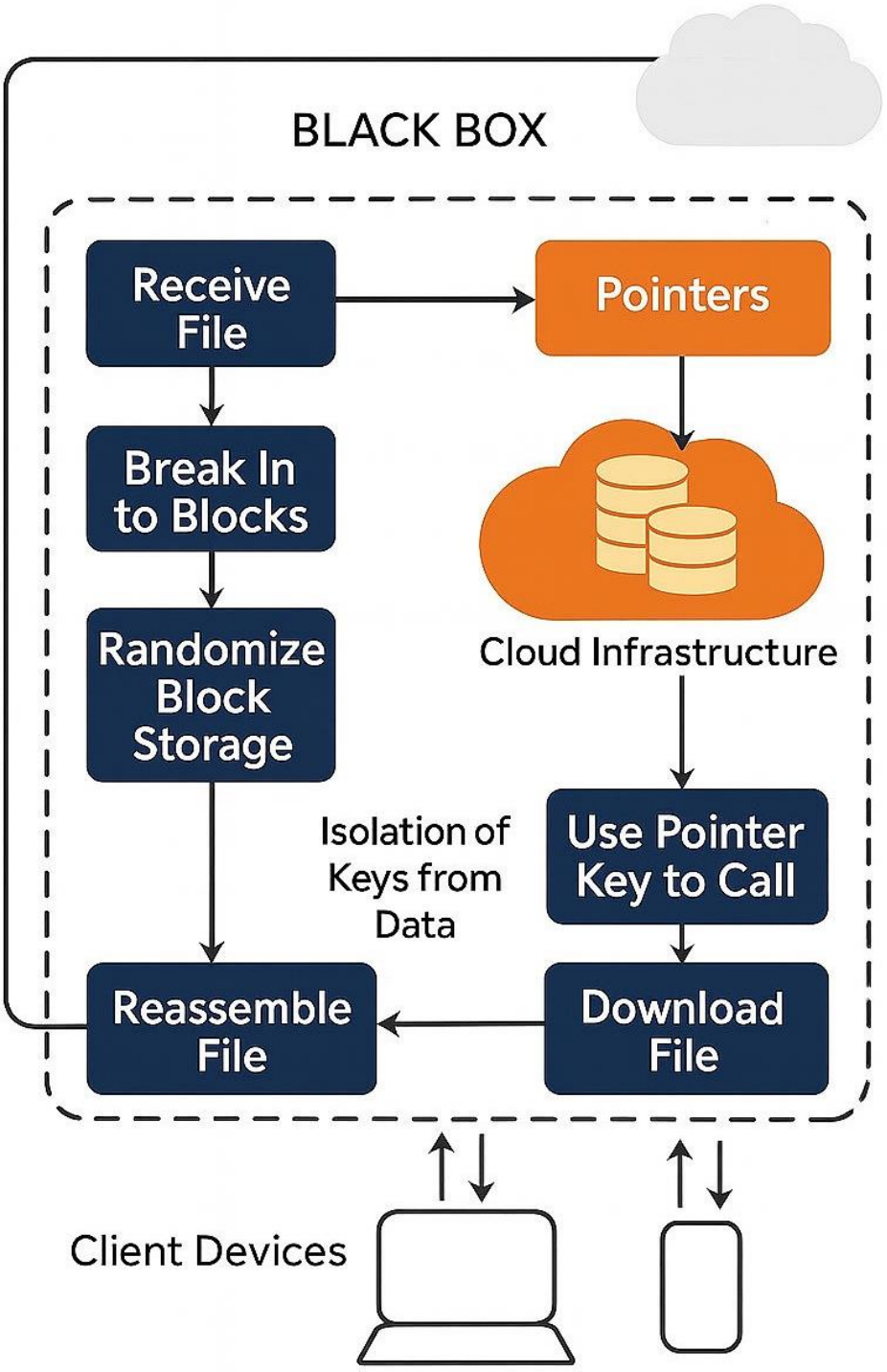
Data Security Using Randomized Features DSURF



Data Security Using Randomized Features DSURF. The best solution to stop large-scale data theft. All Data Center operators need to know this... Reducing the scale of attack surfaces via randomization.

DSURF provides a cloud-native storage security architecture designed for resilience and zero-trust environments.

Prepared by: DSURF Software Development Team July 2025

Contents

---

1. Introduction

DSURF (Data Security Using Randomized Features) is a revolutionary software solution designed to prevent large-scale data theft in cloud-based environments by fragmenting, randomizing, and isolating data storage from pointer-based reassembly keys.

DSURF disrupts traditional storage methods by:

- Randomizing data block sizes
- Distributing data blocks across multiple cloud locations
- Separating reconstruction keys from data
- Preventing client-side access to storage logic (black box design)
- Running on AWS, Azure, Google Cloud, or hybrid infrastructures

The system is seamless from the user's perspective, yet radically more secure in its internal logic.

---

2. DSURF System Flow Diagram

DSURF System Flow Diagram:

The flow diagram (above) summarizes the main aspects of DSURF file handling cycle:

- File reception
- Breaking files apart into data blocks
- Randomized storage of blocks and overflow handling
- Pointer key creation and isolation of keys from data
- File reassembly and return

---

3. Technology Overview and Benefits

DSURF introduces storage unpredictability and data separation and intermixing of blocks from many different files randomized into storage locations to defeat systemic file compromise.

Key steps:

- Files broken into blocks that may be randomly sized
- Random location generation and validation of storage locations
- Collision avoidance and / or overflow-enabled writing
- Pointer array creation and secure isolation
- Parallel processing potential for retrieval of blocks from storage
- Accelerable reassembly of files

Benefits:

Feature	Benefit
Randomized block sizes	Breaks pattern matching and fixed-exploit attacks
Randomized storage paths	Obscures continuity of file structure
Pointer-key isolation	Prevents theft of usable data without full access
Platform agnostic	Compatible with AWS, Azure, GCP
Black box logic	Clients can’t inspect or replicate DSURF internals
Parallel processing	Provides acceleration potential for highspeed performance

---

4. Technical Challenges and Implementation Notes

Challenge	Solution
Ensuring uniqueness of storage locations	Use cryptographic random generation + collision check
Cloud API differences	Abstract through drivers per platform (S3, Blob, GCS)
Handling overflow blocks	Modular logic to track multi-segment blocks
Preventing pointer leakage	Store separately, encrypted, in different network

Challenge	Solution
	paths
Performance of reassembly	Parallel block fetch and memory buffering

5. Pseudocode Portable Language Agnostic Prototype

```
MODULE DSURF_Main

  PROCEDURE StoreFile(filePath, keyStorageLocation)
    DECLARE fileBlocks, pointerArray
    fileBlocks ← SplitFileIntoBlocks(filePath)

    FOR EACH block IN fileBlocks DO
      DECLARE location
      REPEAT
        location ← GenerateRandomStorageLocation(block)
      UNTIL CheckStorageCollision(location, block.SIZE) = FALSE

      IF IsLocationSufficient(location, block.SIZE) THEN
        WriteBlock(location, block)
        pointerArray.ADD(location)
      ELSE
        DECLARE part1, part2
        (part1, part2) ← SplitBlockForOverflow(block, location.SIZE)
        WriteBlock(location, part1)
        overflowLocation ← GenerateOverflowLocation(part2)
        WriteBlock(overflowLocation, part2)
        pointerArray.ADD((location, overflowLocation))
      END IF
    END FOR
    StorePointerKey(pointerArray, keyStorageLocation)
  END PROCEDURE

  PROCEDURE RetrieveFile(pointerKey, destinationPath)
    DECLARE reassembledBlocks
    FOR EACH pointer IN pointerKey DO
      IF pointer IS TUPLE THEN
        part1 ← ReadBlock(pointer[0])
        part2 ← ReadBlock(pointer[1])
        block ← MergeBlockParts(part1, part2)
      ELSE
        block ← ReadBlock(pointer)
      END IF
      reassembledBlocks.ADD(block)
    END FOR
    ReassembleFile(reassembledBlocks, destinationPath)
  END PROCEDURE
```

```
END MODULE

FUNCTION SplitFileIntoBlocks(filePath)
  [Reads file and splits into randomly sized blocks]
  RETURN listOfBlocks
END FUNCTION

FUNCTION GenerateRandomStorageLocation(block)
  [Returns random path or cloud object location based on size]
  RETURN location
END FUNCTION

FUNCTION CheckStorageCollision(location, size)
  [Checks if location is already used or occupied]
  RETURN BOOLEAN
END FUNCTION

FUNCTION IsLocationSufficient(location, size)
  [Checks if block fits fully in location]
  RETURN BOOLEAN
END FUNCTION

PROCEDURE WriteBlock(location, block)
  [Writes the block to cloud storage]
END PROCEDURE

FUNCTION SplitBlockForOverflow(block, maxSize)
  [Splits block into part1 and overflow part2]
  RETURN (part1, part2)
END FUNCTION

FUNCTION GenerateOverflowLocation(block)
  [Generates a new location for overflow part]
  RETURN location
END FUNCTION

FUNCTION ReadBlock(location)
  [Reads block from storage]
  RETURN block
END FUNCTION

FUNCTION MergeBlockParts(part1, part2)
  [Merges partial block and overflow]
  RETURN block
END FUNCTION

PROCEDURE ReassembleFile(blocks, destinationPath)
```

```
  [Writes blocks into a single output file]
END PROCEDURE

PROCEDURE StorePointerKey(pointerArray, keyStorageLocation)
  [Stores pointer list to separate key infrastructure]
END PROCEDURE
```

---

6. C Language and Compiler Source Code Prototype

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_BLOCK_SIZE 4096
#define MIN_BLOCK_SIZE 512
#define MAX_POINTERS   10000

typedef struct {
    char *data;
    size_t size;
} Block;

typedef struct {
    char *location;
    char *overflow_location; // NULL if unused
} Pointer;

typedef struct {
    Pointer pointers[MAX_POINTERS];
    int count;
} PointerArray;

// Function prototypes
Block *split_file_into_blocks(const char *filepath, int *num_blocks);
char *generate_random_location(size_t size);
int check_storage_collision(const char *location, size_t size);
int is_location_sufficient(const char *location, size_t size);
void write_block(const char *location, Block block);
void write_overflow(const char *location, Block block);
void store_pointer_key(PointerArray *key, const char *key_storage);
Block read_block(const char *location);
Block merge_blocks(Block part1, Block part2);
void reassemble_file(Block *blocks, int count, const char
*output_path);
```

```
void store_file(const char *filepath, const char *key_storage) {
    int i, num_blocks;
    PointerArray pointer_array = { .count = 0 };
    Block *blocks = split_file_into_blocks(filepath, &num_blocks);

    for (i = 0; i < num_blocks; i++) {
        char *loc = NULL;
        do {
            loc = generate_random_location(blocks[i].size);
        } while (check_storage_collision(loc, blocks[i].size));

        if (is_location_sufficient(loc, blocks[i].size)) {
            write_block(loc, blocks[i]);
            pointer_array.pointers[pointer_array.count++] = (Pointer){
loc, NULL };
        } else {
            size_t half = blocks[i].size / 2;
            Block part1 = { blocks[i].data, half };
            Block part2 = { blocks[i].data + half, blocks[i].size -
half };

            write_block(loc, part1);
            char *overflow_loc = generate_random_location(part2.size);
            write_block(overflow_loc, part2);
            pointer_array.pointers[pointer_array.count++] = (Pointer){
loc, overflow_loc };
        }
    }

    store_pointer_key(&pointer_array, key_storage);
    for (int i = 0; i < num_blocks; i++) {
        free(blocks[i].data);
    }
    free(blocks);
}

void retrieve_file(PointerArray *key, const char *output_path) {
    Block *blocks = malloc(sizeof(Block) * key->count);

    for (int i = 0; i < key->count; i++) {
        Block part1 = read_block(key->pointers[i].location);

        if (key->pointers[i].overflow_location != NULL) {
            Block part2 = read_block(key-
>pointers[i].overflow_location);
            blocks[i] = merge_blocks(part1, part2);
```

```
        } else {
            blocks[i] = part1;
        }
    }

    reassemble_file(blocks, key->count, output_path);
    for (int i = 0; i < key->count; i++) {
        free(blocks[i].data);
    }
    free(blocks);
}
```

---

7. Action Plan for Developers and Deployment

Includes step-by-step actions for:

- Compiling the DSURF core
- Building platform drivers
- Configuring secure storage zones
- Deploying in test infrastructure
- Benchmarking and hardening

Deployment Roadmap:

Phase 1 – Compile core DSURF modules (2 days)

Phase 2 – Build platform-specific API drivers (3–5 days)

Phase 3 – Configure secure key storage zones (3 days)

Phase 4 – Deploy in test infrastructure (1 week)

Phase 5 – Benchmarking and performance tuning (1 week)

Phase 6 – Hardened deployment in live cloud (2 weeks)

---

8. Conclusion

DSURF changes the game in cloud data protection. With strong randomization, separation of concerns, and practical performance, it meets the needs of modern data center operators.

Developer Steps

- Build, compile, and test modules
- Integrate with target cloud
- Deploy in container or region

Customer Benefits

- Blocks mass data theft
- Aligns with zero-trust strategy
- Enhances platform security

Partner Invitation

We invite:

- Cloud providers, defense agencies, and cybersecurity firms
- Data center operators and SaaS developers
- Interested partners and investors

Contact: dsurf@systemsdi.com

Web: https://SystemsDesignInnovation.com/dsurf/

All rights reserved. © 2005 Systems Design Innovation LLC. DSURF is a newly patented solution from US 17/339,935 A invented by Mark Taylor.

Systems Design Innovation LLC
301 West Broad Street
Suite 226
Falls Church
VA 22046