FastPass - Complete Project Specification

Document Purpose & Maintenance Protocol: This document serves as the authoritative, self-documenting specification for FastPass. It provides complete context to future AI instances and developers about:

- **Current project status** and implementation details
- **Architecture decisions** and technical solutions
- **Lessons learned** from development challenges
- **Usage patterns** and deployment instructions
- **Complete change history** and evolution of the project

Maintenance Requirement: This document MUST be updated whenever significant changes are made to the codebase, architecture, or functionality. It should always reflect the current state of the project and serve as the single source of truth for understanding the entire system.

Project Mission & Purpose

FastPass is a command-line tool that provides universal file encryption and decryption capabilities across multiple file formats. It serves as a unified front-end wrapper for specialized crypto tools (msoffcrypto-tool, PyPDF2) to add or remove password protection from Microsoft Office documents and PDF files.

Core Problem Solved: Eliminates the need to learn and manage multiple separate tools for file encryption/decryption across different formats. Provides a consistent, secure interface for password protection operations while maintaining file integrity and implementing enterprise-grade security practices.

Key Differentiator: Unified CLI interface with enterprise security patterns including file isolation, in-memory validation, password list support, and secure password handling. Follows proven architecture patterns with “it just works” simplicity for reliability and security.

Product Requirements Document (PRD)

Project Overview

- **Project Name:** FastPass
- **Version:** v1.0
- **Target Platform:** Windows Desktop (CLI) with cross-platform Python support
- **Technology Stack:** Python, msoffcrypto-tool, PyPDF2, filetype library, pathlib
- **Timeline:** Development in progress
- **Team Size:** Single developer maintained

Target Users

- **Primary Users:** IT administrators, security professionals, business users
- **Secondary Users:** Developers, system integrators, automation script writers
- **User Experience Level:** Intermediate (comfortable with command-line tools)
- **Use Cases:** Batch file encryption, automated security workflows, document protection

Feature Specifications

Core Functionality

☒ Universal file encryption/decryption interface
☒ Microsoft Office document password protection (modern and legacy formats)
☒ PDF password protection and removal

☒ Batch processing for multiple files
☒ Recursive directory processing with in-place or copy modes
☒ Automatic file format detection using filetype library
☒ Direct import strategy for simplified code management

Security & File Safety

☒ File format validation using filetype library (simplified magic number checking)
☒ Path traversal attack prevention with whitelist approach
☒ Secure temporary file creation with proper permissions (0o600)
☒ Password memory clearing and secure handling
☒ Error message sanitization to prevent information disclosure
☒ Legacy Office format protection (decrypt-only limitation documented)

Password Management

☒ Per-file password specification with automatic pairing
☒ Password management with multiple password support
☒ Password list file support for batch operations
☒ JSON password input via stdin for GUI integration
☒ Secure password handling and memory cleanup
☒ Password validation before file processing

File Operations

☒ In-place modification with validation-based safety
☒ Output directory specification for batch operations
☒ File integrity verification after operations
☒ Duplicate filename handling and conflict resolution
☒ Comprehensive cleanup of temporary files

Utility Features

☒ Dry-run mode for testing operations
☒ File format support detection
☒ Password requirement checking

- ☒ Batch operation progress reporting
- ☒ Detailed logging with debug mode

Success Metrics

- **Performance Targets:** File processing < 10 seconds for typical business documents
- **User Experience:** Zero data loss through validation, "it just works" simplicity, clear error messages
- **Reliability:** 99.9% successful completion rate for valid inputs
- **Security:** No password exposure in logs, secure temporary file handling

Constraints & Assumptions

- **Technical Constraints:** Requires underlying crypto libraries (msoffcrypto-tool, PyPDF2) to be available
- **Platform Constraints:** Cross-platform compatible with pure Python dependencies
- **Security Constraints:** Must maintain file confidentiality and integrity throughout operations
- **User Constraints:** Must have appropriate file permissions for input and output directories
- **Assumptions:** Users understand file encryption concepts and password management practices

Project Directory Structure

```
fast_pass/
├── src/                        # Main source code
│   ├── __init__.py
│   ├── __main__.py             # Makes package executable with 'python -m
src'
│   ├── cli.py                  # CLI argument parsing and validation
│   ├── core/                   # Core business logic
│   │   ├── __init__.py
│   │   ├── file_handler.py     # File processing pipeline
│   │   ├── security.py         # Security validation and path checking
│   │   ├── crypto_handlers/    # Crypto tool integrations
│   │   │   ├── __init__.py
│   │   │   ├── office_handler.py # msoffcrypto-tool integration
│   │   │   └── pdf_handler.py    # PyPDF2 integration
│   │   └── password/           # Password handling modules
│   │       ├── __init__.py
│   │       ├── password_manager.py # Password validation and management
│   │       └── password_list.py    # Password list file handling
│   └── utils/                  # Utility modules
│       ├── __init__.py
│       ├── logger.py           # Logging configuration
│       └── config.py           # Configuration management
├── tests/                      # Test suite
```

```
│   ├── __init__.py
│   ├── test_cli.py
│   ├── test_core.py
│   ├── test_crypto_handlers.py
│   ├── test_security.py
│   ├── test_password_handling.py
│   └── test_integration.py
├── dev/                          # Development documentation
│   └── fast_pass_specification.md
├── requirements.txt              # Python dependencies
├── requirements-dev.txt          # Development dependencies
├── setup.py                      # Package setup
└── README.md                     # User documentation
```

Python Dependencies

Requirements (requirements.txt):

```
msoffcrypto-tool>=5.0.0    # Office document encryption/decryption
PyPDF2>=3.0.0              # PDF processing and encryption
filetype>=1.2.0            # File type detection (replaces python-magic)
```

PyInstaller Integration Notes: - All Python packages will be bundled into executable - No external binaries required - pure Python dependencies - Direct imports for simplified code management

Password Handling Architecture

Simplified password management without reuse concept:

```python
# Password handling with multiple sources and priority algorithm
class PasswordManager:
    def __init__(self):
        self.cli_passwords = []  # Space-delimited passwords from CLI
        self.password_list_file = None  # Path to password list file
        self.password_list = []  # Passwords loaded from file

    def load_password_sources(self, args):
        """Load passwords from all sources"""
        # Space-delimited passwords from CLI
        if hasattr(args, 'cli_passwords') and args.cli_passwords:
            self.cli_passwords = [p for p in args.cli_passwords if p !=
'stdin']

        # Password list file
        if args.password_list:
            self.password_list_file = args.password_list
            self.password_list = self._load_password_list()

    def get_password_candidates(self, file_path):
        """Get prioritized list of passwords to try for a file"""
        candidates = []

        # Priority 1: CLI -p passwords (in order provided)
        candidates.extend(self.cli_passwords)

        # Priority 2: Password list file (line by line)
        candidates.extend(self.password_list)

        # Remove duplicates while preserving order
        return list(dict.fromkeys(candidates))

    def _load_password_list(self):
        """Load passwords from text file, one per line"""
        try:
            with open(self.password_list_file, 'r', encoding='utf-8') as f:
                return [line.strip() for line in f if line.strip()]
        except FileNotFoundError:
            self.log_error(f"Password list file not found:
{self.password_list_file}")
            return []

    def find_working_password(self, file_path, crypto_handler):
        """Find working password for file by trying all candidates"""
        candidates = self.get_password_candidates(file_path)

        for password in candidates:
            if crypto_handler.test_password(file_path, password):
                return password

        return None
```

Command Line Reference

```
Usage: fast_pass {encrypt|decrypt} [options]

Required Arguments:
  encrypt                  Add password protection to files
  decrypt                  Remove password protection from files

File Input Options:
  -i, --input FILE...      Files to process (space-delimited, quotes for
spaces)
  -r, --recursive DIR      Process directory recursively (decrypt/check-
password only)

Password Options:
  -p, --password PASS...   Passwords to try (space-delimited, quotes for
spaces)
```

```
  --password-list FILE     Text file with passwords to try (one per line)
  -p stdin                 Read passwords from JSON via stdin (GUI
integration)
  --check-password [FILE]  Check if file requires password (dry-run mode)

Output Options:
  -o, --output-dir DIR     Output directory (default: in-place modification)

Utility Options:
  --dry-run               Show what would be done without making changes
  --verify                Deep verification of processed files
  --list-supported        List supported file formats
  --debug                 Enable detailed logging and debug output
  -h, --help              Show this help message
  -v, --version           Show version information

Supported File Formats:
  Modern
Office:     .docx, .xlsx, .pptx, .docm, .xlsm, .pptm, .dotx, .xltx, .potx
                     (Encryption: experimental support, Decryption: full
support)
  Legacy Office:     .doc, .xls, .ppt (NOT SUPPORTED - use Office to convert
to modern format)
  PDF Files:         .pdf (Full encryption and decryption support)
Examples:
  # Encrypt single file with password
  fast_pass encrypt -i contract.docx -p "mypassword"

  # Decrypt multiple files with same password
  fast_pass decrypt -i file1.pdf file2.docx file3.xlsx -p "shared_pwd"

  # Multiple passwords via CLI (space-delimited, tries all passwords on all
files)
  fast_pass decrypt -i file1.pdf file2.docx -p "password123" "secret456"
"admin789"

  # Passwords with spaces (quoted)
  fast_pass decrypt -i protected.pdf -p "password1" "password 2" "ps3"

  # Password list file for batch operations
  fast_pass decrypt -i "archive_folder/report1.pdf"
"archive_folder/report2.pdf" --password-list common_passwords.txt

  # Combined approach: specific password + password list fallback
  fast_pass decrypt -i urgent.pdf "archive1.pdf" "archive2.pdf" -p
"urgent_pwd" --password-list common_passwords.txt

  # Passwords from stdin JSON (GUI integration)
  fast_pass decrypt -i file1.pdf file2.docx -p stdin < passwords.json
  # JSON format: {"file1.pdf": "secret1", "file2.docx": "secret2"}
  # Recursively process directory (decrypt only)
  fast_pass decrypt -r ./encrypted_docs/ -p "main_password"

  # Recursive with password list
  fast_pass decrypt -r ./archive/ --password-list passwords.txt

  # Check password protection status (dry-run)
  fast_pass --check-password -r ./documents/ --password-list
test_passwords.txt

  # Mixed file types with output directory
  fast_pass encrypt -i report.pdf data.xlsx presentation.pptx -p "secret"
-o ./secured/

  # Security: Use custom allowed directories
  fast_pass decrypt -i ./restricted/report.pdf --allowed-dirs
/home/user/restricted /tmp -p "password123"

Exit Codes:
  0  Success
  1  General error (file access, crypto tool failure)
  2  Invalid arguments or command syntax
  3  Security violation (path traversal, invalid format)
  4  Password error (wrong password, authentication failure)
```

High-Level Architecture Overview - Core Processing Flow

> 💡 **IMPLEMENTATION CRITICAL**: This pseudocode provides the master reference for code organization. Every code block must map to a specific element ID (e.g., `# A1a`, `# B3c`, etc.)

```python
# MAIN PROGRAM ENTRY POINT
def main():
    """FastPass main entry point with complete error handling"""
    try:
        # A: CLI Parsing & Initialization
        args = parse_command_line_arguments()
        if args.help or args.version or args.list_supported:
            display_information_and_exit(args)  # Exit code 0

        # B: Security & File Validation
        validated_files = perform_security_and_file_validation(args)

        # C: Crypto Tool Selection & Configuration
        crypto_handlers =
setup_crypto_tools_and_configuration(validated_files)

        # D: File Processing & Operations
        processing_results = process_files_with_crypto_operations(
            validated_files, crypto_handlers, args
        )

        # E: Cleanup & Results Reporting
        exit_code = cleanup_and_generate_final_report(processing_results)
        sys.exit(exit_code)

    except SecurityViolationError as e:
        log_sanitized_error(e)
        sys.exit(3)  # Security violation
    except FileFormatError as e:
        log_error(f"File format error: {e}")
        sys.exit(1)  # Format/access error
    except CryptoToolError as e:
        log_error(f"Crypto tool unavailable: {e}")
        sys.exit(1)  # Tool availability error
    except ProcessingError as e:
        cleanup_partial_processing_on_failure()
        sys.exit(1)  # Processing failure
    except Exception as e:
        log_error(f"Unexpected error: {e}")
        sys.exit(2)  # General error

# CONFIGURATION MANAGEMENT SYSTEM
class FastPassConfig:
    """Configuration management with multiple sources and precedence"""
    VERSION = "1.0.0"
    MAX_FILE_SIZE = 500 * 1024 * 1024  # 500MB
    TEMP_DIR_PREFIX = "fastpass_"
    SECURE_FILE_PERMISSIONS = 0o600
    SUPPORTED_FORMATS = {
        '.docx': 'msoffcrypto',
        '.xlsx': 'msoffcrypto',
        '.pptx': 'msoffcrypto',
        '.docm': 'msoffcrypto',
        '.xlsm': 'msoffcrypto',
        '.pptm': 'msoffcrypto',
        '.dotx': 'msoffcrypto',
        '.xltx': 'msoffcrypto',
        '.potx': 'msoffcrypto',
        '.pdf': 'PyPDF2'
    }

    # Configuration file locations (in order of precedence)
    CONFIG_LOCATIONS = [
        Path.home() / '.fastpass' / 'config.json',  # User config
        Path.cwd() / 'fastpass.json',               # Project config
        Path(__file__).parent / 'config.json'      # Default config
    ]
    @classmethod
    def load_configuration(cls, cli_args: argparse.Namespace) -> Dict[str,
Any]:
        """Load configuration from multiple sources with precedence"""
        config = cls._get_default_config()

        # 1. Load from config files (lowest precedence)
        for config_path in cls.CONFIG_LOCATIONS:
            if config_path.exists():
                try:
                    with open(config_path, 'r') as f:
                        file_config = json.load(f)
                        config.update(file_config)
                except (json.JSONDecodeError, IOError) as e:
                    print(f"Warning: Could not load config from
{config_path}: {e}")

        # 2. Load from environment variables
        env_config = cls._load_from_environment()
        config.update(env_config)

        # 3. Override with CLI arguments (highest precedence)
        cli_config = cls._extract_cli_config(cli_args)
        config.update(cli_config)

        return config

    @classmethod
    def _get_default_config(cls) -> Dict[str, Any]:
        """Default configuration values"""
        return {
            'max_file_size': cls.MAX_FILE_SIZE,
            'temp_dir_prefix': cls.TEMP_DIR_PREFIX,
            'secure_permissions': cls.SECURE_FILE_PERMISSIONS,
            'supported_formats': cls.SUPPORTED_FORMATS.copy(),
            'log_level': 'INFO',
            'log_file': None,
            'cleanup_on_error': True,
            # Security hardening settings
            'allow_cwd': False,  # Default: do not allow current directory
access
            'max_password_length': 1024,
            'max_json_size': 1024 * 1024,  # 1MB limit for password JSON
            'max_path_length': 4096,
            'enable_secure_deletion': True,
            'symlink_protection': True,
            'xml_entity_protection': True
        }
    @classmethod
    def _load_from_environment(cls) -> Dict[str, Any]:
        """Load configuration from environment variables"""
        import os
        config = {}

        # Environment variable mapping
        env_mapping = {
            'FASTPASS_MAX_FILE_SIZE': ('max_file_size', int),
            'FASTPASS_LOG_LEVEL': ('log_level', str),
            'FASTPASS_LOG_FILE': ('log_file', str),
            'FASTPASS_CLEANUP_ON_ERROR': ('cleanup_on_error', bool),
            # Security environment variables
            'FASTPASS_ALLOW_CWD': ('allow_cwd', bool),
            'FASTPASS_MAX_PASSWORD_LENGTH': ('max_password_length', int),
            'FASTPASS_MAX_JSON_SIZE': ('max_json_size', int),
            'FASTPASS_ENABLE_SECURE_DELETION': ('enable_secure_deletion',
bool),
            'FASTPASS_SYMLINK_PROTECTION': ('symlink_protection', bool),
            'FASTPASS_XML_ENTITY_PROTECTION': ('xml_entity_protection', bool)
        }

        for env_var, (config_key, type_func) in env_mapping.items():
            if env_var in os.environ:
                try:
                    if type_func == bool:
                        config[config_key] = os.environ[env_var].lower() in
('true', '1', 'yes')
                    else:
                        config[config_key] = type_func(os.environ[env_var])
                except ValueError as e:
                    print(f"Warning: Invalid environment variable {env_var}:
{e}")

        return config

    @classmethod
    def _extract_cli_config(cls, cli_args: argparse.Namespace) -> Dict[str,
Any]:
        """Extract configuration from CLI arguments"""
        config = {}

        if hasattr(cli_args, 'debug') and cli_args.debug:
            config['log_level'] = 'DEBUG'

        if hasattr(cli_args, 'output_dir') and cli_args.output_dir:
            config['output_directory'] = cli_args.output_dir

        return config
```

```python
# CUSTOM EXCEPTION CLASSES
class SecurityViolationError(Exception): pass
class FileFormatError(Exception): pass
class CryptoToolError(Exception): pass
class ProcessingError(Exception): pass
```

Section A: CLI Parsing & Initialization

> **CODE MAPPING CRITICAL**: Each element below corresponds to specific code blocks that must be labeled with the exact IDs shown (e.g., `# A1a: sys.argv processing`)

```python
# A1: COMMAND LINE ARGUMENT PARSING
def parse_command_line_arguments() -> argparse.Namespace:
    import sys
    import argparse
    from pathlib import Path
    from typing import List, Optional

    # A1a: Create argument parser with custom actions
    parser = argparse.ArgumentParser(
        prog='fast_pass',
        description='FastPass - Secure file encryption/decryption tool',
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog='''
Examples:
  fast_pass encrypt -i file.docx -p mypassword
  fast_pass decrypt -i file.pdf --password-list passwords.txt
  fast_pass encrypt -i "file1.xlsx" "file2.xlsx" -o ./encrypted/
        '''
    )

    # A1b: Operation mode (encrypt XOR decrypt)
    operation_group = parser.add_mutually_exclusive_group(required=True)
    operation_group.add_argument('-e', '--encrypt', action='store_true',
                                help='Encrypt files')
    operation_group.add_argument('-d', '--decrypt', action='store_true',
                                help='Decrypt files')

    # A1c: File input options (explicit file specification required)
    parser.add_argument('-i', '--input', nargs='*', type=str, dest='files',
                       help='Files to process (space-delimited, quotes for
spaces)')
    parser.add_argument('-r', '--recursive', type=Path, metavar='DIR',
                       help='Process directory recursively (decrypt/check-
password only)')

    # A1d: Password options with space-delimited support
    parser.add_argument('-p', '--password', nargs='*', type=str,
dest='cli_passwords',
                       help='Passwords to try (space-delimited, quotes for
spaces, or "stdin" for JSON input)')
    parser.add_argument('--password-list', type=Path,
                       help='File containing passwords (one per line)')

    # A1e: Output options
    parser.add_argument('-o', '--output-dir', type=Path,
                       help='Output directory (default: in-place)')

    # A1f: Utility options
    parser.add_argument('--dry-run', action='store_true',
                       help='Show what would be done without making changes')
    parser.add_argument('--verify', action='store_true',
                       help='Deep verification of processed files')
    parser.add_argument('--list-supported', action='store_true',
                       help='List supported file formats')
    parser.add_argument('--debug', action='store_true',
                       help='Enable detailed logging')
    parser.add_argument('--log-file', type=Path,
                       help='Write logs to specified file')
    parser.add_argument('--report-format', choices=['text', 'json', 'csv'],
                       default='text', help='Output report format')

    # A1g: Security options
    parser.add_argument('--allowed-dirs', nargs='+', type=str,
                       help='Additional directories to allow for file access
(default: home directory and current working directory)')
    parser.add_argument('-v', '--version', action='version',
                       version=f'FastPass {FastPassConfig.VERSION}')

    # A1h: Parse arguments with error handling
    try:
        args = parser.parse_args()
    except SystemExit as e:
        if e.code != 0:
            sys.exit(2)  # Invalid arguments
        raise

    # A1h: Handle special modes
    if args.list_supported:
        display_supported_formats()
        sys.exit(0)

    return args

# A2: ARGUMENT VALIDATION AND NORMALIZATION
def validate_operation_mode_and_arguments(args: argparse.Namespace) ->
argparse.Namespace:
    from pathlib import Path
    # A2a: Ensure files or recursive specified
    if not args.files and not args.recursive:
        raise ValueError("Must specify files or --recursive directory")

    if args.files and args.recursive:
        raise ValueError("Cannot specify both files and --recursive")

    # A2a1: Restrict recursive mode to decrypt and check-password only
    if args.recursive and args.encrypt:
        raise ValueError("Recursive mode is only supported for decrypt
operations (security restriction)")

    # A2b: Process explicit file paths and normalize (no glob pattern
support)
    if args.files:
        # Convert to Path objects and resolve (explicit file specification
only)
        args.files = [Path(f).expanduser().resolve() for f in args.files]

    if args.recursive:
        args.recursive = Path(args.recursive).expanduser().resolve()
        if not args.recursive.is_dir():
            raise ValueError(f"Recursive path is not a directory:
{args.recursive}")

    # A2c: Validate output directory
    if args.output_dir:
        args.output_dir = Path(args.output_dir).expanduser().resolve()
        if args.output_dir.exists() and not args.output_dir.is_dir():
            raise ValueError(f"Output path exists but is not a directory:
{args.output_dir}")

    # A2d: Set operation mode flag
    args.operation = 'encrypt' if args.encrypt else 'decrypt'

    return args

# A3: LOGGING SYSTEM INITIALIZATION
def setup_logging_and_debug_infrastructure(args: argparse.Namespace) ->
logging.Logger:
    import logging
    import sys
    from datetime import datetime

    # A3a: Configure logging with TTY detection
    log_level = logging.DEBUG if args.debug else logging.INFO

    # A3a-1: Configure console logging
    console_handler = logging.StreamHandler(sys.stderr)
    # Check if stderr is a TTY for appropriate formatting
    if sys.stderr.isatty():
        console_format = '%(asctime)s - %(levelname)s - %(message)s'
    else:
        # Non-TTY output (e.g., redirected to file) - simpler format
        console_format = '%(levelname)s: %(message)s'

    console_handler.setFormatter(logging.Formatter(console_format))
    console_handler.setLevel(log_level)

    # A3a-2: Configure file logging if specified
    handlers = [console_handler]

    if hasattr(args, 'log_file') and args.log_file:
        try:
            # Ensure log directory exists
            args.log_file.parent.mkdir(parents=True, exist_ok=True)

            file_handler = logging.FileHandler(args.log_file, mode='a')
            file_format = '%(asctime)s - %(name)s - %(levelname)s - %
(message)s'
            file_handler.setFormatter(logging.Formatter(file_format))
            file_handler.setLevel(logging.DEBUG)  # Always debug level for
files
            handlers.append(file_handler)

        except Exception as e:
            print(f"Warning: Could not set up file logging to
{args.log_file}: {e}")

    # A3a-3: Configure root logger
    logger = logging.getLogger('fastpass')
    logger.setLevel(log_level)
    logger.handlers.clear()  # Remove any existing handlers

    for handler in handlers:
        logger.addHandler(handler)

    logger = logging.getLogger('fastpass')

    # A3b: Log startup
    logger.info(f"FastPass v{FastPassConfig.VERSION} starting - operation:
{args.operation}")

    return logger

def handle_password_input_sources(args: argparse.Namespace) -> None:
    """A3c: Handle TTY detection and stdin password input"""
    import sys
    import json

    # Check if 'stdin' is specified in CLI passwords
    if args.cli_passwords and 'stdin' in args.cli_passwords:
        if sys.stdin.isatty():
            raise ValueError("Cannot read JSON from stdin: terminal input
detected")

        try:
            # Read JSON password mapping from stdin
            stdin_data = sys.stdin.read()
            password_mapping = json.loads(stdin_data)

            # Remove 'stdin' from CLI passwords and store mapping
            args.cli_passwords.remove('stdin')
            args.stdin_password_mapping = password_mapping

        except json.JSONDecodeError as e:
            raise ValueError(f"Invalid JSON in stdin password input: {e}")
        except Exception as e:
            raise ValueError(f"Error reading password input from stdin: {e}")
    else:
        args.stdin_password_mapping = {}

# A4: CRYPTO TOOL AVAILABILITY DETECTION
def initialize_crypto_tool_detection() -> Dict[str, bool]:
    import subprocess
    import importlib

    crypto_tools = {}

    # A4a: Test msoffcrypto-tool availability
    try:
        result = subprocess.run(['python', '-m', 'msoffcrypto.cli', '--
version'],
                              capture_output=True, timeout=10)
        crypto_tools['msoffcrypto'] = result.returncode == 0
    except (subprocess.TimeoutExpired, FileNotFoundError):
        crypto_tools['msoffcrypto'] = False

    # A4b: Test PyPDF2 availability
    try:
        importlib.import_module('PyPDF2')
        crypto_tools['PyPDF2'] = True
    except ImportError:
        crypto_tools['PyPDF2'] = False

    # A4c: Check for missing required tools
    required_tools = []
    if not crypto_tools.get('msoffcrypto'):
            required_tools.append('msoffcrypto-tool')
        if not crypto_tools.get('PyPDF2'):
            required_tools.append('PyPDF2')

        if required_tools:
            raise CryptoToolError(f"Missing required tools: {',
'.join(required_tools)}")

        return crypto_tools

# A5: FASTPASS APPLICATION CLASS
class FastPassApplication:
    def __init__(self, args: argparse.Namespace, logger: logging.Logger):
        # A5a: Initialize instance variables
        self.args = args
        self.logger = logger
        self.operation_mode = args.operation
        self.crypto_tools = initialize_crypto_tool_detection()

        # A5b: File tracking lists
        self.temp_files_created: List[Path] = []
        self.processing_results: Dict[Path, str] = {}
        self.operation_start_time = datetime.now()

        # A5c: Load configuration and initialize secure password manager
        self.config = FastPassConfig.load_configuration(args)
        self.password_manager = SecurePasswordManager()
        self.password_manager.load_password_sources(args)

        # A5d: Initialize secure temporary file manager
        self.temp_file_manager = SecureTempFileManager()

        # A5e: State flags
        self.ready_for_processing = True

        self.logger.debug("FastPass application initialized successfully")
```

What's Actually Happening: - A1: Command Line Argument Processing - `sys.argv` processing with explicit file specification - Individual file paths specified directly: `fast_pass encrypt "file1.docx" "file2.pdf"` - Quoted paths for files with spaces: `fast_pass encrypt "my documents/file.txt"` - `args.operation` contains 'encrypt' or 'decrypt' as positional argument - `args.files` becomes list of explicitly specified file paths - `args.cli_passwords` contains space-delimited passwords from -p flag - `args.stdin_password_mapping` contains JSON password mapping if '-p stdin' used

- **A2: Operation Mode & File Path Validation**
  - Validate operation: `args.operation` must be 'encrypt' or 'decrypt'
  - Input validation: must have `args.files` or `args.recursive` (not both unless combining)

    - File existence check: `os.path.exists(file_path)` for each input file or directory
    - Path normalization: `os.path.abspath(os.path.expanduser(file_path))`
    - Per-file password pairing: associate each file with its -p password argument
    - Password source validation: ensure passwords available from CLI, list file, or stdin
    - Build file list: `self.input_files = [{'path': Path, 'password': str, 'source': str}]`
    - Special modes: `--check-password`, `--list-supported` bypass normal password requirements
- **A3: Logging System Configuration with TTY Detection**
    - `sys.stderr.isatty()` detection for appropriate log formatting
    - TTY output: Full timestamp format `'%(asctime)s - %(levelname)s - %(message)s'`
    - Non-TTY output: Simple format `'%(levelname)s: %(message)s'` for file redirection
    - `logging.basicConfig()` with `level=logging.DEBUG` if `args.debug` enabled
    - Handler: `sys.stderr` for console output, doesn't interfere with stdout
    - Password input validation: Check `sys.stdin.isatty()` when '-p stdin' specified
    - JSON password parsing: Parse stdin JSON for per-file password mapping
    - TTY safety: Prevent accidental password exposure in terminal input
- **A4: Crypto Library Availability Detection**
    - Test msoffcrypto-tool: `import msoffcrypto` with ImportError handling
    - Test PyPDF2: `import PyPDF2` with version compatibility check
    - Store availability: `self.crypto_tools = {'msoffcrypto': bool, 'pypdf2': bool}`
    - If required libraries missing: exit with helpful installation instructions
- **A5: Configuration & Default Setup**
    - `self.config = {'backup_suffix': '_backup_{timestamp}', 'temp_dir_prefix': 'FastPass_'}`
    - `self.config['secure_permissions'] = 0o600` (read/write owner only)
    - `self.config['max_file_size'] = 500 * 1024 * 1024` (500MB limit)
    - `self.config['supported_formats'] = {'.docx': 'msoffcrypto', '.pdf': 'pypdf2'}`
    - Password policy: `self.config['min_password_length'] = 1` (no minimum enforced)
    - Cleanup settings: `self.config['cleanup_on_error'] = True`
- **A6: FastPass Application Object Creation**
    - Main `FastPass(args)` object instantiated with parsed arguments
    - `self.operation_mode = args.operation` ('encrypt' or 'decrypt')

    - `self.password_manager = PasswordManager()`
    - `self.file_processors = {}` (will map files to appropriate crypto handlers)
    - `self.temp_files_created = []` (tracking for cleanup)
    - `self.operation_start_time = datetime.now()` for timing
    - State flags: `self.ready_for_processing = True`, `self.cleanup_required = False`

Security Hardening & Attack Vector Mitigation

> **SECURITY CRITICAL**: This section addresses comprehensive security hardening based on threat analysis and attack vector identification. Every mitigation must be implemented exactly as specified to prevent exploitation.

Attack Vector Analysis & Mitigations

1. Path Traversal Attack Prevention

Attack Scenario: Attacker uses paths like `../../../../etc/passwd` to access files outside allowed directories.

Hardened Validation:

```
def validate_path_security_hardened(file_path: Path, explicit_allow_cwd: bool
= False) -> None:
    """Enhanced path traversal protection with strict validation"""
    import os
    from pathlib import Path

    # B1-SEC-1: Canonical path resolution with symlink detection
    try:
        original_path = file_path
        resolved_path = file_path.resolve(strict=True)  # Fail if path
doesn't exist

        # B1-SEC-2: Symlink detection and protection
        if resolved_path != original_path.resolve():
            if original_path.is_symlink() or any(p.is_symlink() for p in
original_path.parents):
                raise SecurityViolationError("Symlink access denied for
security")

        # B1-SEC-3: Restricted allowed directories (NO current directory by
default)
        allowed_dirs = [Path.home().resolve()]

        # Only allow current directory if explicitly enabled
        if explicit_allow_cwd:
            allowed_dirs.append(Path.cwd().resolve())
        # B1-SEC-4: Strict containment checking
        is_allowed = False
        for base_dir in allowed_dirs:
            try:
                relative_path = resolved_path.relative_to(base_dir)
                # Additional check: no parent directory references in
resolved path
                if '..' in str(relative_path):
                    raise SecurityViolationError("Path traversal in resolved
path")
                is_allowed = True
                break
            except ValueError:
                continue

        if not is_allowed:
            raise SecurityViolationError("File access outside permitted
directories")

        # B1-SEC-5: Path component analysis (enhanced)
        forbidden_components = ['..', '.', '', '~']
        for component in original_path.parts:
            if component in forbidden_components:
                raise SecurityViolationError(f"Forbidden path component:
{component}")
            if component.startswith('.') and component not in ['.docx',
'.xlsx', '.pptx', '.pdf']:
                raise SecurityViolationError("Hidden file/directory access
denied")

        # B1-SEC-6: Path length and character validation
        if len(str(resolved_path)) > 4096:  # Reasonable path length limit
            raise SecurityViolationError("Path length exceeds security
limit")

        # Check for null bytes and other dangerous characters
        dangerous_chars = ['\x00', '\n', '\r', '\t']
        if any(char in str(original_path) for char in dangerous_chars):
            raise SecurityViolationError("Dangerous characters in file path")

    except (OSError, FileNotFoundError) as e:
        raise SecurityViolationError(f"Invalid or inaccessible file path:
{e}")
```

2. Command Injection Prevention

Attack Scenario: Malicious file paths with shell metacharacters like `; rm -rf /` injected into subprocess calls.

Secure Implementation:

```
def encrypt_file_secure(self, input_path: Path, output_path: Path, password:
str) -> None:
    """Secure Office encryption using direct library calls (no subprocess)"""

    # B2-SEC-1: Legacy format validation
    file_extension = input_path.suffix.lower()
    if file_extension in ['.doc', '.xls', '.ppt']:
        raise FileFormatError(f"Legacy Office format {file_extension} not
supported")

    # B2-SEC-2: Path validation before processing
    validate_path_security_hardened(input_path)
    validate_path_security_hardened(output_path.parent)

    # B2-SEC-3: Password sanitization
    if len(password) > 1024:  # Reasonable password length limit
        raise ValueError("Password exceeds maximum length")
    if '\x00' in password:
        raise ValueError("Null byte in password")

    # B2-SEC-4: Use direct library calls instead of subprocess
    try:
        import msoffcrypto

        # Use msoffcrypto library directly for encryption (when available)
        # This eliminates subprocess command injection risks
        with open(input_path, 'rb') as input_file:
            # Note: Direct encryption may require different approach
            # Fall back to subprocess with strict argument validation only if
necessary
            if self._direct_encryption_available():
                self._encrypt_direct(input_file, output_path, password)
            else:
                self._encrypt_subprocess_secure(input_path, output_path,
password)

    except Exception as e:
        raise ProcessingError(f"Secure Office encryption failed: {e}")

def _encrypt_subprocess_secure(self, input_path: Path, output_path: Path,
password: str) -> None:
    """Fallback secure subprocess implementation with strict validation"""

    # B2-SEC-5: Strict argument validation for subprocess
    import subprocess
    import shlex

    # Validate all paths are within allowed directories
    validate_path_security_hardened(input_path)
    validate_path_security_hardened(output_path.parent)
    # Use argument list (not shell) to prevent injection
    cmd_args = [
        'python', '-m', 'msoffcrypto.cli',
        '-e', '-p', password,
        str(input_path.resolve()),  # Use absolute paths
        str(output_path.resolve())
    ]

    # B2-SEC-6: Secure subprocess execution
    try:
        result = subprocess.run(
            cmd_args,
            capture_output=True,
            text=True,
            timeout=60,
            shell=False,  # CRITICAL: Never use shell=True
            cwd=None,     # Don't inherit current directory
            env={'PATH': os.environ.get('PATH', '')},  # Minimal environment
            check=False
        )

        if result.returncode != 0:
            # Sanitize error output to prevent information disclosure
            sanitized_error = self._sanitize_error_message(result.stderr)
            raise ProcessingError(f"Office encryption failed:
{sanitized_error}")

    except subprocess.TimeoutExpired:
        raise ProcessingError("Office encryption timed out")
```

3. Password Security & Memory Protection

Attack Scenarios: Password exposure in process lists, memory dumps, swap files, or shell history.

Secure Implementation:

```
class SecurePasswordManager:
    """Enhanced password manager with memory protection"""

    def __init__(self):
        self.cli_passwords = []
        self.password_list_file = None
        self.password_list = []
        self._memory_regions = []  # Track memory for secure clearing

    # B3-SEC-1: Secure password input without command line exposure
    def get_password_secure(self, prompt: str = "Password: ") -> str:
        """Get password without exposing in process list or history"""
        import getpass
        import sys

        if sys.stdin.isatty():
            # Interactive mode - use secure password prompt
            password = getpass.getpass(prompt)
        else:
            # Non-interactive mode - read from stdin (for automation)
            password = sys.stdin.readline().rstrip('\n\r')

        if not password:
            raise ValueError("Empty password not allowed")

        return password

    # B3-SEC-2: Secure memory clearing for passwords
    def clear_password_memory(self, password_str: str) -> None:
        """Attempt to clear password from memory (best effort)"""
        import ctypes

        try:
            # Get string object memory location
            address = id(password_str)
            size = len(password_str)

            # Overwrite memory with zeros (best effort in Python)
            ctypes.memset(address, 0, size)

        except Exception:
            # Memory clearing is best effort - don't fail operation
            pass

    # B3-SEC-3: Secure JSON password parsing with validation
    def parse_stdin_passwords_secure(self, stdin_data: str) -> Dict[str,
str]:
        """Secure JSON password parsing with strict validation"""
        import json

        # Validate JSON size to prevent DoS
        if len(stdin_data) > 1024 * 1024:  # 1MB limit
            raise ValueError("Password JSON exceeds size limit")

        try:
            password_mapping = json.loads(stdin_data)

            if not isinstance(password_mapping, dict):
                raise ValueError("Password input must be JSON object")

            # Validate each entry
            validated_mapping = {}
            for filename, password in password_mapping.items():
                if not isinstance(filename, str) or not isinstance(password,
str):
                    raise ValueError("Password mapping must contain only
strings")

                # Validate filename for security
                file_path = Path(filename)
                validate_path_security_hardened(file_path)

                # Validate password
                if len(password) > 1024:
                    raise ValueError(f"Password for {filename} exceeds length
limit")
                if '\x00' in password:
                    raise ValueError(f"Invalid characters in password for
{filename}")

                validated_mapping[filename] = password

            return validated_mapping

        except json.JSONDecodeError as e:
            raise ValueError(f"Invalid JSON in password input: {e}")
        finally:
            # Clear input data from memory
            self.clear_password_memory(stdin_data)
```

4. Secure Temporary File Operations

Attack Scenarios: Race conditions, symlink attacks, predictable file names, insecure permissions.

Secure Implementation:

```
class SecureTempFileManager:
    """Enhanced temporary file manager with security hardening"""

    def __init__(self):
        self.temp_directories = []
        self.temp_files = []
        self.cleanup_registered = False

    # B4-SEC-1: Secure temporary directory creation
    def create_secure_temp_directory(self) -> Path:
        """Create cryptographically secure temporary directory"""
        import tempfile
        import secrets
        import os
        from datetime import datetime

        # B4-SEC-2: Cryptographically secure random naming
        random_suffix = secrets.token_hex(16)  # 32 character hex string
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        pid = os.getpid()

        # Combine multiple entropy sources
        temp_name = f"fastpass_sec_{timestamp}_{pid}_{random_suffix}"

        # B4-SEC-3: Create with most restrictive permissions
        old_umask = os.umask(0o077)  # Ensure only owner can access
        try:
            temp_dir = Path(tempfile.mkdtemp(prefix=temp_name))

            # Set extremely restrictive permissions
            os.chmod(temp_dir, 0o700)  # Owner read/write/execute only

            # Create subdirectories with same restrictive permissions
            for subdir in ['processing', 'output']:
                subdir_path = temp_dir / subdir
                subdir_path.mkdir(mode=0o700)

            self.temp_directories.append(temp_dir)

            # Register cleanup if not already done
            if not self.cleanup_registered:
                import atexit
                atexit.register(self.emergency_cleanup)
                self.cleanup_registered = True

            return temp_dir

        finally:
            os.umask(old_umask)  # Restore original umask

    # B4-SEC-4: Secure atomic file operations
    def atomic_file_write(self, content: bytes, target_path: Path) -> None:
        """Secure atomic file write to prevent race conditions"""
        import secrets
        import os

        # Create temporary file in same directory as target (for atomic move)
        temp_name = f".tmp_{secrets.token_hex(8)}_{target_path.name}"
        temp_path = target_path.parent / temp_name

        try:
            # Write to temporary file with secure permissions
            with open(temp_path, 'wb') as f:
                os.chmod(temp_path, 0o600)  # Restrictive permissions before
writing
                f.write(content)
                f.flush()
                os.fsync(f.fileno())  # Ensure data is written to disk

            # Atomic move to final location
            temp_path.replace(target_path)

        except Exception:
            # Clean up temporary file on failure
            if temp_path.exists():
                try:
                    temp_path.unlink()
                except Exception:
                    pass
            raise

    # B4-SEC-5: Secure file deletion with overwriting
    def secure_delete_file(self, file_path: Path) -> None:
        """Securely delete file with overwriting (best effort)"""
        import os

        try:
            if file_path.exists() and file_path.is_file():
                file_size = file_path.stat().st_size

                # Overwrite with random data (best effort)
                if file_size > 0 and file_size < 100 * 1024 * 1024:  # Only
for files < 100MB
                    with open(file_path, 'r+b') as f:
                        # Multiple passes with different patterns
                        patterns = [b'\x00', b'\xFF',
secrets.token_bytes(min(file_size, 1024))]
                        for pattern in patterns:
                            f.seek(0)
                            if len(pattern) == 1:
                                f.write(pattern * file_size)
                            else:
                                # Write random pattern
                                remaining = file_size
                                while remaining > 0:
                                    chunk_size = min(remaining, len(pattern))
                                    f.write(pattern[:chunk_size])
                                    remaining -= chunk_size
                            f.flush()
                            os.fsync(f.fileno())

                # Remove file
                file_path.unlink()

        except Exception:
            # Don't fail operation if secure deletion fails
            try:
                file_path.unlink()  # Fallback to normal deletion
            except Exception:
                pass
```

5. File Format Attack Prevention

Attack Scenarios: XXE injection, ZIP bombs, malicious macros, polyglot files.

Secure Implementation:

```
def validate_file_format_secure(file_path: Path) -> str:
    """Enhanced file format validation with attack prevention"""
    import filetype
    import zipfile
    import xml.etree.ElementTree as ET

    # B5-SEC-1: File size validation (prevent DoS)
    file_size = file_path.stat().st_size
    if file_size > FastPassConfig.MAX_FILE_SIZE:
        raise FileFormatError(f"File too large: {file_size} bytes")
    if file_size == 0:
        raise FileFormatError("Empty file not allowed")

    # B5-SEC-2: Magic number validation (primary authority)
    detected_type = filetype.guess(str(file_path))
    file_extension = file_path.suffix.lower()

    # Allowed magic number mappings
    allowed_magic_types = {
        'application/vnd.openxmlformats-
officedocument.wordprocessingml.document': '.docx',
        'application/vnd.openxmlformats-officedocument.spreadsheetml.sheet':
'.xlsx',
        'application/vnd.openxmlformats-
officedocument.presentationml.presentation': '.pptx',
        'application/pdf': '.pdf'
    }

    detected_format = None
    if detected_type and detected_type.mime in allowed_magic_types:
        detected_format = allowed_magic_types[detected_type.mime]

        # B5-SEC-3: Cross-validate magic number vs extension
        if file_extension != detected_format:
            raise FileFormatError(f"File format mismatch: extension
{file_extension} vs detected {detected_format}")

    # B5-SEC-4: Extension-based validation (fallback)
    if not detected_format:
        if file_extension in FastPassConfig.SUPPORTED_FORMATS:
            detected_format = file_extension
        else:
            raise FileFormatError(f"Unsupported file format:
{file_extension}")

    # Security validation moved to separate function (after encryption
detection)
    return detected_format

def validate_format_specific_security(file_path: Path, file_format: str) ->
None:
    """
    B5-SEC: Format-specific security validation (runs after encryption
detection)
    Only validates unencrypted files - encrypted files are validated after
decryption
    """
    # B5-SEC-5: Office document security validation
    if ~~detected_format~~file_format in ['.docx', '.xlsx', '.pptx']:
        validate_office_document_security(file_path)

    # B5-SEC-6: PDF security validation
    elif ~~detected_format~~file_format == '.pdf':
        validate_pdf_document_security(file_path)

    ~~return detected_format~~

def validate_office_document_security(file_path: Path) -> None:
    """Validate Office document against security threats"""
    import zipfile
    import xml.etree.ElementTree as ET

    try:
        # B5-SEC-7: ZIP bomb protection
        with zipfile.ZipFile(file_path, 'r') as zip_file:
            total_uncompressed = 0
            file_count = 0

            for info in zip_file.infolist():
                file_count += 1
                total_uncompressed += info.file_size

                # Prevent ZIP bombs
                if file_count > 1000:  # Reasonable file count limit
                    raise FileFormatError("Office document contains too many
files")

                if total_uncompressed > 100 * 1024 * 1024:  # 100MB
uncompressed limit
                    raise FileFormatError("Office document uncompressed size
too large")
```

```
                # Check compression ratio for ZIP bomb detection
                if info.file_size > 0 and info.compress_size > 0:
                    ratio = info.file_size / info.compress_size
                    if ratio > 100:  # High compression ratio indicates
potential ZIP bomb
                        raise FileFormatError("Suspicious compression ratio
in Office document")

            # B5-SEC-8: XML content validation (prevent XXE)
            xml_files = [name for name in zip_file.namelist() if
name.endswith('.xml')]
            for xml_file in xml_files[:10]:  # Limit XML files checked
                try:
                    with zip_file.open(xml_file) as f:
                        xml_content = f.read(1024 * 1024)  # Limit XML size
read
                        validate_xml_security(xml_content)
                except Exception:
                    # Don't fail on XML parsing errors, but log suspicion
                    continue

    except zipfile.BadZipFile:
        raise FileFormatError("Corrupted Office document")

def validate_xml_security(xml_content: bytes) -> None:
    """Validate XML content for security threats"""

    # B5-SEC-9: XXE prevention - check for entity declarations
    xml_str = xml_content.decode('utf-8', errors='ignore')

    # Look for suspicious patterns
    suspicious_patterns = [
        '<!ENTITY',     # Entity declarations
        'SYSTEM',       # System entity references
        'file://',      # File protocol
        'http://',      # HTTP requests in XML
        'https://',     # HTTPS requests in XML
        '&lt;!ENTITY', # HTML-encoded entity declarations
    ]

    xml_lower = xml_str.lower()
    for pattern in suspicious_patterns:
        if pattern.lower() in xml_lower:
            raise FileFormatError("Potentially malicious XML content
detected")

    # Additional size check
    if len(xml_content) > 10 * 1024 * 1024:  # 10MB XML limit
        raise FileFormatError("XML content too large")
```

```
def validate_pdf_document_security(file_path: Path) -> None:
    """Validate PDF document against security threats"""

    # B5-SEC-10: Basic PDF structure validation
    with open(file_path, 'rb') as f:
        header = f.read(1024)

        # Check PDF header
        if not header.startswith(b'%PDF-'):
            raise FileFormatError("Invalid PDF header")

        # Look for suspicious content
        suspicious_content = [
            b'/JavaScript',  # JavaScript in PDF
            b'/JS',         # JavaScript abbreviation
            b'/Launch',     # Launch actions
            b'/GoToR',      # Go to remote actions
        ]

        content_sample = f.read(10 * 1024)  # Read first 10KB for scanning
        for suspicious in suspicious_content:
            if suspicious in content_sample:
                raise FileFormatError("Potentially malicious PDF content
detected")
```

Section B: Security & File Validation

SECURITY CRITICAL: Every security check must map to specific code with proper error handling and sanitization. Label each implementation block with the exact ID shown.

```
# B1: FILE PATH RESOLUTION AND SECURITY VALIDATION
def perform_security_and_file_validation(args: argparse.Namespace) ->
List[FileManifest]:
    import os
    import filetype
    from pathlib import Path
    from typing import List, Dict, Any

    validated_files: List[FileManifest] = []

    # B1a: Collect all files to process
    files_to_process = []
    if args.files:
        files_to_process = args.files
    elif args.recursive:
        files_to_process = collect_files_recursively(args.recursive)
    for file_path in files_to_process:
        # B1b: Path resolution and normalization
        resolved_path = Path(file_path).expanduser().resolve()

        # B1c: Security validation - hardened path traversal protection
        validate_path_security_hardened(resolved_path,
explicit_allow_cwd=args.allow_cwd)

        # B1d: File existence and access validation
        validate_file_access(resolved_path)

        # B1e: File format validation ~~with~~(without security
~~hardening~~validation for encrypted files)
        file_format = validate_file_format_secure(resolved_path)

        # B1f: Encryption status detection
        encryption_status = detect_encryption_status(resolved_path,
file_format)

        # B1g: Security validation (only for unencrypted files)
        # Note: Encrypted files will be validated after decryption
        if not encryption_status:
            validate_format_specific_security(resolved_path, file_format)

        # B1g: Build file manifest entry
        manifest_entry = FileManifest(
            path=resolved_path,
            format=file_format,
            size=resolved_path.stat().st_size,
            is_encrypted=encryption_status,
            crypto_tool=FastPassConfig.SUPPORTED_FORMATS[file_format.suffix]
        )

        validated_files.append(manifest_entry)

    if not validated_files:
        raise FileFormatError("No valid files found to process")

    return validated_files

def validate_path_security(file_path: Path) -> None:
    """B2: Path traversal and security validation"""
    import os
    from pathlib import Path

    # B2a: Resolve absolute path and check for dangerous patterns
    try:
        # Get the absolute path of the intended base directories
        user_home = Path.home().resolve()
        current_dir = Path.cwd().resolve()
        allowed_dirs = [user_home, current_dir]

        # Get the absolute path of the user-provided file path
        resolved_path = file_path.resolve()

        # B2b: Check if the resolved path is within allowed directories
        is_allowed = False
        for base_dir in allowed_dirs:
            try:
                # Check if the resolved path is within the base directory
                resolved_path.relative_to(base_dir)
                is_allowed = True
                break
            except ValueError:
                # Path is not relative to this base directory, try next
                continue

        if not is_allowed:
            raise SecurityViolationError("File access outside allowed
directories")

        # B2c: Additional component analysis for dangerous patterns
        for component in file_path.parts:
            if component in ['..', '.', ''] or component.startswith('.'):
                raise SecurityViolationError("Path traversal attempt
detected")

    except (OSError, ValueError) as e:
        raise SecurityViolationError("Invalid file path")

def validate_file_access(file_path: Path) -> None:
    """B3: File access and permission validation"""
    # B3a: Existence check
    if not file_path.exists():
        raise FileNotFoundError(f"File not found: {file_path}")

    # B3b: Read permission check
    if not os.access(file_path, os.R_OK):
        raise PermissionError(f"No read permission: {file_path}")

    # B3c: Size limit check
    file_size = file_path.stat().st_size
    if file_size > FastPassConfig.MAX_FILE_SIZE:
        raise FileFormatError(f"File too large: {file_size} bytes")

    # B3d: Write permission check for in-place operations
    parent_dir = file_path.parent
    if not os.access(parent_dir, os.W_OK):
        raise PermissionError(f"No write permission in directory:
{parent_dir}")
```

```python
def validate_file_format(file_path: Path) -> str:
    """B4: File format validation using magic number detection first"""
    import filetype

    # B4a: Primary validation - magic number detection
    detected_type = filetype.guess(str(file_path))
    file_extension = file_path.suffix.lower()

    # B4b: Magic number to format mapping (primary authority)
    magic_to_format = {
        'application/vnd.openxmlformats-
officedocument.wordprocessingml.document': '.docx',
        'application/vnd.openxmlformats-officedocument.spreadsheetml.sheet':
'.xlsx',
        'application/vnd.openxmlformats-
officedocument.presentationml.presentation': '.pptx',
        'application/pdf': '.pdf',
    }

    if detected_type and detected_type.mime in magic_to_format:
        # Magic number detected - use this as authoritative format
        authoritative_format = magic_to_format[detected_type.mime]

        # B4c: Cross-validate with file extension
        if file_extension != authoritative_format:
            # Log warning but trust magic number over extension
            print(f"Warning: Extension mismatch for {file_path.name}:
{file_extension} vs detected {authoritative_format}")

        # Check if detected format is supported
        if authoritative_format not in FastPassConfig.SUPPORTED_FORMATS:
            raise FileFormatError(f"Detected file format not supported:
{authoritative_format}")

        return authoritative_format

    # B4d: Fallback to extension-based validation
    if file_extension in FastPassConfig.SUPPORTED_FORMATS:
        print(f"Warning: Could not detect magic number for {file_path.name},
trusting extension: {file_extension}")
        return file_extension

    # B4e: Neither magic number nor extension indicate supported format
    raise FileFormatError(f"Unsupported or undetectable file format:
{file_extension}")

def detect_encryption_status(file_path: Path, file_format: str) -> bool:
    """B5: Detect if file is password protected"""
    if file_format in ['.docx', '.xlsx', '.pptx']:
        # B5a: Office document encryption detection
        import msoffcrypto
        with open(file_path, 'rb') as f:
            office_file = msoffcrypto.OfficeFile(f)
            return office_file.is_encrypted()

    elif file_format == '.pdf':
        # B5b: PDF encryption detection
        import PyPDF2
        with open(file_path, 'rb') as f:
            pdf_reader = PyPDF2.PdfReader(f)
            return pdf_reader.is_encrypted

    return False

@dataclass
class FileManifest:
    """File manifest entry for processing pipeline"""
    path: Path
    format: str
    size: int
    is_encrypted: bool
    crypto_tool: str
```

What's Actually Happening: - B1: File Path Processing & Normalization - Input processing: `args.files` list or `args.recursive` directory path - Path expansion: `os.path.expanduser('~/Documents/file.docx')` → `/home/user/Documents/file.docx` - Canonical paths: `pathlib.Path.resolve()` resolves symlinks and relative paths - File existence: `os.path.exists(file_path)` for each target file - Build file list: `validated_files = [Path objects with metadata]` - Missing files tracked: `missing_files = []` for error reporting - If any files missing: exit with detailed error message listing all missing files

- **B2: Path Traversal Security Analysis**
    - Absolute path resolution: `file_path.resolve()` to get canonical path with symlinks resolved
    - Base directory validation: Check if resolved path is within `Path.home().resolve()` or `Path.cwd().resolve()`
    - Containment checking: Use `resolved_path.relative_to(base_dir)` to verify path is within allowed boundaries
    - Component analysis: Reject paths containing `..`, `.`, hidden files, or empty components
    - System paths: Automatic rejection of paths outside configured allowed directories (default: user home and current working directory)
    - Error handling: Convert OSError/ValueError to SecurityViolationError with sanitized messages
    - Critical exit: if security violations detected, `sys.exit(3)` with generic "security violation" message

- **B3: File Format Magic Number Validation (Primary Authority)**
  - **Priority 1**: Magic number detection via `filetype.guess(file_path)` - authoritative format detection
  - **Priority 2**: File extension validation as fallback when magic number undetectable
  - Magic number mapping (trusted authority):

    ```
    magic_to_format = {
        'application/vnd.openxmlformats-officedocument.wordprocessingml.document': '.docx',
        'application/vnd.openxmlformats-officedocument.spreadsheetml.sheet': '.xlsx',
        'application/vnd.openxmlformats-officedocument.presentationml.presentation': '.pptx',
        'application/pdf': '.pdf',
        'application/zip': '.zip'
    }
    ```
  - Cross-validation: When magic number and extension disagree, trust magic number but log warning
  - Fallback strategy: If magic number undetectable, validate extension against supported formats
  - Format violations: Unsupported formats (by either method) trigger `FileFormatError`
- **B4: File Access & Permission Verification**
  - Read access test: `open(file_path, 'rb')` with exception handling
  - Sample read: read first 1024 bytes to verify file accessibility and detect corruption
  - Size validation: `os.path.getsize(file_path)` vs `max_file_size = 500MB` limit
  - Empty file check: `file_size == 0` indicates potential corruption or invalid file
  - Output directory access: if `--output-dir` specified, test write access to target directory
  - Permission violations: collected in `access_violations = []`
  - If access violations: `sys.exit(1)` with detailed permission error messages
- **B5: Password Protection Status Detection**
  - **Office Documents**: `msoffcrypto.OfficeFile(file_stream).is_encrypted()` returns boolean
  - **PDF Files**: `PyPDF2.PdfReader(file_stream).is_encrypted` property check
  - Store status: `password_status = {'file_path': bool}` for each file

    - **Special case**: If operation is 'encrypt' and file already encrypted, add to warnings
    - **Special case**: If operation is 'decrypt' and file not encrypted, add to warnings
- **B6: Validated File Manifest Creation**
    - Build manifest: `file_manifest = []` containing complete file metadata
    - Manifest entry structure:

      ```
      manifest_entry = {
          'path': Path,
          'extension': str,
          'format': str,
          'size': int,
          'is_password_protected': bool,
          'crypto_tool': str,  # 'msoffcrypto', 'pypdf2'
          'temp_file_needed': bool
      }
      ```
    - Tool assignment: map file extension to appropriate crypto tool
    - Summary calculation: `total_files = len(file_manifest)`, `protected_files = count(is_password_protected)`
    - If critical errors: `sys.exit(3)` with validation summary
    - Success state: `validation_complete = True`, ready for crypto tool setup

Section C: Crypto Tool Selection & Configuration

TOOL INTEGRATION CRITICAL: Each crypto tool handler must be implemented exactly as diagrammed. Label each handler class and method with corresponding IDs.

```
# C1: CRYPTO TOOL HANDLER SETUP
def setup_crypto_tools_and_configuration(validated_files: List[FileManifest])
-> Dict[str, Any]:
    """Initialize and configure crypto tool handlers based on file types"""

    # C1a: Determine required tools
    required_tools = set(manifest.crypto_tool for manifest in
validated_files)

    crypto_handlers = {}

    # C1b: Initialize Office document handler
    if 'msoffcrypto' in required_tools:
        crypto_handlers['msoffcrypto'] = OfficeDocumentHandler()
```

```
    # C1c: Initialize PDF handler
    if 'PyPDF2' in required_tools:
        crypto_handlers['PyPDF2'] = PDFHandler()

    return crypto_handlers

class OfficeDocumentHandler:
    """Handler for Office document encryption/decryption using msoffcrypto"""

    def __init__(self):
        import msoffcrypto
        self.msoffcrypto = msoffcrypto

    def encrypt_file(self, input_path: Path, output_path: Path, password:
str) -> None:
        """C2a: Secure Office document encryption with hardened security"""
        # Use the secure implementation that includes all security
validations
        encrypt_file_secure(self, input_path, output_path, password)

    def decrypt_file(self, input_path: Path, output_path: Path, password:
str) -> None:
        """C2b: Decrypt Office document"""
        with open(input_path, 'rb') as input_file:
            office_file = self.msoffcrypto.OfficeFile(input_file)
            office_file.load_key(password=password)

            with open(output_path, 'wb') as output_file:
                office_file.save(output_file)

        # C2b-SEC: Post-decryption security validation
        # Validate the decrypted file for security threats now that it's in
readable format
        validate_office_document_security(output_path)

    def test_password(self, file_path: Path, password: str) -> bool:
        """C2c: Test if password works for Office document"""
        try:
            with open(file_path, 'rb') as f:
                office_file = self.msoffcrypto.OfficeFile(f)
                office_file.load_key(password=password)
                return True
        except Exception:
            return False

class PDFHandler:
    """Handler for PDF encryption/decryption using PyPDF2"""

    def __init__(self):
        import PyPDF2
        self.PyPDF2 = PyPDF2

    def encrypt_file(self, input_path: Path, output_path: Path, password:
str) -> None:
        """C3a: Encrypt PDF document"""
        with open(input_path, 'rb') as input_file:
            pdf_reader = self.PyPDF2.PdfReader(input_file)
            pdf_writer = self.PyPDF2.PdfWriter()

            # Copy all pages
            for page in pdf_reader.pages:
                pdf_writer.add_page(page)

            # Encrypt with password
            pdf_writer.encrypt(password)

            with open(output_path, 'wb') as output_file:
                pdf_writer.write(output_file)

    def decrypt_file(self, input_path: Path, output_path: Path, password:
str) -> None:
        """C3b: Decrypt PDF document"""
        with open(input_path, 'rb') as input_file:
            pdf_reader = self.PyPDF2.PdfReader(input_file)

            if pdf_reader.is_encrypted:
                pdf_reader.decrypt(password)

            pdf_writer = self.PyPDF2.PdfWriter()

            # Copy all pages
            for page in pdf_reader.pages:
                pdf_writer.add_page(page)

            with open(output_path, 'wb') as output_file:
                pdf_writer.write(output_file)

    def test_password(self, file_path: Path, password: str) -> bool:
        """C3c: Test if password works for PDF"""
        try:
            with open(file_path, 'rb') as f:
                pdf_reader = self.PyPDF2.PdfReader(f)
                if pdf_reader.is_encrypted:
                    return pdf_reader.decrypt(password) == 1
                return True
        except Exception:
            return False

# C4: PASSWORD MANAGEMENT SYSTEM
class PasswordManager:
    """Manages password priority system and validation"""

    def __init__(self, cli_passwords: List[str], password_list_file:
Optional[Path]):
        self.cli_passwords = cli_passwords or []
        self.password_list_file = password_list_file
        self.password_list: List[str] = []

        # C4a: Load password list from file
        if password_list_file:
            self.load_password_list()

    def load_password_list(self) -> None:
        """C4b: Load passwords from file"""
        try:
            with open(self.password_list_file, 'r', encoding='utf-8') as f:
                self.password_list = [line.strip() for line in f if
line.strip()]
        except FileNotFoundError:
            raise FileNotFoundError(f"Password list file not found:
{self.password_list_file}")

    def get_password_candidates(self, file_path: Path) -> List[str]:
        """C4c: Get password candidates in priority order"""
        candidates = []

        # Priority 1: CLI passwords
        candidates.extend(self.cli_passwords)

        # Priority 2: Password list file
        candidates.extend(self.password_list)

        # Remove duplicates while preserving order
        seen = set()
        unique_candidates = []
        for pwd in candidates:
            if pwd not in seen:
                seen.add(pwd)
                unique_candidates.append(pwd)

        return unique_candidates

    def find_working_password(self, file_path: Path, crypto_handler: Any) ->
Optional[str]:
        """C4d: Find working password for file"""
        candidates = self.get_password_candidates(file_path)

        for password in candidates:
            if crypto_handler.test_password(file_path, password):
                return password
        return None
```

What's Actually Happening: - **C1: File Format Analysis & Tool Mapping** - Process validated file manifest: `for file_entry in self.file_manifest:` - Extension-to-tool mapping: `python tool_mapping = { '.docx': 'msoffcrypto', '.xlsx': 'msoffcrypto', '.pptx': 'msoffcrypto', '.doc': 'msoffcrypto', '.xls': 'msoffcrypto', '.ppt': 'msoffcrypto', '.pdf': 'PyPDF2' }` - Assign crypto tool: `file_entry['crypto_tool'] = tool_mapping[file_entry['extension']]` - Group by tool: `self.tool_groups = {'msoffcrypto': [], 'PyPDF2': []}` - Availability check: ensure required tools are available for file types present - If tool missing: `sys.exit(1)` with "Required crypto tool not available: {tool_name}"

- **C2: Crypto Tool Handler Initialization**
  - **msoffcrypto Handler**:

    ```
    class OfficeHandler:
        def __init__(self):
            self.tool_path = 'python -m msoffcrypto.cli'
            self.temp_files = []

        def encrypt(self, input_path, output_path, password):
            # Implementation using msoffcrypto

        def decrypt(self, input_path, output_path, password):
            # Implementation using msoffcrypto
    ```

  - **PyPDF2 Handler**:

    ```
    class PDFHandler:
        def __init__(self):
            self.pdf_library = 'PyPDF2'

        def encrypt(self, input_path, output_path, password):
            # Implementation using PyPDF2 library
    ```

- **C4: msoffcrypto-tool Configuration**
  - Test tool availability: `subprocess.run(['python', '-m', 'msoffcrypto.cli', '--version'])`
  - Configure encryption options:

    ```
    office_config = {
        'password_method': 'standard',  # Use standard Office
    encryption
        'temp_dir': self.temp_working_dir,
        'preserve_metadata': True
    }
    ```

- Set handler methods: `self.office_handler.set_config(office_config)`
- Store in pipeline: `self.crypto_handlers['msoffcrypto'] = office_handler`

- **C5: PyPDF2 Configuration**
  - Initialize PDF library:

    ```
    import PyPDF2
    self.pdf_library = 'PyPDF2'
    # Verify version compatibility for encryption features
    if hasattr(PyPDF2, 'PdfWriter'):  # Check for newer API
        self.writer_class = PyPDF2.PdfWriter
    else:
        self.writer_class = PyPDF2.PdfFileWriter  # Legacy API
    ```

  - Configure PDF encryption settings:

    ```
    pdf_config = {
        'encryption_algorithm': 'AES-256',
        'permissions': {'print': True, 'modify': False, 'copy':
    True},
        'user_password': None,  # Will be set per operation
        'owner_password': None  # Same as user password by default
    }
    ```

- **C7: Tool-Specific Option Configuration**
  - **Office Documents**: Set metadata preservation, compatible encryption methods
  - **PDF Files**: Configure user/owner passwords, permission settings
  - Password validation: ensure passwords meet tool-specific requirements
  - Error handling: configure timeout values, retry attempts for each tool
  - Logging: set up per-tool debug logging if enabled
- **C8: Processing Pipeline Creation**
  - Build processing queue: `self.processing_queue = []`
  - For each file, create processing task:

    ```
    task = {
        'file_path': Path,
        'operation': 'encrypt' | 'decrypt',
        'crypto_handler': handler_object,
        'password': str,
        'output_path': Path,
        'temp_files': []
    }
    ```

  - Sort by file size: process smaller files first for faster feedback

- Dependency resolution: if files depend on each other, order appropriately
- Pipeline validation: ensure all tasks have required inputs and handlers
- Ready state: `self.pipeline_ready = True`, `self.total_tasks = len(processing_queue)`

Section D: File Processing & Operations

PROCESSING CRITICAL: Each step must handle errors gracefully with proper cleanup. Map every processing step to exact code implementation.

```
# D1: SECURE TEMPORARY DIRECTORY SETUP
def create_secure_temporary_directory() -> Path:
    """Create secure temporary working directory with proper permissions"""
    import tempfile
    import os
    from datetime import datetime

    # D1a: Generate unique temp directory name
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    pid = os.getpid()
    temp_name = f"{FastPassConfig.TEMP_DIR_PREFIX}{timestamp}_{pid}"

    # D1b: Create temp directory with secure permissions
    temp_dir = Path(tempfile.mkdtemp(prefix=temp_name))
    os.chmod(temp_dir, 0o700)  # Owner read/write/execute only

    # D1c: Create subdirectories
    (temp_dir / 'processing').mkdir()
    (temp_dir / 'output').mkdir()

    return temp_dir

# D1d: ENHANCED TEMPORARY FILE MANAGEMENT WITH CLEANUP TRACKING
class TempFileManager:
    """Centralized temporary file management with guaranteed cleanup"""

    def __init__(self):
        self.temp_directories = []
        self.temp_files = []
        self.cleanup_registered = False

    def create_temp_directory(self) -> Path:
        """Create tracked temporary directory with automatic cleanup
registration"""
        temp_dir = create_secure_temporary_directory()
        self.temp_directories.append(temp_dir)
        if not self.cleanup_registered:
            import atexit
            atexit.register(self.emergency_cleanup)
            self.cleanup_registered = True

        return temp_dir

    def emergency_cleanup(self):
        """Emergency cleanup for atexit registration"""
        for temp_dir in self.temp_directories:
            try:
                cleanup_temporary_directory(temp_dir)
            except Exception:
                pass  # Silent emergency cleanup

# D1e: CONTEXT MANAGER FOR SECURE TEMPORARY DIRECTORIES
class SecureTempDirectory:
    """Context manager ensuring automatic cleanup even on exceptions"""

    def __init__(self):
        self.temp_dir = None

    def __enter__(self) -> Path:
        self.temp_dir = create_secure_temporary_directory()
        return self.temp_dir

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.temp_dir:
            cleanup_temporary_directory(self.temp_dir)

# D2: FILE PROCESSING PIPELINE
def process_files_with_crypto_operations(
    validated_files: List[FileManifest],
    crypto_handlers: Dict[str, Any],
    args: argparse.Namespace
) -> ProcessingResults:
    """Main file processing pipeline with crypto operations"""

    # D2a: Create secure temporary directory
    temp_dir = create_secure_temporary_directory()

    try:
        processing_results = ProcessingResults()

        for file_manifest in validated_files:
            try:
                # D2b: Process individual file
                result = process_single_file(
                    file_manifest,
                    crypto_handlers[file_manifest.crypto_tool],
                    temp_dir,
                    args
                )
                processing_results.successful_files.append(result)

            except Exception as e:
                error_info = FileProcessingError(
                    file_path=file_manifest.path,
                    error_message=str(e),
                    error_type=type(e).__name__
                )
                # Sanitize error message before storing
                error_info.error_message = 
sanitize_error_message(error_info.error_message)
                processing_results.failed_files.append(error_info)

                # Clean up any partial processing for this file
                cleanup_failed_file_processing(file_manifest.path)

        return processing_results

    finally:
        # D2c: Guaranteed cleanup with error isolation
        try:
            cleanup_temporary_directory(temp_dir)
        except Exception as cleanup_error:
            # Log cleanup failure but don't mask processing results
            print(f"Warning: Cleanup failed for {temp_dir}: {cleanup_error}")

def process_single_file(
    file_manifest: FileManifest,
    crypto_handler: Any,
    temp_dir: Path,
    args: argparse.Namespace
) -> FileProcessingResult:
    """D3: Process a single file through crypto operations"""

    # D3a: Find working password
    password = crypto_handler.password_manager.find_working_password(
        file_manifest.path, crypto_handler
    )

    if not password:
        raise ProcessingError(f"No working password found for 
{file_manifest.path}")

    # D3b: Setup temporary file paths
    temp_input = temp_dir / 'processing' / f"input_{file_manifest.path.name}"
    temp_output = temp_dir / 'output' / f"output_{file_manifest.path.name}"
    # D3c: Copy input to temp location
    shutil.copy2(file_manifest.path, temp_input)

    # D3d: Perform crypto operation
    if args.operation == 'encrypt':
        crypto_handler.encrypt_file(temp_input, temp_output, password)
    else:  # decrypt
        crypto_handler.decrypt_file(temp_input, temp_output, password)

    # D3e: Validate output file
    validate_processed_file(temp_output, args.operation, crypto_handler)

    # D3f: Atomic move to final destination with error handling
    final_path = determine_output_path(file_manifest.path, args.output_dir)

    try:
        # Ensure target directory exists
        final_path.parent.mkdir(parents=True, exist_ok=True)

        # Atomic move to final destination
        shutil.move(temp_output, final_path)
    except Exception as e:
        # Clean up temp output file if move fails
        if temp_output.exists():
            temp_output.unlink()
        raise ProcessingError(f"Failed to move processed file to destination:
{e}")

    return FileProcessingResult(
        original_path=file_manifest.path,
        final_path=final_path,
        operation=args.operation,
        password_used=password,
        file_size_before=file_manifest.size,
        file_size_after=final_path.stat().st_size
    )

def validate_processed_file(output_path: Path, operation: str,
crypto_handler: Any) -> None:
    """D4: Validate that processed file is correct"""

    # D4a: Check file exists and has reasonable size
    if not output_path.exists():
        raise ProcessingError("Output file was not created")

    if output_path.stat().st_size == 0:
        raise ProcessingError("Output file is empty")

    # D4b: Format-specific validation
    file_format = output_path.suffix.lower()
```

```python
    if file_format in ['.docx', '.xlsx', '.pptx']:
        validate_office_document(output_path, operation)
    elif file_format == '.pdf':
        validate_pdf_document(output_path, operation)

def validate_office_document(file_path: Path, operation: str) -> None:
    """D4c: Validate Office document integrity"""
    import msoffcrypto

    try:
        with open(file_path, 'rb') as f:
            office_file = msoffcrypto.OfficeFile(f)

            if operation == 'encrypt':
                # After encryption, file should be encrypted
                if not office_file.is_encrypted():
                    raise ProcessingError("File was not properly encrypted")
            else:  # decrypt
                # After decryption, file should not be encrypted
                if office_file.is_encrypted():
                    raise ProcessingError("File was not properly decrypted")
    except Exception as e:
        raise ProcessingError(f"Office document validation failed: {e}")

def validate_pdf_document(file_path: Path, operation: str) -> None:
    """D4d: Validate PDF document integrity"""
    import PyPDF2

    try:
        with open(file_path, 'rb') as f:
            pdf_reader = PyPDF2.PdfReader(f)

            if operation == 'encrypt':
                # After encryption, PDF should be encrypted
                if not pdf_reader.is_encrypted:
                    raise ProcessingError("PDF was not properly encrypted")
            else:  # decrypt
                # After decryption, PDF should not be encrypted
                if pdf_reader.is_encrypted:
                    raise ProcessingError("PDF was not properly decrypted")

            # Test that we can read at least one page
            if len(pdf_reader.pages) == 0:
                raise ProcessingError("PDF has no readable pages")

    except Exception as e:
        raise ProcessingError(f"PDF validation failed: {e}")

@dataclass
```

```
class ProcessingResults:
    successful_files: List[FileProcessingResult] =
field(default_factory=list)
    failed_files: List[FileProcessingError] = field(default_factory=list)

@dataclass
class FileProcessingResult:
    original_path: Path
    final_path: Path
    operation: str
    password_used: str
    file_size_before: int
    file_size_after: int

@dataclass
class FileProcessingError:
    file_path: Path
    error_message: str
    error_type: str

def cleanup_failed_file_processing(file_path: Path) -> None:
    """Clean up processing artifacts for a failed file"""
    import tempfile
    import shutil

    try:
        # Remove any temporary files associated with this file
        temp_patterns = [
            f"*{file_path.stem}*",
            f"temp_{file_path.name}*",
            f"processing_{file_path.name}*"
        ]

        # Clean up from common temp locations
        temp_dirs = [Path.cwd() / 'temp', Path('/tmp'),
Path(tempfile.gettempdir())]

        for temp_dir in temp_dirs:
            if temp_dir.exists():
                for pattern in temp_patterns:
                    for temp_file in temp_dir.glob(pattern):
                        try:
                            if temp_file.is_file():
                                temp_file.unlink()
                            elif temp_file.is_dir():
                                shutil.rmtree(temp_file)
                        except Exception:
                            # Continue cleanup even if some files can't be
removed
                            pass
    except Exception:
        # Don't let cleanup errors propagate
        pass
```

What's Actually Happening: - **D1: Secure Temporary Directory Setup** - Generate unique temp directory: `temp_name = f'FastPass_{datetime.now():%Y%m%d_%H%M%S}_{os.getpid()}'` - Create with secure permissions: `tempfile.mkdtemp(prefix=temp_name)` then `os.chmod(temp_dir, 0o700)` - Directory structure: `temp_dir/processing/` for input files, `temp_dir/output/` for processed files - Cleanup tracking: `self.temp_directories_created = [temp_dir]` for later cleanup

- **D2: Processing Pipeline Execution**
  - Queue processing: `for task in self.processing_queue:`
  - File isolation: copy each file to `temp_dir/processing/` before processing
  - Tool routing: select appropriate crypto handler based on file format
  - Password application: use `password_manager.find_working_password()` for each file
  - Operation dispatch: call `handler.encrypt()` or `handler.decrypt()` based on mode
  - Output validation: verify processed file integrity and correct encryption status
  - Error handling: collect failures in `failed_files = []`, continue processing remaining files
- **D3: Individual File Processing**
  - **Input preparation**: Copy file to temp location with `shutil.copy2(original, temp_input)`
  - **Password validation**: Test password with crypto tool before processing
  - **Processing execution**:
    - For Office files: use msoffcrypto library via subprocess or direct API
    - For PDF files: use PyPDF2 with PdfReader/PdfWriter classes
  - **Output verification**: Confirm processed file has correct encryption status
  - **File movement**: Move from temp location to final destination (in-place or output directory)
- **D4: File Integrity Validation**
  - **Existence check**: Verify output file was created and is non-empty
  - **Format validation**: Ensure file still opens correctly with appropriate tool
  - **Encryption status**: Verify encrypt/decrypt operation achieved expected result:
    - After encryption: file should be password-protected
    - After decryption: file should not require password
  - **Content integrity**: For PDFs, verify at least one page readable; for Office docs, verify document structure intact
  - **Size sanity check**: File size should be reasonable (not 0 bytes, not dramatically different unless expected)

- **D5: Enhanced Temporary File Management**
    - **Cleanup tracking**: `TempFileManager` class tracks all temporary files and directories
    - **Emergency cleanup**: `atexit.register()` ensures cleanup even on unexpected termination
    - **Context managers**: `SecureTempDirectory` provides automatic cleanup with `try/finally`
    - **Retry logic**: Multiple cleanup attempts with exponential backoff for permission issues
    - **Secure deletion**: Overwrite sensitive temporary files with zeros before deletion
    - **Error isolation**: Cleanup failures don't mask original processing errors
- **D6: Error Handling & Recovery**
    - **Per-file errors**: Collect in `processing_errors = []` with details, continue processing other files
    - **Critical errors**: Stop processing, restore all backups, cleanup temp files
    - **Password errors**: Distinguish between wrong password vs crypto tool failure
    - **File corruption**: Detect if input file becomes corrupted during processing
    - **Partial success**: Some files succeed, some fail - report both with detailed status

Section E: Cleanup & Results Reporting

CLEANUP CRITICAL: All temporary files, passwords in memory, and system state must be properly cleaned up. Map every cleanup operation to code.

```
# E1: RESULTS SUMMARIZATION AND CLEANUP
def cleanup_and_generate_final_report(processing_results: ProcessingResults)
-> int:
    """Generate final report and determine exit code"""

    # E1a: Calculate summary statistics
    total_files = len(processing_results.successful_files) +
len(processing_results.failed_files)
    successful_count = len(processing_results.successful_files)
    failed_count = len(processing_results.failed_files)

    # E1b: Generate report
    generate_operation_report(processing_results, total_files,
successful_count, failed_count)

    # E1c: Clear sensitive data from memory
    clear_sensitive_data()

    # E1d: Determine exit code
    if failed_count == 0 and successful_count > 0:
        return 0  # Success
    elif failed_count > 0 and successful_count > 0:
        return 1  # Partial success
    elif failed_count > 0 and successful_count == 0:
        return 1  # Complete failure
    else:
        return 2  # No files processed

def generate_operation_report(
    processing_results: ProcessingResults,
    total_files: int,
    successful_count: int,
    failed_count: int,
    report_format: str = 'text'
) -> None:
    """E2: Generate comprehensive operation report in specified format"""

    if report_format == 'json':
        generate_json_report(processing_results, total_files,
successful_count, failed_count)
    elif report_format == 'csv':
        generate_csv_report(processing_results, total_files,
successful_count, failed_count)
    else:  # text format (default)
        generate_text_report(processing_results, total_files,
successful_count, failed_count)

def generate_text_report(
    processing_results: ProcessingResults,
    total_files: int,
    successful_count: int,
    failed_count: int
) -> None:
    """Generate human-readable text report"""

    print("\n" + "="*50)
    print("FastPass Operation Complete")
    print("="*50)

    # E2a: Summary statistics
    print(f"Total files processed: {total_files}")
    print(f"Successful: {successful_count}")
    print(f"Failed: {failed_count}")

    # E2b: List successful files
    if processing_results.successful_files:
        print(f"\n✓ Successful files:")
        for result in processing_results.successful_files:
            size_change = result.file_size_after - result.file_size_before
            size_indicator = f"({size_change:+d} bytes)" if size_change != 0
else ""
            print(f"  • {result.original_path.name} →
{result.final_path.name} {size_indicator}")

    # E2c: List failed files
    if processing_results.failed_files:
        print(f"\n✗ Failed files:")
        for error in processing_results.failed_files:
            print(f"  • {error.file_path.name}: {error.error_message}")

    # E2d: Next steps
    if failed_count > 0:
        print(f"\nTroubleshooting:")
        print("- Verify passwords are correct")
        print("- Check file permissions")
        print("- Ensure files are not corrupted")

def generate_json_report(
    processing_results: ProcessingResults,
    total_files: int,
    successful_count: int,
    failed_count: int
) -> None:
    """Generate machine-readable JSON report"""
    import json
    from datetime import datetime

    report = {
        "timestamp": datetime.now().isoformat(),
        "summary": {
            "total_files": total_files,
            "successful": successful_count,
            "failed": failed_count,
            "success_rate": successful_count / total_files if total_files > 0
else 0
        },
        "successful_files": [
            {
                "original_path": str(result.original_path),
                "final_path": str(result.final_path),
                "operation": result.operation,
                "file_size_before": result.file_size_before,
                "file_size_after": result.file_size_after,
                "size_change": result.file_size_after -
result.file_size_before
            }
            for result in processing_results.successful_files
        ],
        "failed_files": [
            {
                "file_path": str(error.file_path),
                "error_message": error.error_message,
                "error_type": error.error_type
            }
            for error in processing_results.failed_files
        ]
    }

    print(json.dumps(report, indent=2))

def generate_csv_report(
    processing_results: ProcessingResults,
    total_files: int,
    successful_count: int,
    failed_count: int
) -> None:
    """Generate CSV format report"""
    import csv
    import sys

    writer = csv.writer(sys.stdout)

    # Write header
    writer.writerow(['file_path', 'status', 'operation', 'size_before',
'size_after', 'error_message'])

    # Write successful files
    for result in processing_results.successful_files:
        writer.writerow([
            str(result.original_path),
            'success',
            result.operation,
            result.file_size_before,
            result.file_size_after,
            ''
        ])

    # Write failed files
    for error in processing_results.failed_files:
        writer.writerow([
            str(error.file_path),
            'failed',
            '',
            '',
            '',
            error.error_message
        ])

def clear_sensitive_data() -> None:
    """E3: Clear passwords and sensitive data from memory"""
    import gc

    # E3a: This would be implemented to overwrite password variables
    # In practice, Python doesn't provide direct memory overwriting
    # but we can delete variables and force garbage collection

    # Clear any global password variables
    globals_to_clear = [k for k in globals().keys() if 'password' in
k.lower()]
    for var_name in globals_to_clear:
        if var_name in globals():
            del globals()[var_name]

    # Force garbage collection
    gc.collect()

def cleanup_temporary_directory(temp_dir: Path) -> None:
    """E4: Secure cleanup with retry logic and secure file deletion"""
    import shutil
    import time
    import os

    if not temp_dir.exists():
        return

    # E4a: Multiple cleanup attempts with exponential backoff
    max_attempts = 3
    for attempt in range(max_attempts):
        try:
            # E4b: Secure deletion of sensitive files (attempt to overwrite)
            for file_path in temp_dir.rglob('*'):
                if file_path.is_file():
                    try:
                        file_size = file_path.stat().st_size
                        # Only attempt secure deletion for reasonably sized
files
                        if 0 < file_size < 10 * 1024 * 1024:  # < 10MB
                            with open(file_path, 'r+b') as f:
                                f.write(b'\x00' * file_size)
                                f.flush()
                                os.fsync(f.fileno())
                    except Exception:
                        # Secure deletion failed, continue with normal
deletion
                        pass

            # E4c: Remove entire directory tree
            shutil.rmtree(temp_dir)
            return  # Success - exit retry loop
        except (PermissionError, OSError) as e:
            if attempt < max_attempts - 1:
                # Exponential backoff for retry
                time.sleep(0.1 * (2 ** attempt))
                continue
            else:
                print(f"Warning: Could not clean up temp directory
{temp_dir}: {e}")
                break
```

What's Actually Happening: - **E1: Operation Summary & Statistics Calculation** - Count files: `total_files = len(self.processing_results)` - Success rate: `successful_files = len([r for r in results if r.status == 'success'])` - Failure breakdown: categorize failures by type (password, permission, corruption, tool failure) - Processing time: `total_time = datetime.now() - self.operation_start_time` - Performance stats: files per second, total bytes processed, average file size

- **E2: Comprehensive Results Report Generation**
    - **Header section**: FastPass version, operation mode, timestamp
    - **Summary statistics**: Total files, success count, failure count, processing time
    - **Successful files list**:

      ```
      ✓ Successful files:
        • document1.docx → document1.docx (encrypted, +1,247 bytes)
        • report.pdf → secured/report.pdf (decrypted, -892 bytes)
        • data.xlsx → data.xlsx (encrypted, +2,156 bytes)
      ```

    - **Failed files list**:

      ```
      ✗ Failed files:
        • protected.pdf: Wrong password
        • corrupt.docx: File format error
        • readonly.xlsx: Permission denied
      ```

    - **Troubleshooting section**: If failures occurred, provide specific guidance based on failure types
- **E3: Sensitive Data Memory Cleanup**
    - **Password variables**: Explicitly delete all password variables from memory
    - **Command line args**: Clear args.passwords, args.password_list contents
    - **Processing state**: Clear password_manager internal state
    - **Garbage collection**: Force `gc.collect()` to ensure memory cleanup
    - **Note**: Python doesn't guarantee memory overwriting, but this is best effort cleanup

- **E4: Temporary File & Directory Cleanup**
    - **Temp directory removal**: `shutil.rmtree(temp_dir)` for each temp directory created
    - **Intermediate files**: Clean up any partial processing files left behind
    - **Lock files**: Remove any file locks or temp markers created during processing
    - **Error handling**: Log warnings for cleanup failures but don't fail the operation
- **E5: Final Exit Code Determination**
    - **Exit Code 0**: All files processed successfully, no errors
    - **Exit Code 1**: Some files failed, some succeeded (partial success)
    - **Exit Code 2**: All files failed to process, or no files processed
    - **Exit Code 3**: Security violation detected, operation aborted
    - **Exit Code 4**: Authentication failure (wrong passwords for all files)
- **E6: Operation State Reset**
    - Clear processing queues: `self.processing_queue = []`
    - Reset file manifests: `self.file_manifest = []`
    - Clear handler references: `self.crypto_handlers = {}`
    - Reset application state: `self.ready_for_processing = False`
    - Final log entry: `logger.info(f"FastPass operation completed in {total_time} with {successful_count}/{total_files} files successful")`

Security Implementation Summary

Comprehensive Security Hardening Implemented

FastPass includes enterprise-grade security hardening based on comprehensive threat analysis and attack vector identification. All security measures are mandatory and must be implemented exactly as specified.

Security Mitigations by Attack Vector

Attack Vector	Mitigation Implemented	Security Function
Path Traversal	Hardened path validation with symlink detection	`validate_path_security_hardened()`
Command Injection	Direct library calls + secure subprocess	`encrypt_file_secure()`
Password Exposure	Memory clearing + secure input handling	`SecurePasswordManager`
Race Conditions	Atomic operations + secure temp files	`SecureTempFileManager`
XXE Injection	XML entity detection + content validation	`validate_xml_security()`

Attack Vector	**Mitigation Implemented**	**Security Function**
ZIP Bombs	Compression ratio analysis + size limits	`validate_office_document_security()`
Malicious PDFs	JavaScript detection + content scanning	`validate_pdf_document_security()`
Symlink Attacks	Symlink detection + strict path resolution	`validate_path_security_hardened()`
DoS Attacks	Input size limits + resource constraints	Multiple validation functions
File Format Confusion	Magic number validation + strict matching	`validate_file_format_secure()`

Security Validation Strategy

Key Security Design Decision: Security validation timing is critical for encrypted file processing.

Validation Pipeline Order: 1. **Pre-processing validation** (all files): - Path security validation - File existence and access validation - File format detection (without content validation) - Encryption status detection

1. **Security validation (conditional)**:
   - **Unencrypted files**: Immediate security validation after format detection
   - **Encrypted files**: Security validation AFTER decryption (when content is readable)

Rationale: Encrypted Office documents are not valid ZIP files until decrypted. Security validation that attempts to read ZIP structure will fail on encrypted files. The validation must occur after decryption when the file is in readable format.

Implementation: - `validate_format_specific_security()` - Only runs on unencrypted files during initial processing - `validate_office_document_security()` - Runs on decrypted files in post-decryption handlers - Maintains security protection while enabling encrypted file processing

Security Configuration Options

```
# CLI Security Flags
--allowed-dirs          # Specify custom allowed directories (space-separated)

# Environment Variables
FASTPASS_CUSTOM_ALLOWED_DIRS=""         # Custom allowed directories (comma-separated)
FASTPASS_MAX_PASSWORD_LENGTH=1024       # Password length limit
FASTPASS_MAX_JSON_SIZE=1048576          # 1MB JSON input limit
FASTPASS_ENABLE_SECURE_DELETION=true    # Overwrite files before deletion
FASTPASS_SYMLINK_PROTECTION=true        # Block symlink access
FASTPASS_XML_ENTITY_PROTECTION=true     # XXE protection enabled
```

Security-First Design Principles

- ☐ **Principle of Least Privilege**: By default, allow access to user home directory and current working directory for practical usability
- ☐ **Defense in Depth**: Multiple layers of validation and sanitization
- ☐ **Fail Secure**: Security violations result in immediate termination
- ☐ **Input Validation**: All user inputs validated against strict criteria
- ☐ **Secure by Default**: Most restrictive settings enabled by default
- ☐ **Memory Protection**: Best-effort password clearing and secure handling
- ☐ **Atomic Operations**: Prevent race conditions and partial state corruption
- ☐ **Content Validation**: Deep inspection of file contents for malicious patterns

Critical Security Requirements

MANDATORY IMPLEMENTATION: The following security measures are not optional and must be implemented exactly as specified:

- ☐ ✅ **Path Traversal Protection**: `validate_path_security_hardened()` with symlink detection
- ☐ ✅ **Command Injection Prevention**: Direct library calls or secure subprocess with argument validation
- ☐ ✅ **Password Memory Protection**: `SecurePasswordManager` with memory clearing
- ☐ ✅ **Secure Temporary Files**: Cryptographically secure naming and restrictive permissions (0o600)
- ☐ ✅ **File Format Validation**: Magic number + extension cross-validation with attack detection
- ☐ ✅ **Input Sanitization**: All user inputs validated for length, content, and dangerous patterns
- ☐ ✅ **Error Sanitization**: No sensitive information disclosed in error messages
- ☐ ✅ **Resource Limits**: File size, path length, and processing time constraints

Security Testing Requirements

Each security measure must be tested with specific attack scenarios:

```
# Security Test Cases (Required)
test_path_traversal_attacks()        # ../../../../etc/passwd
test_symlink_attacks()               # Symlink to sensitive files
test_command_injection()             # ; rm -rf / in file paths
test_password_exposure()             # Process list monitoring
test_xxe_injection()                 # XML external entity attacks
test_zip_bomb_detection()            # Compression ratio attacks
test_malicious_pdf_content()         # JavaScript/Launch actions
test_race_condition_prevention()     # Concurrent file operations
test_dos_via_large_inputs()          # Oversized files and inputs
test_file_format_confusion()         # Polyglot and mismatched formats
```

Security Audit Checklist

Before deployment, verify each security control:

- ☐ Path validation blocks `../../../etc/passwd` access attempts
- ☐ Symlink access denied with clear error message

- ☐ Subprocess calls use argument arrays, never shell execution
- ☐ Passwords not visible in `ps aux` output during processing
- ☐ JSON password input validated for size and content
- ☐ Temporary files created with 0o600 permissions
- ☐ File format mismatches rejected (e.g., .pdf with .docx magic number)
- ☐ ZIP bomb detection triggers on high compression ratios
- ☐ PDF JavaScript content blocked
- ☐ XXE entity declarations in Office documents blocked
- Memory clearing attempted for password variables

1. Error messages sanitized of sensitive information

Implementation Status & Next Steps

Current Development Phase

- **Phase**: Architecture specification complete
- **Status**: Ready for implementation
- **Next Priority**: Begin implementation of Section A (CLI Parsing & Initialization)

Implementation Order

- **Section A**: CLI parsing and basic application structure
- **Section B**: Security validation and file format detection

- **Section C**: Crypto tool integration and handler classes
- **Section D**: File processing pipeline with error handling
- **Section E**: Cleanup, reporting, and finalization

Key Implementation Notes

- Each code section must be labeled with exact IDs from this specification (e.g., `# A1a`, `# B3c`)
- All error handling must follow the patterns defined in the pseudocode
- Security validations are mandatory and cannot be simplified or skipped
- Password handling must implement the complete priority system as specified
- File processing must use the secure temporary directory approach

Comprehensive Testing Strategy

Unit Testing Framework

- **Framework**: pytest with coverage reporting (pytest-cov)
- **Test Structure**: Mirror source code structure in tests/ directory
- **Coverage Target**: Minimum 85% code coverage for all modules
- **Mocking**: Use unittest.mock for external dependencies and file system operations

Test Categories

1. Security Testing (Critical Priority)

```python
# tests/test_security.py
class TestPathTraversalSecurity:
    def test_reject_parent_directory_traversal(self):
        """Test rejection of '../' path traversal attempts"""

    def test_reject_absolute_paths_outside_allowed(self):
        """Test rejection of paths outside configured allowed directories"""

    def test_symlink_resolution_security(self):
        """Test proper handling of symbolic links"""

    def test_windows_path_traversal_patterns(self):
        """Test Windows-specific path traversal patterns"""

    def test_url_encoded_path_injection(self):
        """Test rejection of URL-encoded traversal attempts"""

class TestPasswordSecurity:
    def test_password_memory_clearing(self):
        """Test password variables are cleared from memory"""

    def test_password_not_logged(self):
        """Test passwords never appear in log outputs"""

    def test_error_message_sanitization(self):
        """Test sensitive data removed from error messages"""

class TestFileFormatSecurity:
    def test_magic_number_validation(self):
        """Test file format detection via magic numbers"""

    def test_malicious_file_rejection(self):
        """Test rejection of files with mismatched extensions"""

    def test_large_file_rejection(self):
        """Test rejection of files exceeding size limits"""
```

2. Crypto Handler Testing

```python
# tests/test_crypto_handlers.py
class TestOfficeDocumentHandler:
    def test_encrypt_docx_file(self):
        """Test DOCX encryption with valid password"""

    def test_decrypt_protected_xlsx(self):
        """Test XLSX decryption with correct password"""

    def test_wrong_password_handling(self):
        """Test graceful handling of incorrect passwords"""

    def test_corrupted_file_detection(self):
        """Test detection and handling of corrupted Office files"""

class TestPDFHandler:
    def test_pdf_encryption_standard(self):
        """Test PDF encryption with standard security"""

    def test_pdf_decryption_validation(self):
        """Test PDF decryption and integrity validation"""

    def test_pdf_permission_handling(self):
        """Test handling of PDF permission restrictions"""
```

3. Integration Testing

```python
# tests/test_integration.py
class TestEndToEndWorkflows:
    def test_full_encryption_workflow(self):
        """Test complete file encryption from CLI to output"""

    def test_batch_processing_mixed_formats(self):
        """Test processing multiple file types in single operation"""

    def test_recursive_directory_processing(self):
        """Test recursive directory processing with filters"""

    def test_error_recovery_and_cleanup(self):
        """Test system recovery from processing errors"""

class TestPasswordManagement:
    def test_password_priority_system(self):
        """Test CLI > list file priority order"""


    def test_stdin_json_password_input(self):
        """Test JSON password input via stdin"""
```

4. Error Handling Testing

```python
# tests/test_error_handling.py
class TestErrorScenarios:
    def test_missing_file_handling(self):
        """Test graceful handling of missing input files"""

    def test_permission_denied_scenarios(self):
        """Test handling of read/write permission failures"""

    def test_disk_space_exhaustion(self):
        """Test behavior when disk space runs out during processing"""

    def test_crypto_tool_unavailable(self):
        """Test fallback when required crypto tools missing"""

    def test_partial_processing_cleanup(self):
        """Test cleanup of partially processed files on failure"""
```

5. Performance Testing

```python
# tests/test_performance.py
class TestPerformanceBenchmarks:
    def test_large_file_processing_time(self):
        """Test processing time for files up to size limit"""

    def test_batch_processing_scalability(self):
        """Test performance with increasing number of files"""

    def test_memory_usage_monitoring(self):
        """Test memory usage stays within reasonable bounds"""
```

Test Data Management

- **Test Fixtures**: Create representative Office/PDF files for testing
- **Encrypted Samples**: Pre-encrypted files with known passwords
- **Malicious Samples**: Files designed to test security vulnerabilities
- **Large Files**: Test files of various sizes up to the limit
- **Corrupted Files**: Intentionally corrupted files for error testing

Continuous Integration

GitHub Actions: Run tests on multiple Python versions (3.8+)
Platform Testing: Test on Windows, macOS, and Linux
Security Scanning: Integrate SAST tools (bandit, safety)
Coverage Reporting: Automated coverage reports and enforcement

Manual Testing Checklist

User Acceptance Testing: Manual testing of CLI workflows
Cross-Platform Testing: Verify behavior across operating systems
Edge Cases: Manual testing of unusual file combinations
Documentation Validation: Ensure examples in docs actually work

Test Execution Commands

```
# Run all tests with coverage
pytest tests/ --cov=src --cov-report=html --cov-report=term

# Run only security tests
pytest tests/test_security.py -v

# Run performance benchmarks
pytest tests/test_performance.py --benchmark-only

# Run integration tests
pytest tests/test_integration.py -v
```

Implementation Quality Gates

Phase 1 - Security Foundation (Must Pass) - All security tests pass (path traversal, file validation, password handling) - Error message sanitization verified - Temporary file cleanup confirmed

Phase 2 - Core Functionality (Must Pass)
- All crypto handler tests pass - File processing pipeline tests pass - Configuration management tests pass

Phase 3 - Integration & Performance (Must Pass) - End-to-end workflow tests pass - Performance benchmarks meet targets - Error recovery tests pass

Phase 4 - Production Readiness (Must Pass) - 85%+ code coverage achieved - All manual test scenarios pass - Documentation examples verified

This specification serves as the complete blueprint for FastPass implementation. All code must conform to this architecture, implement the exact pseudocode patterns shown above, and pass the comprehensive testing strategy before deployment.