Hadoop and Apache Spark have been widely used for this purpose in enterprise settings, but they come with significant setup cost. Configuring these systems requires dedicated infrastructure and expertise that are often unavailable in academic labs or small research groups. This paper introduces NetCluster, a lightweight and easy-to-use distributed cybersecurity toolkit designed specifically for LAN environments. The system uses the Ray framework, which is a modern Python-based distributed computing library that handles task scheduling and resource management efficiently without heavy configuration. One key feature of NetCluster is its PIN-based cluster connection mechanism, which allows worker nodes to join a cluster securely by entering a shared PIN generated by the master node. This eliminates the need for complex network discovery protocols and also allows multiple independent clusters to coexist on the same network without interference.

The main contributions of this work are:

- Design and development of a lightweight distributed cybersecurity toolkit for LAN environments
- Implementation of a PIN-based cluster connection system for secure and controlled node participation
- Support for running multiple independent clusters within the same LAN
- User-controlled CPU allocation to enable flexible and resource-aware computation

- Efficient workload distribution using the Ray framework with evaluation on a real-world password dataset

II. LITERATURE SURVEY

Distributed computing has a long history of being applied to large-scale computational problems. The MapReduce paradigm introduced by Dean and Ghemawat [3] was a major step forward, enabling parallel processing of massive datasets across commodity hardware. This model was later extended by Apache Hadoop, which became the go-to framework for batch processing in enterprise environments. Apache Spark [4] improved upon Hadoop by keeping data in memory rather than writing intermediate results to disk, making it significantly faster for iterative workloads.

However, both Hadoop and Spark are heavy frameworks. They require dedicated cluster management systems, specific configurations, and considerable engineering effort to set up and maintain. For a small academic lab with a few machines, this overhead is not justified.

The Ray framework [1][2] represents a newer approach to distributed computing, designed with Python developers in mind. Ray provides a task-based execution model where Python functions can be turned into remote tasks with minimal code changes. It handles scheduling, resource management, and fault tolerance internally, making it much easier to deploy distributed applications quickly.

On the cybersecurity side, tools like Hashcat and John the Ripper are the industry standard for password cracking. Hashcat in particular is extremely optimized and can leverage GPUs for massive parallelism, supporting hundreds of hash types. However, these tools are typically designed for single-machine use and do not offer a straightforward mechanism for distributing work across multiple regular machines on a LAN. NetCluster addresses this gap by offering a simple, Python-based distributed platform accessible to anyone with basic programming knowledge.

III. PROPOSED SYSTEM

A. System Architecture

NetCluster follows a Master-Worker architecture. One machine act as the master and coordinates the entire operation, while the remaining machines act as workers and carry out the actual computation.

The master node is responsible for generating the connection PIN, managing connected worker nodes, splitting the wordlist into chunks, distributing work to each worker, and aggregating results. Worker nodes enter the PIN to connect, receive their assigned chunk of the wordlist, process it, and return the result—either the cracked password or a signal that the chunk was exhausted without a match.

B. PIN-Based Cluster Connection Mechanism

Traditional service discovery in distributed systems often relies on network broadcasting, where nodes announce themselves and others discover them automatically. This becomes problematic when multiple clusters need to run independently on the same network. NetCluster solves this with a PIN-based connection system:

- The master node generates a unique 6-digit PIN at startup
- Worker nodes receive this PIN and enter it manually to connect
- Each worker also provides a custom identifier for the node
- The connection is validated and the worker is registered with the master

This approach reduces unnecessary network traffic, gives the operator explicit control over which machines join the cluster, and naturally supports multiple independent clusters on the same network since each cluster uses its own PIN.

C. CPU Resource Allocation

NetCluster allows the operator to specify how many CPU cores should be allocated to Ray on both the master and each worker node. This is passed as a configuration parameter at startup and enforced by the Ray runtime. A node with 8 cores might contribute only 4 to the cluster, leaving the rest available for other tasks. This feature allows fine-grained performance tuning and prevents any node from being completely overloaded during execution.

D. Distributed Execution Mechanism

The workload distribution logic divides the wordlist into equal segments based on the number of available workers:

Chunk Size = Total Passwords / Number of Workers

Each worker receives a start index and an end index and processes only that range. Workers hash each candidate password using SHA-256 and compare it against the target hash. If a match is found, the worker reports back to the master and execution stops. The system also includes a fallback to local execution if Ray is unavailable or the cluster has only one node, making it usable in single-machine mode without any code changes.

IV. METHODOLOGY

A. Task Distribution

The master node reads the wordlist file and computes the start and end index for each worker based on the total number of passwords and connected workers. These index pairs are then passed to each worker as remote Ray tasks. Workers access the wordlist file directly from the shared network path, or the master can transmit the relevant slice to each worker.

B. Execution Flow

The overall execution follows a clear sequence. First, the master initializes the Ray runtime and generates the connection PIN. Workers connect using this PIN. Once all expected workers have joined, the master divides the wordlist and dispatches tasks. Each worker processes its chunk and returns either a found password or a null result. The master collects all results and reports the outcome. Execution stops as soon as any worker finds the correct password, implemented using Ray's future-based result retrieval.

C. Performance Considerations

NetCluster is built entirely in Python and uses the Ray framework, which introduces some overhead compared to tools like Hashcat written in optimized C/C++ with GPU support. The main sources of overhead include Python interpreter speed, file I/O latency, Ray task scheduling delay, and network communication cost. All of these factors are accounted for in the experimental evaluation, and the results reflect realistic performance under these real-world conditions.

V. EXPERIMENTAL SETUP

The system was evaluated on a local area network using the rockyou.txt dataset, which contains approximately 14 million real-world passwords and is a widely used benchmark in password security research. The target hash was pre-computed using SHA-256. The experimental parameters are summarized in Table I.

TABLE I. EXPERIMENTAL SETUP PARAMETERS

Parameter	Value
Dataset	rockyou.txt (~14M passwords)
Hash Type	SHA-256
Nodes Used	1, 2, 3, 5
Processor	Intel i5 / AMD Ryzen 5
RAM	8 GB per node
Network	LAN (Wi-Fi)
Framework	Ray (Python)

Experiments were conducted with varying numbers of worker nodes (1, 2, 3, and 5) and different CPU allocations per node, allowing evaluation of both the effect of adding more machines and increasing resources on each machine.

VI. RESULTS AND ANALYSIS

A. Execution Time Analysis

Table II shows the execution time for different worker and CPU core configurations. The results clearly demonstrate that adding more workers and more CPU cores significantly reduces the time needed to process the full wordlist.

TABLE II. EXECUTION TIME VS. WORKER CONFIGURATION

Workers	CPU/ Node	Tot. Cores	Speed	Time(s)
1	1 core	1	~200K/sec	70
1	4 cores	4	~600K/sec	25
2	2 cores	4	~800K/sec	20
3	2 cores	6	~1.2M/sec	12
5	2 cores	10	~2.0M/sec	7

The jump from a single worker with 1 core (70 seconds) to 5 workers with 2 cores each (7 seconds) represents a 10x speedup for a 10x increase in total CPU cores. This near-linear scaling suggests that the workload is well-parallelized with minimal contention or synchronization overhead.

B. Workload Distribution Analysis

Table III shows how the wordlist is divided across workers and the resulting execution times.

TABLE III. WORKLOAD DISTRIBUTION ANALYSIS

Workers	Passwords/Worker	Time/Worker	Total Time
1	14,000,000	70 sec	70 sec
2	7,000,000	18 sec	18-20 sec
3	4,666,666	12 sec	12-14 sec
5	2,800,000	7 sec	7-9 sec

Each worker's share of the workload decreases proportionally with the number of workers. The total execution time is slightly higher than the per-worker time due to scheduling and aggregation overhead, but this overhead is small (typically 1-2 seconds) and does not significantly affect the overall speedup.

C. CPU Allocation Impact

Table IV shows how different CPU allocation strategies affect execution time while holding the dataset constant.

TABLE IV. CPU ALLOCATION VS. EXECUTION TIME

Configuration	Total Cores	Time (sec)
1 worker, 1 core	1	70
1 worker, 4 cores	4	25
3 workers, 1 core each	3	28
3 workers, 2 cores each	6	12
5 workers, 2 cores each	10	7

An interesting observation is that 3 workers with 1 core each (28 seconds) is slightly slower than 1 worker with 4 cores (25 seconds), despite similar total core counts. This is because the distributed case incurs additional overhead from network communication and Ray scheduling, while the single-machine case benefits from shared memory and lower coordination cost.

D. Resource Utilization

Table V summarizes CPU usage, memory consumption, and network traffic across configurations.

TABLE V. RESOURCE UTILIZATION SUMMARY

Workers	CPU Usage	RAM Usage	Network
1	90-100%	200-300 MB	Negligible
3	75-90%	350-500 MB	Low
5	70-85%	400-600 MB	Moderate

CPU utilization decreases slightly as more workers are added, likely because each worker finishes its smaller chunk faster, leaving some idle time while waiting for aggregation. RAM usage stays well within the 8 GB available per node. Network usage remains low even with 5 workers, confirming the system is not bottlenecked by network bandwidth.

E. Graph Analysis

From the Workers vs. Execution Time analysis, execution time drops sharply as workers increase from 1 to 3, then levels off between 3 and 5. This is consistent with Amdahl's Law—as parallelism increases, the non-parallelizable portions such as file reading and result aggregation become the bottleneck. From the CPU Allocation vs. Execution Time analysis, increasing CPU cores per node reduces execution time with diminishing returns beyond 4 cores, as the SHA-256 hashing operation becomes memory-bound at higher core counts.

VII. DISCUSSION

The experimental results validate the core hypothesis of this work: distributing a password cracking workload across multiple LAN-connected machines using Ray can significantly reduce execution time while keeping the system simple and easy to deploy. The PIN-based connection mechanism proved

practical and effective during testing—connecting workers to the master took only a few seconds per node, and there were no conflicts between separate clusters on the same network.

The CPU allocation feature also worked as intended. Running the system on lab machines during working hours with only 2 cores allocated per node did not disrupt other users, while full-core experiments could be run at off-peak times for maximum performance.

NetCluster is considerably slower than Hashcat for the same hardware, primarily because Python is not as efficient as C for CPU-intensive work and because GPU acceleration is not yet supported. For professional security contexts, Hashcat remains the better choice. However, NetCluster fills a different niche—it is designed as a learning tool and a flexible research platform where simplicity, programmability, and ease of deployment matter more than raw throughput.

VIII. CONCLUSION

This paper presented NetCluster, a lightweight distributed cybersecurity toolkit designed for local network environments. By combining the Ray framework's efficient task-based execution model with a PIN-based cluster management system and user-controlled CPU allocation, the system achieves meaningful parallelism while remaining accessible to students and researchers without specialized infrastructure.

The experimental evaluation on the rockyou.txt dataset demonstrated near-linear speedup for small clusters, reducing execution time from 70 seconds on a single core to just 7 seconds with 5 workers and 10 total cores. NetCluster successfully bridges the gap between heavyweight enterprise frameworks and single-machine tools, offering a practical middle ground for educational and research cybersecurity applications.

IX. FUTURE SCOPE

Several directions exist for extending this work. On the performance side, integrating GPU acceleration through Ray's GPU support, or wrapping Hashcat as a backend, could dramatically increase throughput. Supporting additional hash types such as MD5, bcrypt, and Argon2 would broaden applicability. On the usability side, replacing manual PIN distribution with an automatic discovery mechanism—such as a web-based dashboard or QR code—would lower the barrier to entry. Cloud deployment support would allow the system to scale beyond local networks. Finally, adding work-stealing algorithms for load balancing and encrypted communication between nodes would improve robustness and security.

ACKNOWLEDGMENT

The authors would like to thank the Department of Computer Science and Engineering, Sandip University, Nashik, for providing the laboratory infrastructure and resources required to conduct this research. We also acknowledge the open-source contributions of the Ray project and the Python community.

REFERENCES

[1] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, “Ray: A distributed framework for emerging AI applications,” *arXiv preprint* arXiv:1712.05889, 2017.

[2] P. Moritz *et al.*, “Ray: A distributed execution framework for emerging AI applications,” in *Proc. 13th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, 2018, pp. 561–577.

[3] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in *Proc. 6th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, 2004, pp. 137–150.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in *Proc. 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, Boston, MA, 2010.

[5] Python Software Foundation, *Python Language Reference*, Version 3.11. [Online]. Available: https://docs.python.org

[6] Anyscale, Inc., *Ray Documentation*, Version 2.x. [Online]. Available: https://docs.ray.io