AI Security

Understanding Binary Patch Diffing

Understanding Binary Patch Diffing

Author: Scott Thornton, perfecXion.ai

Published: January 25, 2026

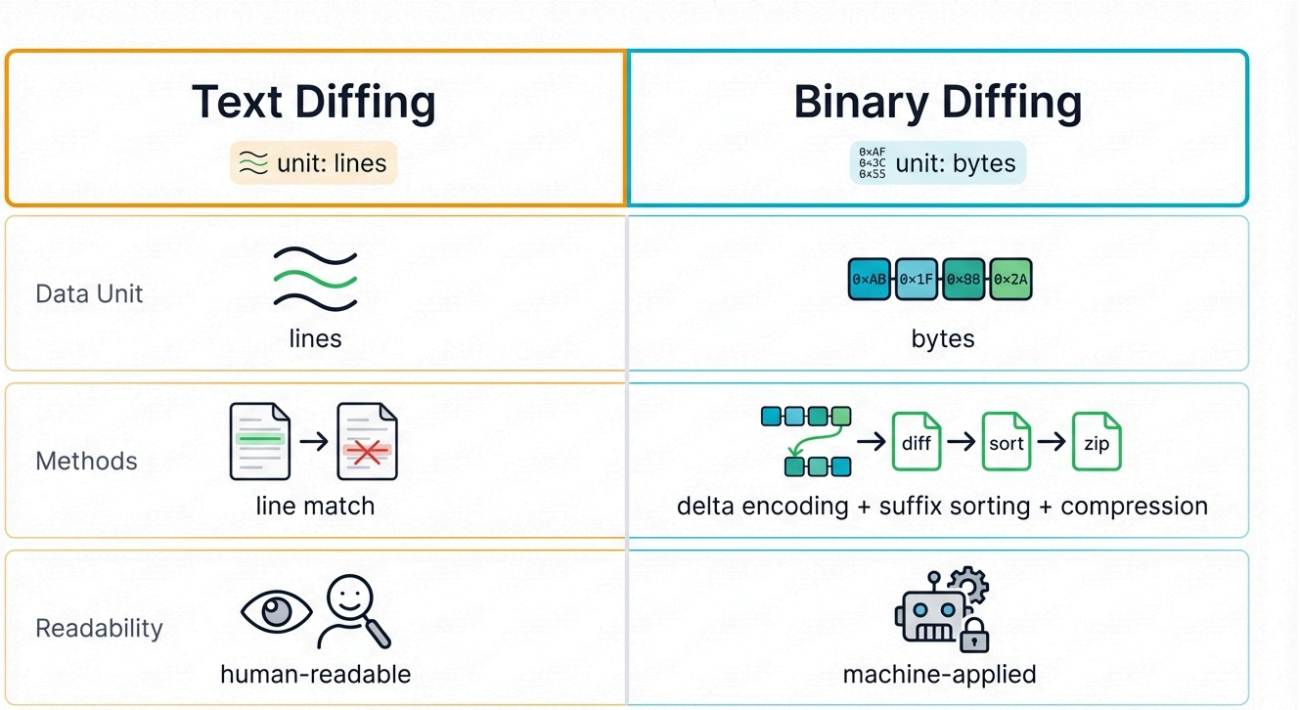
Read Time: 10 minutes

Binary patch diffing compares two binaries. You take two files—executables, libraries, firmware—and hunt for differences. The tool generates a compact patch file that transforms the original into its modified twin, capturing only what changed while leaving everything else untouched.

Text diffing won't cut it here. Source code lives in lines, but binaries speak in bytes, and those bytes demand specialized algorithms that focus on delta compression to shrink patch sizes down to the absolute minimum. This isn't about human readability—it's about precision, efficiency, and ruthless optimization at the byte level.

Software updates rely on this technique to ship small patches instead of massive full binaries, saving bandwidth and deployment time. Reverse engineers use it to dissect security patches, tracking exactly what changed and why. Version control systems lean on it for binary assets, keeping repositories lean while maintaining perfect history. The applications span from routine updates to high-stakes vulnerability analysis where every byte matters.

Key Differences from Text Diffing

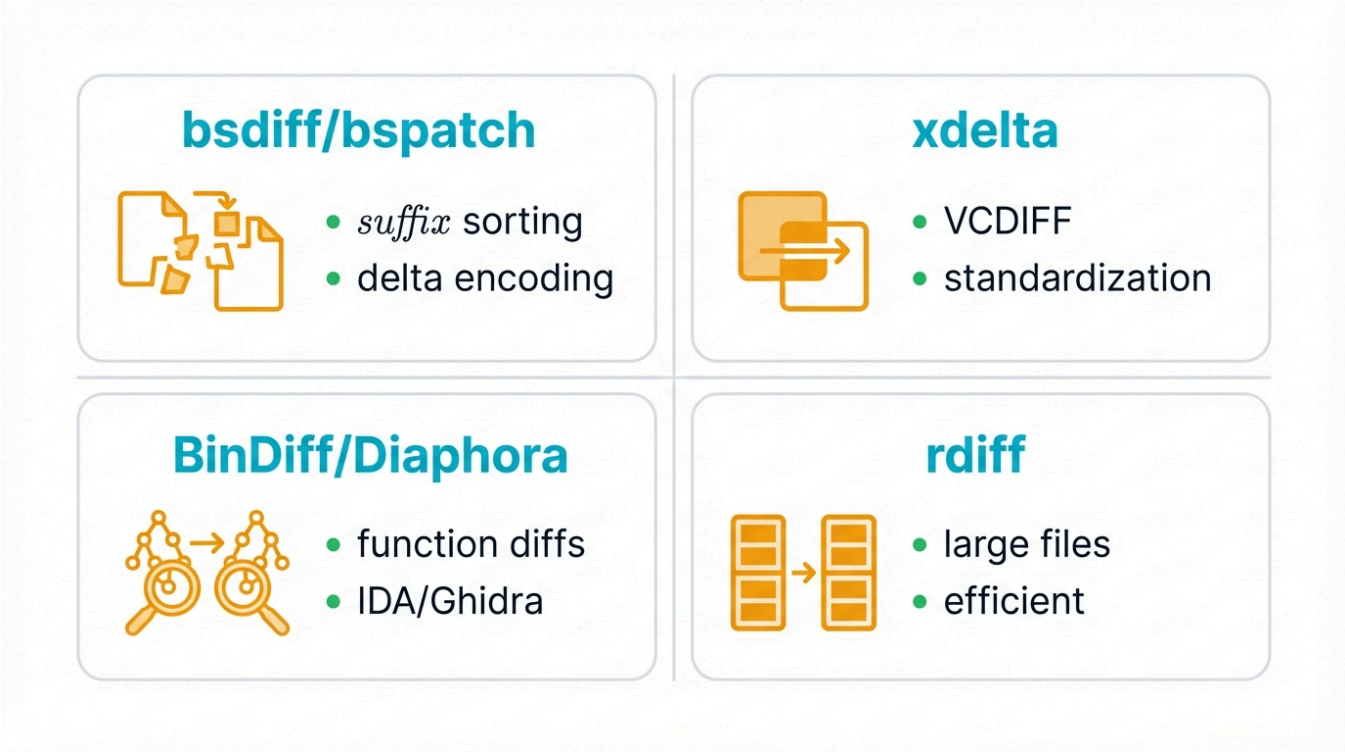


Text vs Binary Diffing

- **Text diffs** rely on line-by-line comparisons. Tools like `diff -u` match lines, track context, and produce human-readable output that developers scan with their eyes.

- **Binary diffs** operate at the byte level. They use delta encoding, suffix sorting, and compression optimizations because binaries don't have natural line breaks—just raw streams of data that demand different strategies entirely.
- Forget human readability. Patches exist for machines to apply with perfect precision, not for humans to review casually, and that fundamental difference shapes everything about how we generate and handle them.

Popular Binary Diffing Tools



Binary Diffing Tools Overview

bsdiff/bspatch

Colin Percival built this tool for efficient general binary patches. It combines suffix sorting with delta encoding to produce remarkably small patches, and it's become the gold standard for many update systems because it just works.

xdelta

Similar functionality, different flavor. This tool supports the VCDIFF format defined in RFC 3284, making it popular in backup systems and version control where standardization matters as much as performance.

BinDiff/Diaphora

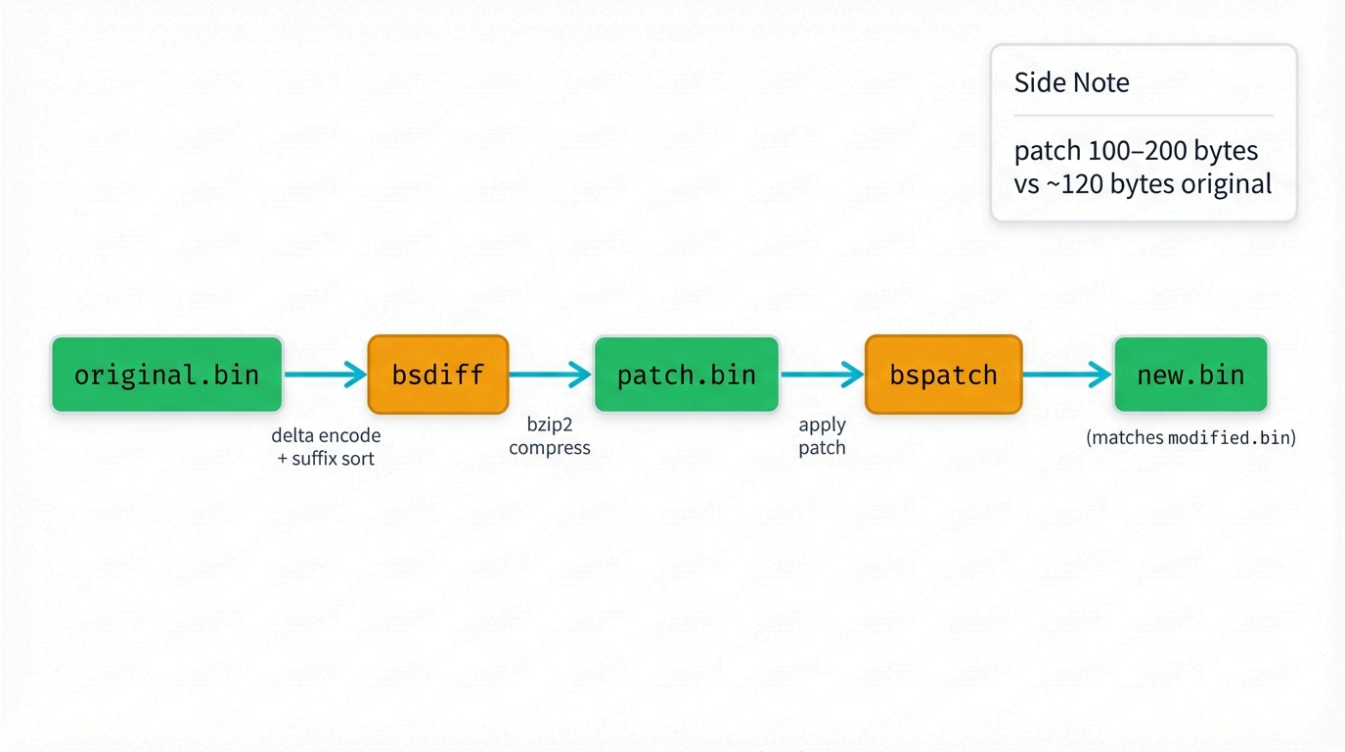
Reverse engineers reach for these when analyzing structural changes in code, integrating with powerhouses like IDA Pro or Ghidra to highlight function differences, code flow modifications, and architectural shifts that matter in security research.

rdiff

Part of librsync, this tool tackles large files like disk images with specialized optimizations. When you're diffing gigabytes of data, rdiff handles the substantial binary workload efficiently without choking on memory constraints.

Step-by-Step Example with bsdiff

Let's walk through a real example. We'll use **bsdiff** and **bspatch** to create and apply binary patches on a Unix-like system—Linux or macOS—assuming you've got bsdiff installed and ready to roll.



bsdiff Patch Workflow

Installation: Ubuntu users run `sudo apt install bsdiff`. macOS folks use `brew install bsdiff`. Or compile from source if you're feeling adventurous.

1 Prepare the Original Binary File

Start with a simple binary. We'll create a file representing a small program or data blob using hexadecimal data to simulate two versions of the same binary.

```
00000000: 4d5a 9000 0300 0000 0400 0000 ffff 0000  MZ..............
00000010: b800 0000 0000 0000 4000 0000 0000 0000  ........@.......
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 8000 0000 0e1f ba0e  ................
00000040: 00b4 09cd 21b8 014c cd21 5468 6973 2070  ....!..L.!This p
00000050: 726f 6772 616d 2063 616e 6e6f 7420 6265  rogram cannot be
00000060: 2072 756e 2069 6e20 444f 5320 6d6f 6465  run in DOS mode
00000070: 2e0d 0d0a 2400 0000 0000 0000            ....$.......
```

This stub comes from a PE executable header—the kind Windows EXE files use. Save it as `original.bin` with Python.

```
with open('original.bin', 'wb') as f:
    f.write(b'\x4d\x5a\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00\xff\xff\x00\x00\xb8\x00\x00\
```

2 Create the Modified Binary File

Now build `modified.bin` with deliberate changes. We'll transform the message from "cannot be run" to "must not run" and toss in some padding for good measure.

Changes: Bytes 0x55-0x60 carry the modified text. Null bytes pad the end, expanding the file slightly.

3 Generate the Binary Patch

Fire up bsdiff to create the patch:

```
bsdiff original.bin modified.bin patch.bsdiff
```

Watch bsdiff work its magic, computing deltas using a suffix array that matches common subsequences with ruthless efficiency, then encoding the differences with bzip2 compression that squeezes every unnecessary byte out of the result. Your patch file might clock in around 100-200 bytes for this small change, dwarfed by the ~120 bytes of the original files—that's the power of delta encoding in action.

4 Apply the Binary Patch

Transform `original.bin` back into its modified version. Use bspatch.

```
bspatch original.bin new.bin patch.bsdiff
```

This creates `new.bin`. It should match `modified.bin` byte for byte.

```
# Verification
cmp modified.bin new.bin  # Should show no differences
sha256sum modified.bin new.bin  # Compare checksums
```

Python Script for Binary Patch Diffing

Here's a Python script that shows you a simple binary diffing and patching process from scratch. This isn't as efficient as specialized tools like bsdiff—those are battle-tested and optimized—but it's pure Python, and it demonstrates the core concepts in code you can actually read and modify.

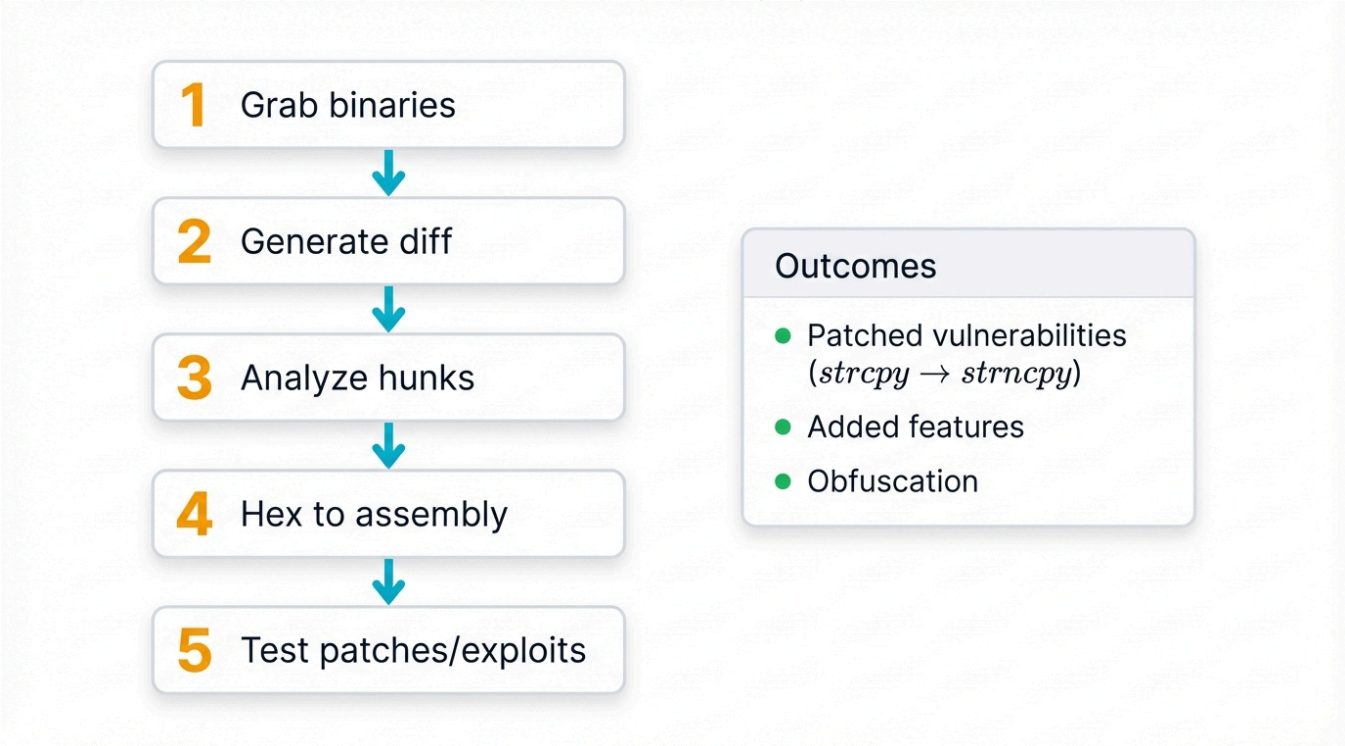
```
import difflib
import binascii

# Function to split hex string into lines for diff readability
def split_into_lines(data, chars_per_line=32):
    return [data[i:i+chars_per_line] for i in range(0, len(data), chars_per_line)]

# Function to generate a unified diff of two binaries (in hex for readability)
def binary_diff(original_bytes, modified_bytes):
    orig_hex = binascii.hexlify(original_bytes).decode('ascii').upper()
    mod_hex = binascii.hexlify(modified_bytes).decode('ascii').upper()

    orig_lines = split_into_lines(orig_hex)
    mod_lines = split_into_lines(mod_hex)

    diff = difflib.unified_diff(
        orig_lines, mod_lines,
        fromfile='original.bin (hex)',
        tofile='modified.bin (hex)',
        lineterm=''
    )
    return '\n'.join(diff)

# Simple patching function: Applies a list of (offset, remove_bytes, add_bytes) deltas
def apply_patch(original_bytes, deltas):
    result = bytearray(original_bytes)
    offset_adjust = 0
    for offset, remove_len, add_bytes in deltas:
        adj_offset = offset + offset_adjust
        result = result[:adj_offset] + add_bytes + result[adj_offset + remove_len:]
        offset_adjust += len(add_bytes) - remove_len
    return bytes(result)

# Example usage
if __name__ == "__main__":
    # Sample original and modified binaries
    original = b'\x4d\x5a\x90\x00...'  # (truncated for brevity)
    modified = b'\x4d\x5a\x90\x00...'  # (truncated for brevity)

    # Generate and print diff
    diff_output = binary_diff(original, modified)
    print("Unified Diff (Hex Representation):\n")
    print(diff_output)

    # Example deltas for patching
    deltas = [
        (85, 18, b'\x6d\x75\x73\x74\x20\x6e\x6f\x74\x20\x72\x75\x6e...'),
        (112, 0, b'\x00\x00\x00\x00')  # Added padding
    ]
```

```
    # Apply patch and verify
    patched = apply_patch(original, deltas)
    assert patched == modified, "Patch failed!"
    print("\nPatch applied successfully!")
```

Note: This script teaches concepts. difflib excels at visualization but stumbles on large binaries and compact patches. For production work, reach for specialized libraries like xdelta or implement algorithms like those powering bsdiff.

Reverse Engineering with Binary Patch Diffing

Reverse engineering thrives on comparison. You grab two versions—vulnerable versus patched, old versus new—and analyze the differences to understand exactly what changed, why it matters, and what it reveals about the underlying system. Binary patch diffing spotlights modified code sections without forcing you to fully disassemble both files, saving time while revealing critical insights.



Reverse Engineering Diff Analysis Flow

Analysis Steps

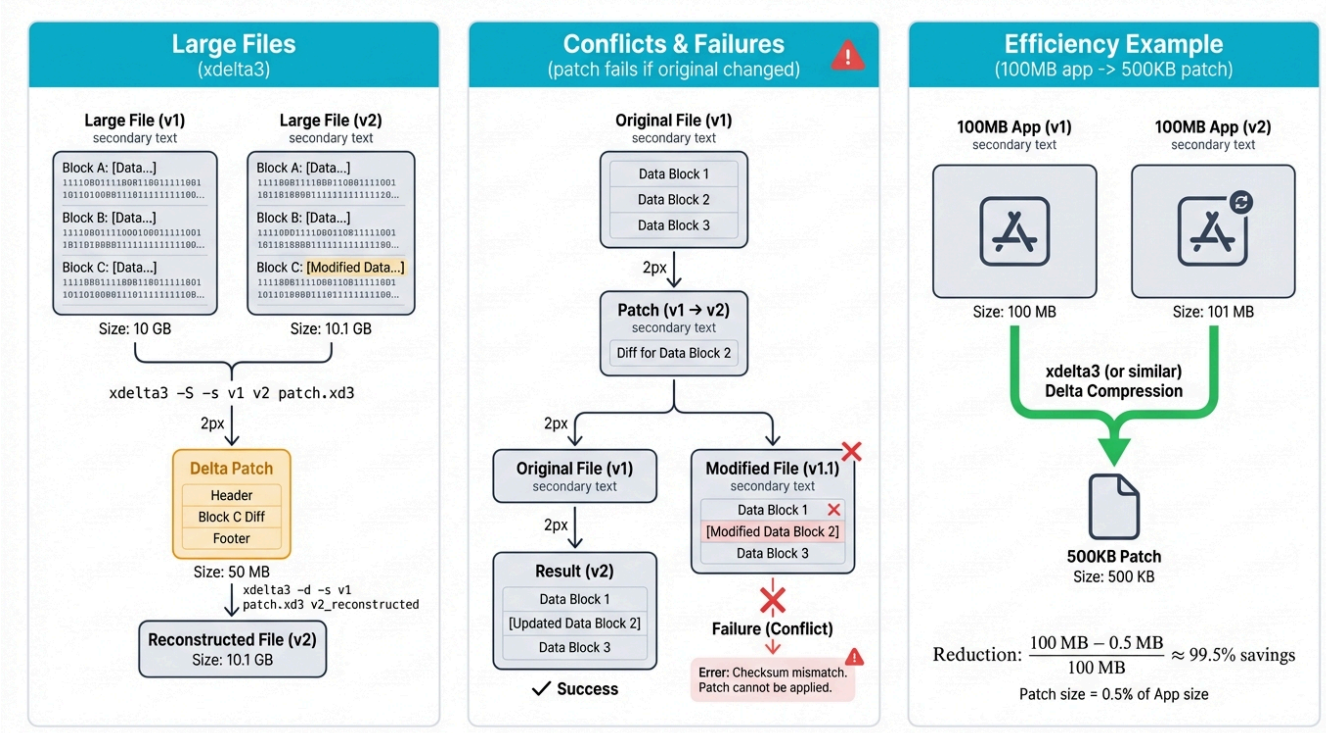
- 1 Grab both binary versions. Vulnerable versus patched.

- 2 Generate the diff using your chosen tools or scripts.
- 3 Analyze hunks for changes that matter, filtering noise from signal.
- 4 Convert hex to assembly, translating bytes into instructions you can understand.
- 5 Test patches and derive exploits, turning analysis into actionable intelligence.

What Diffs Reveal

- **Patched Vulnerabilities:** Watch strcpy transform into strncpy, exposing exactly where buffer overflows lurked and how developers plugged the hole.
- **Added Features:** New code blocks appear, fresh library calls emerge, revealing functionality that didn't exist before.
- **Obfuscation:** Randomized changes scatter across the binary to evade detection, signature analysis, and pattern matching.

Handling Advanced Scenarios



Advanced Scenarios Snapshot

Large Files

Gigabyte-sized files demand specialized tools. Consider xdelta3.

```
xdelta3 -e -s original.bin modified.bin patch.delta
xdelta3 -d -s original.bin patch.delta new.bin
```

Reverse Engineering Analysis

Ghidra's "Version Tracking" tool handles structural analysis beautifully. Integrate BinDiff for function matching that reveals architectural changes across versions.

Conflicts & Failures

Original binary changed? Your patch fails. Tools like bspatch don't resolve conflicts automatically—they just bail out, leaving you to figure out what went wrong and how to fix it.

Efficiency Example

Picture a 100MB app with 1MB of actual changes. bsdiff can produce a 500KB patch, slashing bandwidth costs and deployment time while maintaining perfect accuracy.

Ready to Dive Deeper?

Explore advanced patch diffing techniques, reverse engineering methodologies, and comprehensive analysis frameworks that take your skills to the next level.

Explore More Guides (/pages/knowledge-hub.html) Python Security Tools (/articles/python-security-arsenal-tools-automation.html)

Thank You for Reading

Explore more AI security research at **perfecxion.ai**

This document was generated from perfecXion.ai
For the latest updates, visit the online version