PIPpro Technical Reference

Obfuscation Engine Internals & Security Analysis

Version	2.0.0
Date	April 2026
Product	PIPpro by DenseDefense
Engine	AST-based, 7-phase pipeline, stdlib-only core
Publisher	DenseDefense -- DFW, Texas
Classification	Internal / Partner Technical Reference

Table of Contents

1. Architecture Overview

PIPpro v2.0.0 is implemented as a single Python module (obfuscate_source.py) with zero external dependencies for the core obfuscation engine. The architecture uses Python's ast module to parse source code into an Abstract Syntax Tree, apply transformations via NodeTransformer subclasses, and unparse the modified tree back to valid Python source. v2.0.0 adds two new AST transformer classes and hardens the encryption engine with SHA-256 stream cipher and HMAC-SHA256 integrity checking.

Core Components

Component	Class/Function	Purpose
Safe List	_build_safe_names()	Compute names that must never be renamed
PIPpro Signature	_PIPPRO_SIGNATURE	Marker for already-obfuscated detection
Stripper	_Stripper(NodeTransformer)	Remove docstrings, type annotations
Encryptor	_StringEncryptor(NodeTransformer)	SHA-256 stream cipher + HMAC string/bytes encryption
Control Flow	_ControlFlowObfuscator(NodeTransformer)	Opaque predicates, dead code injection (NEW)
Collector	_NameCollector(NodeVisitor)	Collect all defined names across files
Renamer	_Renamer(NodeTransformer)	Apply rename map to AST nodes
Import Obfusc.	_ImportObfuscator(NodeTransformer)	Encrypt non-stdlib import statements (NEW)
Pipeline	obfuscate_files()	Orchestrate phases 1-3.5 for batch processing
Compiler	compile_binary()	Invoke Nuitka with proper flags
Verifier	verify_binary()	Scan binary for plaintext leaks
CLI	main()	Argument parsing and pipeline invocation

Data Flow

```
Source .py files
  |-- ast.parse() --> AST
  |-- _Stripper.visit() --> stripped AST
  |-- _StringEncryptor.visit() --> encrypted AST (SHA-256 stream cipher)
  |-- [optional] _ControlFlowObfuscator.visit() --> obfuscated control flow
  |-- [optional] _ImportObfuscator.visit() --> encrypted imports
```

```
  |-- _StringEncryptor.inject_preamble() --> AST with decryptor + HMAC
  |-- _Renamer.visit() --> mangled AST (ALL names by default)
  |-- ast.fix_missing_locations() --> valid AST
  |-- ast.unparse() --> obfuscated source string
  |-- ast.parse() --> verify (round-trip parse)
  |-- write to output
  |
  [Optional] Nuitka compile --> native binary
  [Optional] verify_binary() --> leak scan
```

2. Pipeline Execution Order

The phase ordering is deliberate and non-negotiable. Changing the order breaks the security properties. v2.0.0 inserts two new phases (2.5 and 3.5) at specific points in the pipeline.

Why Strip First (Phase 1)

Docstrings and annotations are removed before encryption so they are not encrypted and stored in the table (wasting space and increasing the encrypted table size). Stripping first means the encryptor only processes strings that are actually used at runtime.

Why Encrypt Before Rename (Phase 2 Before 3)

The string encryptor injects decryptor functions named _s, _b, and variables named _K, _S, _C, _H, _V, _B, _D, _HB, _VB, _PP. Because these all start with an underscore, the renamer (Phase 3) will mangle them into random identifiers. This is a critical security property: the decryptor functions are indistinguishable from application code after renaming. An attacker cannot simply search for _s() calls to find the decryptor.

Why Control Flow After Encrypt (Phase 2.5)

Control flow obfuscation runs after string encryption so that the opaque predicates and dead code are applied to the already-encrypted code. The decryptor functions themselves get control flow obfuscation, making them even harder to identify and analyze.

Why Import Obfuscation Before Preamble (Phase 3.5)

Import obfuscation runs after the string encryption visitor but before preamble injection. This allows import module names to be added to the string table during the import obfuscation pass. The preamble is then injected with the complete string table (including module names).

Why Verify Last (Phase 5)

Verification must happen after compilation because it scans the final binary artifact. The binary may contain strings that were not in the original source (Nuitka metadata, C runtime strings, etc.). Only the final binary represents the attack surface.

3. Phase 1: AST Stripper (_Stripper)

The _Stripper class extends ast.NodeTransformer. It visits Module, ClassDef, FunctionDef, and AsyncFunctionDef nodes to remove docstrings, and visits AnnAssign nodes to remove type annotations.

Docstring Detection

A docstring is detected as the first statement in a module/class/function body that is an Expr node containing a Constant with a string value. This matches Python's own docstring convention. When removed, if the body becomes empty, a Pass node is inserted.

Annotation Stripping

For FunctionDef/AsyncFunctionDef nodes:

- node.returns is set to None (removes return type annotation)
- All args, kwonlyargs, posonlyargs have .annotation set to None
- vararg and kwarg annotations are set to None

For AnnAssign (annotated assignment) nodes:

- If the node has a value (x: int = 5), it becomes a plain Assign
- If the node has no value (x: int), the entire node is removed (returns None)

4. Phase 2: String Encryptor (_StringEncryptor) -- SHA-256 Stream Cipher

The _StringEncryptor is the most complex component. It implements SHA-256 key-derived stream cipher encryption with HMAC-SHA256 integrity verification, deduplication, caching, and special handling for f-strings.

Initialization

On construction, the encryptor generates:

- A 32-byte master key from os.urandom(32) -- cryptographically random
- Empty string table (_str_table) and dedup index (_str_index)
- Empty bytes table (_bytes_table) and dedup index (_bytes_index)
- Counters for encrypted/skipped strings

SHA-256 Stream Cipher (_derive_stream + _encrypt)

PIPpro v2.0.0 uses SHA-256 in counter mode to derive a unique keystream for each string. The derivation function concatenates the master key, the string's table index (4 bytes big-endian), and a block counter (4 bytes big-endian), then hashes with SHA-256. Each hash produces 32 bytes of keystream. Multiple blocks are concatenated for strings longer than 32 bytes.

```
def _derive_stream(self, index, length):
    stream = b''
    block = 0
    idx_bytes = index.to_bytes(4, 'big')
    while len(stream) < length:
        h = hashlib.sha256(
            self.key + idx_bytes + block.to_bytes(4, 'big')
        )
        stream += h.digest()
        block += 1
    return stream[:length]

def _encrypt(self, plaintext_bytes, index):
    stream = self._derive_stream(index, len(plaintext_bytes))
    return bytes(p ^ s for p, s in zip(plaintext_bytes, stream))
```

> **v1.x to v2.0.0 Encryption Upgrade**
> v1.x used simple XOR with key cycling: p[j] ^ K[j % 32]. This has a repeating key weakness -- identical plaintext at the same position mod 32 produces identical ciphertext. v2.0.0 derives a unique stream per string index via SHA-256, eliminating all repeating-key patterns.

HMAC-SHA256 Integrity (_compute_hmac)

After all strings are encrypted, PIPpro computes HMAC-SHA256 over the concatenation of all encrypted entries using the master key. This digest is stored as _H (string table) or _HB (bytes table) in the preamble.

```
def _compute_hmac(self, encrypted_entries):
    concat = b''
    for entry in encrypted_entries:
        concat += entry
    return hmac.new(self.key, concat, hashlib.sha256).digest()
```

Constant Visitor (visit_Constant)

For each Constant node in the AST:

- If the value is a str with length >= min_length: encrypt and replace with _s(idx)
- If the value is bytes with length >= min_length: encrypt and replace with _b(idx)
- Otherwise: skip (counted in strings_skipped)

JoinedStr Visitor (visit_JoinedStr)

f-strings are represented as JoinedStr nodes containing a mix of Constant nodes (literal text parts) and FormattedValue nodes (expression parts). The encryptor:

- Iterates over JoinedStr.values
- Replaces qualifying Constant parts with FormattedValue wrapping _s(idx)
- Recursively processes nested format specs (format specs can be JoinedStr)
- Leaves FormattedValue (expression) nodes unchanged

Preamble Injection (inject_preamble)

After the visitor has processed the entire module, inject_preamble() prepends the decryptor infrastructure to module.body. The injected code is constructed as AST nodes for the key, tables, and HMAC digests. The decryptor functions are parsed from source strings for clarity.

Injected preamble structure:

```
_PP = '# PIPpro-obfuscated'    # signature marker
import hashlib, hmac            # runtime deps
_K = b'\x9a\x3f...'            # 32-byte master key
_S = [b'\xf2\xa1...', ...]      # encrypted string table
_H = b'\x7c\xd2...'            # HMAC-SHA256 of string table
_C = dict()                      # decryption cache
_V = False                       # HMAC verified flag
def _s(_i):                      # string decryptor
    global _V
    if not _V:                   # verify HMAC on first call
        _cc = b''
        for _e in _S: _cc += _e
        if hmac.new(_K,_cc,hashlib.sha256).digest() != _H:
            raise RuntimeError('integrity')
        _V = True
    if _i in _C: return _C[_i]   # cache hit
    _c = _S[_i]                  # SHA-256 stream decrypt
    _st = b''; _ib = _i.to_bytes(4,'big'); _bk = 0
    while len(_st) < len(_c):
        _st += hashlib.sha256(_K + _ib + _bk.to_bytes(4,'big')).digest()
        _bk += 1
    _v = bytes(_c[_j] ^ _st[_j] for _j in range(len(_c))).decode('utf-8')
    _C[_i] = _v
    return _v
```

The bytes decryptor (_b) follows the same pattern with separate table (_B), HMAC (_HB), cache (_D), verification flag (_VB), and an index offset of 1,000,000 to avoid keystream collision with the string table.

5. Phase 2.5: Control Flow Obfuscator (_ControlFlowObfuscator) -- NEW

The _ControlFlowObfuscator is a new AST transformer in v2.0.0 that restructures the program's control flow to defeat static analysis.

Opaque Predicates (_wrap_condition_in_closure)

Every if/else condition is wrapped in a lambda closure:

```
# Before:
if condition:
    ...

# After:
if (lambda: [condition])()[0]:
    ...
```

The lambda returns a single-element list containing the original condition. Calling the lambda and subscripting [0] recovers the original value. This is semantically identical but forces decompilers and static analyzers to evaluate closures -- something most tools cannot do.

Dead Code Injection (_make_dead_code)

The obfuscator inserts if-branches with conditions that are mathematically always False. Five tautology patterns are rotated:

Pattern	Values	Why Always False
(a * b) % mod != 0	7 * 13, mod=91	7*13=91, 91%91=0
(a * b) % mod != 0	17 * 19, mod=323	17*19=323, 323%323=0
(a * b) % mod != 0	11 * 23, mod=253	11*23=253, 253%253=0
(a * b) % mod != 0	3 * 37, mod=111	3*37=111, 111%111=0
(a * b) % mod != 0	13 * 29, mod=377	13*29=377, 377%377=0

Dead code is assigned to variables like _d1, _d2, etc. The branches are inserted before ~30% of function body statements and after every 5th module-level statement (up to 3).

Visitor Coverage

- visit_If: wraps conditions in opaque predicates
- visit_FunctionDef / visit_AsyncFunctionDef: injects dead code into function bodies
- visit_Module: injects dead code at module level

6. Phase 3: Name Renamer

_NameCollector (ast.NodeVisitor)

Before renaming, _NameCollector walks the AST to collect all names defined in the source. It visits:

Node Type	Names Collected
FunctionDef / AsyncFunctionDef	Function name + all argument names (args, kwonly, posonly, *args, **kwargs)
ClassDef	Class name
Name (Store context)	Variable names being assigned to
Global / Nonlocal	All declared names
For	Loop target variables (including tuple unpacking)
ExceptHandler	Exception variable name (as e)

Rename Decision Logic (_should_rename)

For each collected name, _should_rename() applies these rules in order. The aggressive parameter controls the key behavioral change in v2.0.0:

- Is it a dunder name (__init__, etc.)? -> NO
- Is it in SAFE_NAMES (builtins/keywords/stdlib)? -> NO
- Is it in the preserve set? -> NO
- Is it defined in the source files? -> Must be YES to proceed
- If aggressive=True (--rename-all, DEFAULT): -> YES, rename it
- If aggressive=False (--safe-rename-only): starts with _? -> YES, else NO

> **Aggressive by Default (v2.0.0 Breaking Change)**
> PIPpro v2.0.0 renames ALL user-defined names by default (aggressive=True). This is a critical security improvement: public names like Scanner, run_scan, and process_result are all renamed, not just _-prefixed internals. Use --safe-rename-only for v1.x behavior. The preserve list (--preserve, --preserve-file) is essential for protecting API contracts and cross-module imports.

_Renamer (ast.NodeTransformer)

The _Renamer applies the computed rename map to these AST nodes:

- FunctionDef.name, AsyncFunctionDef.name
- ClassDef.name
- Name.id (variable references and assignments)
- Function argument names (args, kwonlyargs, posonlyargs, vararg, kwarg)
- Global and Nonlocal declaration name lists
- ImportFrom alias names and asnames (intra-project imports)
- ExceptHandler variable names

Critically, the _Renamer does NOT rename attribute accesses (node.attr in ast.Attribute nodes). This prevents breaking calls to stdlib, library methods, or any dynamically resolved attributes.

Random ID Generation (_rand_id)

Mangled names are 12-character random identifiers starting with an underscore and a lowercase letter, followed by 10 alphanumeric characters. Example: _a7kx9m2p4b1q. The generator uses secrets.SystemRandom() by default (cryptographically random) or random.Random(seed) if --seed is specified.

The algorithm ensures no collisions by checking each generated ID against both the used_ids set and the original all_defined set.

7. Phase 3.5: Import Obfuscator (_ImportObfuscator) -- NEW

The _ImportObfuscator is a new AST transformer in v2.0.0 that replaces standard import statements with encrypted __import__ calls.

visit_Import

Transforms import foo into _mN = __import__(_s(idx)):

- Only processes single-name imports (multi-import skipped for safety)
- Standard library modules are preserved unchanged
- Module name is added to the encrypted string table
- Target variable is alias.asname or the top-level module name

visit_ImportFrom

Transforms from foo import bar into bar = getattr(__import__(_s(idx1)), _s(idx2)):

- Module name and each imported name are added to the string table
- Star imports (from foo import *) are preserved unchanged
- Standard library modules are preserved unchanged
- Each imported name becomes a separate getattr + __import__ call

Stdlib Detection

The _is_stdlib() method checks if a module's top-level package is in the _STDLIB_MODULES set. This set is built from sys.stdlib_module_names (Python 3.10+) plus a hardcoded fallback of ~100 common stdlib modules. This prevents obfuscating imports of os, sys, hashlib, etc., which would break the runtime since __import__ would receive encrypted names for modules that the encryptor preamble itself depends on.

8. Phase 4: Nuitka Compiler (compile_binary)

The compile_binary() function wraps Nuitka invocation with proper argument construction, timeout handling, and binary detection.

Nuitka Discovery

- First checks PATH for nuitka or nuitka3 executables (shutil.which)
- Falls back to python -m nuitka (module invocation)
- Returns a clear error if neither is available

Command Construction

The function builds a Nuitka command line with:

- --standalone or --module depending on mode
- --output-dir for build artifacts
- --assume-yes-for-downloads (non-interactive)
- --include-data-dir / --include-data-file for bundled resources
- --include-module / --include-package for explicit inclusions
- --nofollow-import-to for exclusions
- Any extra args passed through --nuitka-args

Binary Detection

After Nuitka completes, the function locates the binary:

- Standalone mode: looks in <output>/<stem>.dist/ for <stem>.exe (Windows) or <stem> (Linux)
- Module mode: looks for <stem>.pyd (Windows) or <stem>.so (Linux)
- Handles Nuitka's .bin suffix on some Linux configurations

Timeout

Compilation timeout is set to 1800 seconds (30 minutes), which is sufficient for large projects. The duration is reported on success.

9. Phase 5: Binary Verifier (verify_binary)

The verifier implements a simplified version of the Unix strings command, extracting all printable ASCII sequences from the binary and checking them against known sensitive patterns.

String Extraction Algorithm

```
for byte in binary_data:
    if 32 <= byte <= 126:    # printable ASCII
        current.append(chr(byte))
    else:
        if len(current) >= min_length:  # default 6
            extracted.append(''.join(current))
        current = []
```

This is a byte-level scan, not a string table parser. It catches strings embedded anywhere in the binary, including in data segments, string tables, and code sections.

Pattern Matching

The verifier compiles regex patterns and tests every extracted string. Built-in patterns cover system commands, remediation keys, credential patterns, and internal file paths. User-specified strings (--verify-strings) are matched case-insensitively as substrings.

Deduplication

Leak results are deduplicated by the found_in string (first 100 chars). This prevents a single repeated string from generating hundreds of identical warnings.

10. Safe Name Computation

The SAFE_NAMES set is computed once at module import time by _build_safe_names(). It contains:

Source	Count (approx)	Examples
dir(builtins)	~160	print, len, dict, int, Exception, True, None
keyword.kwlist	~35	if, for, while, def, class, return, import
keyword.softkwlist	~3	match, case, type (Python 3.12+)
sys.stdlib_module_names	~300	os, sys, ast, json, re, pathlib
Hardcoded	4	self, cls, args, kwargs

Total: approximately 500 protected names. This is computed dynamically from the running Python interpreter, so it automatically adapts to new Python versions that add builtins, keywords, or stdlib modules.

11. Rename Map Generation

The rename map generation process:

- 1. All files are parsed and _NameCollector extracts defined names across all files
- 2. Each name is tested with _should_rename(aggressive=True) to determine eligibility
- 3. Eligible names are sorted alphabetically (deterministic iteration order)
- 4. Each name gets a random 12-character ID via _rand_id()
- 5. Collision detection: new IDs must not exist in used_ids or all_defined
- 6. The complete map is applied to all files (not per-file)

The sorted iteration and global map mean that adding --seed produces identical output given identical input. This is essential for reproducible builds in CI/CD pipelines.

Note: with --rename-all (default), the rename map typically contains 3-5x more entries than v1.x's underscore-only mode, because ALL user-defined names (Scanner, process_host, result_data, etc.) are now included alongside traditional _-prefixed internals.

12. Already-Obfuscated Detection (PIPpro Signature)

PIPpro v2.0.0 detects already-obfuscated files using a new detection mechanism with backward compatibility:

- Primary: checks for _PP = '# PIPpro-obfuscated' signature in first 500 chars
- Primary: checks for '# PIPpro-obfuscated' string in first 500 chars

- Legacy: checks if first non-whitespace line starts with '_K = b' or '_K=b' (v1.x XOR key)

The PIPpro signature is injected during preamble generation as an assignment: _PP = '# PIPpro-obfuscated'. After renaming, _PP becomes a random identifier, but the string value remains detectable (it is in the first few statements of the file).

> **Why This Matters**
> Double-encrypting a file causes exponential growth in the encrypted string table because the first-pass decryptor code (which contains large bytes literals) gets encrypted again. This can cause Nuitka to consume 16+ GB of RAM during compilation. The guard prevents this scenario entirely.

13. HMAC-SHA256 Integrity Verification -- NEW

PIPpro v2.0.0 adds HMAC-SHA256 integrity verification to both the string and bytes encrypted tables. This is implemented as a runtime check on first decryption.

Build-Time

- All encrypted string entries are concatenated: concat = S[0] + S[1] + ... + S[n]
- HMAC-SHA256 is computed: _H = hmac.new(_K, concat, hashlib.sha256).digest()
- Same process for bytes table: _HB = hmac.new(_K, concat_B, hashlib.sha256).digest()

Runtime Verification

- First call to _s() checks global _V flag
- If not verified: concatenate all _S entries, compute HMAC, compare to _H
- If mismatch: raise RuntimeError('integrity') -- application crashes immediately
- If match: set _V = True, never re-verify
- Bytes table has independent verification via _VB and _HB

Security Properties

- Detects modification of any encrypted table entry (even a single bit flip)
- Detects modification of the HMAC digest itself (key-dependent)
- Detects modification of the master key (all HMACs fail)
- Does not protect against complete replacement (key + tables + HMACs)

The HMAC verification adds minimal runtime overhead: it runs exactly once (on first string access) and then the flag prevents re-checking.

14. Secure Temp Staging & Strict Mode -- NEW

Secure Temp Staging

When --compile is used without --output-dir or --in-place, PIPpro creates a secure temporary directory using tempfile.mkdtemp(prefix='pippro_'). An atexit handler (atexit.register(shutil.rmtree, staging_dir, ignore_errors=True)) ensures the directory is cleaned up on process exit, including abnormal termination via exception or signal.

Strict Mode

When --strict is enabled (or implied by --compile), any file processing error raises RuntimeError immediately instead of being collected in the errors list. This ensures:

- Compilation never proceeds with partially-obfuscated source
- CI/CD pipelines fail fast on any obfuscation error
- Error messages include the specific file and exception details

15. Security Properties & HallMonitor Scorecard

What PIPpro v2.0.0 Protects Against

Attack	Protection Level	How
Source code theft	High	Source compiled to native binary (Phase 4)
String extraction (strings cmd)	High	SHA-256 stream cipher encrypted (Phase 2)
String table tampering	High	HMAC-SHA256 integrity check (Phase 2)
Symbol name analysis	High	ALL names randomized by default (Phase 3)

Static analysis / decompilation	High	Opaque predicates + dead code (Phase 2.5)
Dependency graph analysis	Medium+	Import statements encrypted (Phase 3.5)
Decompilation (uncompyle6)	High	Native binary, no .pyc (Phase 4)
Documentation mining	High	All docstrings/annotations stripped (Phase 1)
Binary string scanning	Medium+	Verified clean by Phase 5 scanner

What PIPpro Does NOT Protect Against

- Runtime memory dumping (strings exist in RAM during execution)
- Debugging/tracing of the compiled binary (gdb, WinDbg)
- Dynamic analysis / API call tracing
- Attribute name recovery (obj.method_name is never renamed)

PIPpro is a deterrence tool, not a DRM system. It raises the cost of reverse engineering from trivial (reading .py files) to expensive (binary analysis of native code with SHA-256-encrypted strings, HMAC-protected tables, obfuscated control flow, and hidden import graphs). For most commercial scenarios, this cost increase is sufficient.

HallMonitor Security Scorecard

Metric	v1.2.0 Score	v2.0.0 Score	Improvement
Overall	4.4 / 10	7.4 / 10	+3.0 (+68%)
Encryption Strength	2 / 10	8 / 10	XOR -> SHA-256 stream cipher
Integrity Protection	0 / 10	9 / 10	None -> HMAC-SHA256
Name Coverage	3 / 10	8 / 10	_-only -> ALL names
Control Flow	0 / 10	6 / 10	None -> opaque predicates
Import Protection	0 / 10	6 / 10	None -> encrypted __import__
Build Security	5 / 10	8 / 10	Strict mode + temp cleanup

16. Performance Characteristics

Obfuscation (Phases 1-3.5)

- Processing time: ~1-3ms per file (AST parse + transform + unparse)
- Control flow obfuscation adds ~0.5ms per file
- Import obfuscation adds ~0.2ms per file
- Memory: proportional to largest file AST size
- Output size: similar to input (encrypted tables add ~20-40% for string-heavy files)

Runtime Overhead

- First access to each string: SHA-256 stream derivation + XOR + dict store
- HMAC verification: one-time cost on first _s() or _b() call (~1ms for typical tables)
- Subsequent accesses: dictionary lookup (O(1))
- Steady-state overhead: negligible (all strings cached after initialization)
- Control flow overhead: negligible (lambda + subscript per if-condition)

Compilation (Phase 4)

- Build time: 2-30 minutes depending on project size and dependencies
- Binary size: 20-60 MB for standalone (includes Python interpreter)
- Module size: 0.5-5 MB for .so/.pyd

17. Competitive Analysis

Feature	PIPpro v2.0.0	PyArmor	Cython	Nuitka (alone)
String encryption	SHA-256 stream +	Custom VM	No	No
Name mangling	ALL names (default)	Yes (all)	No	No
Control flow obfusc.	Yes (opaque pred.)	Yes (VM)	No	No
Import obfuscation	Yes (__import__)	No	No	No

Docstring strip	Yes	Yes	N/A	No
f-string encryption	Yes	No	No	No
Integrity checking	HMAC-SHA256	VM integrity	No	No
Native compilation	Via Nuitka	No	Yes (.so)	Yes (.exe)
Binary verification	Built-in	No	No	No
Zero dependencies	Yes (core)	No (runtime)	No (Cython)	No (C compiler)
Strict CI/CD mode	Yes	No	No	No
Composable pipeline	Yes (7 phases)	Partial	No	N/A
Deterministic output	Yes (--seed)	No	Yes	Yes
Already-obfusc. guard	Yes (signature)	No	N/A	N/A
Preserve lists	Yes (flexible)	Limited	N/A	N/A

Key Differentiators

· **Composability:** PIPpro is designed to work WITH Nuitka, not instead of it. The obfuscation phases (1-3.5) add protection that Nuitka alone cannot provide (SHA-256 string encryption, HMAC integrity, aggressive name mangling, control flow obfuscation, import obfuscation), then Nuitka adds protection that PIPpro alone cannot provide (native compilation).

· **Zero Dependencies:** The core obfuscation engine uses only Python's standard library (ast, hashlib, hmac, secrets). No pip install, no runtime libraries, no license servers. This makes PIPpro suitable for air-gapped environments and restrictive CI/CD pipelines.

· **f-string Encryption:** PIPpro is the only tool that encrypts constant parts within f-strings. Other tools either skip f-strings entirely or only encrypt simple string literals.

· **Built-in Verification:** No other tool includes a binary leak scanner. PIPpro verifies its own output, closing the loop between obfuscation and binary delivery.

· **HallMonitor-Hardened:** v2.0.0 is the result of a systematic security audit that identified and addressed 8 findings. The scorecard improved from 4.4/10 to 7.4/10 -- a 68% improvement in measured security posture.

18. Source File Reference

File	Purpose
build_tools/obfuscate_source.py	PIPpro v2.0.0 engine -- 7 phases, CLI, HallMonitor-hardened
build_tools/preserve_names.txt	Default preserve list for ForteFide builds (74 names)
build_obfuscated.bat	Windows batch script: backup, obfuscate, Nuitka build, restore
pippro_installer.iss	Inno Setup installer config (Windows .exe)
build_pippro_deb.py	Debian .deb package builder (pure Python, no dpkg-deb)
dist_pippro/	Nuitka compilation output directory

Version History

Version	Changes
1.0.0	Initial release: strip, encrypt (XOR), rename (phases 1-3)
1.1.0	Added Phase 4 (Nuitka compiler) and Phase 5 (binary verifier)
1.2.0	f-string encryption, double-obfuscation guard, dedup cache, bytes literal encryption, module compilation mode
2.0.0	HallMonitor-hardened: SHA-256 stream cipher, HMAC-SHA256 integrity, aggressive rename (--rename-all default), control flow obfuscation (Phase 2.5), import obfuscation (Phase 3.5), strict mode, secure temp staging, PIPpro signature detection. Score: 4.4 -> 7.4/10