Pushan: Trace-Free Deobfuscation of Virtualization-Obfuscated Binaries

Ashwin Sudhir, Zion Leonahenahe Basque, Wil Gibbs, Ati Priya Bajaj,
Pulkit Singh Singaria, Mitchell Zakocs, Jie Hu, Moritz Schloegel*,
Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, Ruoyu Wang
Arizona State University
* CISPA Helmholtz Center for Information Security
{asudhir1,zbasque,wfgibbs,atipriya,psingari,mzakocs,jiehu12,tbao,doupe,yans,fishw}@asu.edu
moritz.schloegel@cispa.de

***Abstract*—In the ever-evolving battle against malware, binary obfuscation techniques are a formidable barrier to the effective analysis of malware by both human security analysts and automated systems. In particular, virtualization or VM-based obfuscation is one of the strongest protection mechanisms that evade automated analysis. Despite widespread use of virtualization in practice, existing automated deobfuscation techniques suffer from three major drawbacks. First, they only work on execution traces, which prevents them from recovering *all* logic in an obfuscated binary. Second, they depend on dynamic symbolic execution, which is expensive and does not scale in practice. Third, they cannot generate "well-formed" code, which prevents existing binary decompilers from generating human-friendly decompilation output.**

This paper introduces PUSHAN, a novel and generic technique for deobfuscating virtualization-obfuscated binaries while overcoming the limitations of existing techniques. PUSHAN is trace-free and avoids path-constraint accumulation by using VPC-sensitive, constraint-free symbolic emulation to recover a complete CFG of the virtualized function. It is the first approach that also decompiles the protected code into high-quality C pseudocode to enable effective analysis. Crucially, PUSHAN circumvents the traditional reliance on path satisfiability, which is a known NP-hard problem that hampers the scalability of existing deobfuscation techniques. We evaluate PUSHAN on a diverse set of more than $1,000$ binaries, including targets protected by academic state of the art, Tigress, and by the commercial-strength obfuscators VMProtect and Themida. PUSHAN successfully deobfuscates these binaries, retrieves their *complete* CFGs, and decompiles them to C pseudocode. We further demonstrate practical applicability by analyzing a previously unanalyzed VMProtect-obfuscated malware sample from VirusTotal, where our decompiled output enables LLM-assisted code simplification, reuse and program understanding, making our approach the first to enable effective end-to-end analysis of code protected by virtualization.

I. INTRODUCTION

Over 50 years have passed since the birth of the first computer virus in 1971, and malware is still a critical and widespread threat to Internet users. Security companies report that they detected more than 500,000 unique malware samples per day in 2025, which is a 7% increase over 2024 [1]. The constant flow of newly discovered malware has driven the recruitment of human security analysts and the development of additional intelligent automated techniques for malware analysis. To thwart such attempts, malware authors turned to protection schemes, with virtualization-based binary obfuscation techniques (also referred to as Virtual Machine (VM)-based obfuscation) being the most notable [2].

A virtualization-based obfuscator protects a binary program from both manual and automated reverse engineering by translating the original instructions into virtual machine (VM) bytecode that uses a custom, usually randomized instruction set. Then, it injects one or more VM interpreters (also called virtual CPUs) into the protected binary, which interpret the newly generated VM bytecode during run time. Essentially, virtualization-based obfuscation hides the original control flow, which is often critical for reverse engineering [3], and instead exposes only the control flow of the VM interpreters. To reverse engineer such a binary, security analysts must first reverse engineer the VM interpreters, which is tedious, time-consuming, and error-prone.

While academic efforts to automate the deobfuscation of virtualization-obfuscated binaries are extensive [4]–[9], three key limitations hinder their practical adoption: (1) *Incomplete Path Coverage*: Existing techniques are primarily trace-based, reasoning about one execution path at a time to recover a control flow graph (CFG). Given the complex logic and multitude of paths in real-world malware, achieving complete path coverage is virtually impossible, potentially leaving analysts with an incomplete CFG and missing critical logic. (2) *Low Scalability*: These techniques rely on dynamic symbolic execution [10] for path exploration, which is computationally expensive and inherently unscalable due to the path explosion problem and the NP-completeness [11] of satisfiability checking. Furthermore, dynamic symbolic exploration is notoriously brittle and easily defeated by common obfuscation countermeasures [12]–[16]. (3) *Low Usability*: Existing techniques output raw instruction traces that are not conducive to analysis. Security analysts rely on familiar tools like binary decompilers

Network and Distributed System Security (NDSS) Symposium 2027
22–26 March 2027, Seoul, Republic of Korea
ISBN 978-1-970672-09-1
https://dx.doi.org/10.14722/ndss.2027.232963
www.ndss-symposium.org

to interpret code [17]–[20], which improve the speed and accuracy of malware analysis [21]. However, these tools expect structured, "well-formed" assembly code and a complete CFG [22] for correct performance. As Section III shows, even industry-standard tools like the Hex-Rays Decompiler fail to produce clean C pseudocode from deobfuscated traces, forcing analysts back into tedious manual reversing.

In this paper, we present PUSHAN, a scalable technique that deobfuscates virtualization-obfuscated binaries in a *trace-free* manner, recovers a *complete* CFG of the deobfuscated program, and generates decompiled C pseudocode as the output. PUSHAN builds on the concept of Virtual Program Counter (VPC) sensitivity [23], which is drawn from the concept of sensitivities in program analysis: We can recover the *original* CFG of a virtualization-obfuscated binary by making CFG recovery sensitive to both the actual program counter (which always points to instructions in an VM interpreter) and the VPC, which always points to a location in the VM bytecode region. To achieve VPC sensitivity, we design a new CFG recovery algorithm based on *constraint-free symbolic emulation*, which symbolically evaluates instructions without accumulating path constraints or performing path exploration. PUSHAN recovers a VPC-sensitive, flattened CFG (termed a *flat CFG*) that contains both the (now flattened) interpreter logic and the logic of the original program. It then iteratively performs semantics-preserving simplifications to simplify the flat CFG. The goal of simplification is to remove any logic related to VM interpreters and only retain the logic of the original, unobfuscated code. Lastly, PUSHAN uses a custom binary decompiler to generate C pseudocode, which makes malware easier to analyze [21]. This advantage largely stems from the correctness of the generated CFG used during decompilation [22], [24].

We evaluate PUSHAN on several datasets to understand its correctness, capabilities, and impact. First, we used 14 programs, including malware samples, open-source software, and synthetic binaries, obfuscated using the widely adopted, commercial obfuscators VMProtect [25] in standard and Ultra configurations and Themida [26], yielding 42 main commercial-obfuscator cases. Our evaluation shows that PUSHAN is not hindered by the path satisfiability problem, can fully recover the CFGs, and achieves 100% similarity to the original CFGs for 28 out of the 42 cases, with high similarity in the remaining samples. Then, we evaluated PUSHAN on 1,000 binaries generated with Tigress using three different VM configurations, finding that PUSHAN fully analyzed and successfully deobfuscated and decompiled 988 of them. We compared this against the state of the art [5], which only recovered 68 complete CFGs. We also evaluated PUSHAN on five CTF challenges obfuscated with bespoke VM implementations. It succeeds in all five samples and for one even revealed the embedded flag value in the decompiled output, highlighting its ability to recover semantically meaningful logic from custom VMs.

Finally, we evaluated PUSHAN on a real-world VMProtect-obfuscated binary sample and recovered the logic of a virtualization-obfuscated function in the binary. This allowed us to determine the true nature of this binary and demonstrates that PUSHAN produces decompiler-consumable control flow that enables practical decompilation of virtualization-obfuscated binaries.

Contributions. This paper makes three key contributions:

- We review state-of-the-art devirtualization approaches and identify a fundamental problem that limits their scalability and usability.
- We propose PUSHAN, a scalable deobfuscation framework based on *VPC-sensitive, constraint-free symbolic emulation* that recovers complete control flow from virtualization-obfuscated binaries.
- We evaluate PUSHAN on binaries protected by commercial-strength obfuscators and show its ability to recover complete CFGs and produce decompiled code semantically consistent with original code.

We release the source code and evaluation artifacts of PUSHAN on GitHub [27]. We also port PUSHAN to the latest version of angr to support future research on binary deobfuscation.

II. VIRTUAL MACHINE DEOBFUSCATION

Many obfuscation techniques exist, including Mixed Boolean-Arithmetic (MBAs) [29] or Opaque Predicates [30], [31]. One of the most powerful techniques is *virtualization-based obfuscation*, also referred to as *VM-based obfuscation*. It runs the to-be-protected code inside a custom interpreter that uses a custom bytecode representation. Figure 1 illustrates a typical interpreter design, featuring a centralized fetch-decode-execute loop that dispatches VM bytecode instructions to corresponding handlers. Different architectures are possible, including threaded code [32], [33], where the address of the next bytecode (direct threading) or the fetch and decode step (indirect threading) are inlined into the VM handlers, avoiding the easy to identify VM dispatcher loop. The Virtual Program Counter (VPC), akin to its analogous counterpart on a CPU, tracks the execution through the bytecode.

Complete CFG. We use a *complete CFG* to mean a recovered CFG that contains every feasible control-flow path in the original, unobfuscated function. Completeness is a coverage property, not graph isomorphism: a complete recovered CFG may contain additional nodes or edges introduced by the obfuscator or retained by conservative recovery, as long as it does not miss executable paths from the original function.

Removing VM-based obfuscation requires (1) identifying the VM, (2) mapping VM handlers, (3) reconstructing handler semantics, and (4) developing a disassembler for the VM. Only after these steps can an analyst begin studying the underlying program logic. To alleviate this burden, various works have proposed automating Steps (2) to (4).

Table I lists the state-of-the-art solutions in VM deobfuscation with the goal of recovering the original code. Studying their underlying primitives, we can identify two philosophies: techniques that rely on tracing and mount an analysis on these traces, and Kinder's VPC-sensitive abstract interpretation.

TABLE I: A qualitative comparison of PUSHAN against existing techniques for virtualization-based deobfuscation.

	Key Techniques	Output	Complex Programs	CFG Complete?	Code Decompilable?
Kinder [23]	abstract interpretation, VPC sensitivity	calls+arguments	?[1]	–	✘
Coogan [28]	tracing, slicing	subtrace	✓	–	✘
Yadegari [4]	tracing, taint analysis	CFG	✓	✘	✘
VMHUNT [6]	tracing, symbolic execution	Trace	✓	–	✘
Salwan [5]	tracing, taint, formula slicing	CFG	✘[2]	✘	✓
PUSHAN	emulation, VPC sensitivity	CFG	✓	✓	✓

[1]: Only one toy example tested [2]: Supports obfuscated code that contains not more than one or two branches

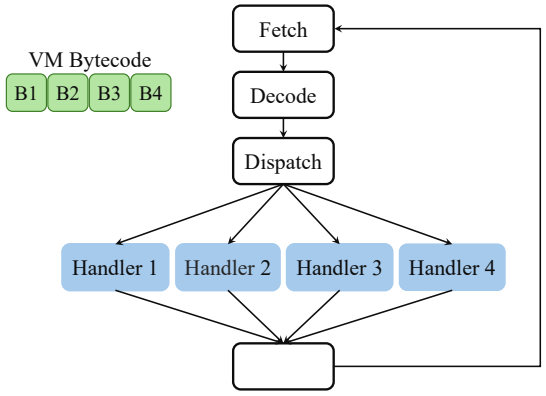


Fig. 1: An illustration of virtualization-based obfuscation, where the to-be-protected code has been translated into **bytecode**. A *fetch-decode-execute* loop with a central dispatcher dispatches execution to individual **VM handler**s. Each bytecode instruction will be handled by a VM handler.

Tracing-based analysis. Trace-based techniques bypass virtualization by observing VM behavior on concrete executions. They (1) capture an execution trace for a given input, and (2) simplify the trace using techniques such as taint analysis to isolate the underlying logic from input-dependent or obfuscation-related noises. Some techniques use program synthesis to find equivalent but simpler expressions for VM handlers [8], [9]. While they differ in implementation details and complexity, this approach is fundamentally limited to one execution path and cannot capture the *complete* program behavior. Optionally, (3) they may derive input cases (e.g., by querying an SMT solver) to generate additional exeuction traces, which is popular among prior work [4]–[6], [28]. However, symbolic execution is particularly brittle in the context of obfuscation, which often employs safeguards [14], [16], and it still does not guarantee complete path coverage, as we will show in the following section.

VPC-sensitive abstract interpretation. Kinder [23] proposes VPC-sensitive abstract interpretation. While traditional static analysis would merely analyze the VM's interpreter, VPC sensitivity accounts for this layer of indirection. The key insight is that in programs under virtualization-based obfuscation, different abstract states exist for different execution contexts, but are merged under the same program location. For example, the VM's `add` handler may be called from different contexts, whenever addition is needed. We can account for this by adding a dimension, bytecode location sensitivity, to our analysis: By tracking the location of the bytecode (i.e., point in program execution) calling the VM handler, static analysis can now differentiate between calls to the same location. Kinder demonstrates this idea using VPC-sensitive abstract interpretation to recover static-analysis facts such as possible external-call argument values, but it does not target complete, simplified, decompilable CFG recovery for commercial-strength virtualization-obfuscated binaries. PUSHAN builds on the same VPC-sensitivity principle, but improves its scalability and real-world applicability using constraint-free symbolic execution, symbolization, and semantics-preserving simplifications. Appendix A quantifies the impact of our improvements.

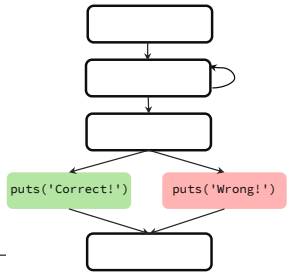
Existing techniques either remain coverage-limited when relying on traces, or provide partial analysis results rather than fully deobfuscated program representations. Scalable recovery of complete CFGs and human-consumable output for binaries protected by commercial-strength virtualizers remains unsolved in practice.

III. CHALLENGES

Two challenges limit prior work and motivate our approach. Consider the example program in Figure 2a, which implements a simple hash-based input checker. While the original CFG (Figure 2b) is straightforward, the VMProtect-obfuscated version (Figure 3a) is significantly larger and structurally unrecognizable. This highlights the main obstacle: Conventional CFG recovery yields the structure of the VM interpreter rather than the original control flow of the protected program.

```
1  while (input_num != 0) {
2      hash = ((hash << 5) + hash) +
         ↪ (input_num % 10);
3      hash = hash ^ (hash >> 3);
4      input_num /= 10;
5  }
6  hash = hash & 0xFFFFFFFFFFFFFFFF;
7
8  // the hash of 4294967295
9  if (hash == 226292709949525)
10     puts("Correct!");
11 else
12     puts("Wrong!");
```

(a) Original source code



(b) **Unobfuscated** CFG

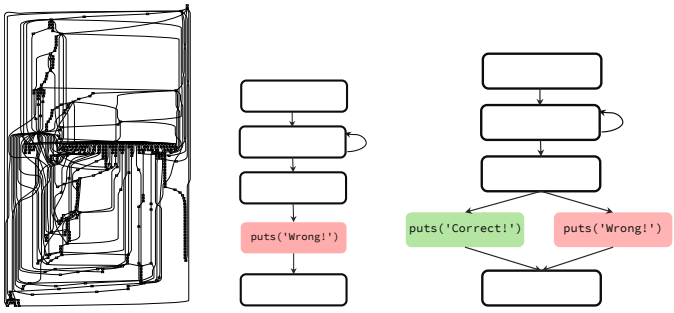
Fig. 2: Motivating example.

Challenge 1: Incomplete program behaviors captured. Prior work on VM deobfuscation relies on program *traces*

to identify executed code. A single trace, as in VMHunt [6] and Coogan et al. [28], is insufficient to retrieve the *complete* behavior. Even the simple `hash` example requires at least two traces to capture both the "success" and "wrong" paths. A single trace (assuming it triggers the "wrong" path) will lead to an incomplete deobfuscated CFG as shown in Figure 3b.

Challenge 2: Constraint solving-based input generation does not scale. Two works attempt to address Challenge 1 [4], [5]. To observe *all feasible* input paths, they turn to repeated dynamic symbolic execution (DSE). They symbolically execute a trace and collect path constraints, then negate them at branching points and generate an input to trigger the alternate branch. They repeat the process to discover more traces, hoping to reach full path coverage of the protected code.

Unfortunately, this approach inherits the downsides of symbolic execution: It suffers from *path explosion*, where there are (nearly) an infinite number of paths in complex, real-world programs, and it lacks a model for the *environment*, which complicates its application in general. Most importantly, both obfuscation and the original program logic can field *constraints that are difficult to solve* for theorem provers [15], [16]. The sample program `hash` will only execute `puts("Correct!")` if the user input hashes to a certain value. Without having the intended user input (in this case, `4294967295`), a theorem prover must break the hash algorithm's preimage resistance and find an input that satisfies `hash == 226292709949525`. We attempted to solve for the correct input using angr and Z3 on the unobfuscated binary. Z3 was unable to find a valid 64-bit input within three hours, which shows the practical limitations of constraint-solving-based approaches. These weaknesses hamper the scalability of symbolic execution as a means of generating new inputs to discover more program behaviors. Obfuscation has moved to specifically target symbolic execution [13]–[15], [34], rendering it even more ineffective.



(a) (b) (c)

Fig. 3: CFGs recovered (a) directly from VM-obfuscated code, (b) by trace-based approach, and (c) by PUSHAN.

IV. THE DESIGN OF PUSHAN

PUSHAN aims to recover a *complete control-flow structure* of a virtualization-obfuscated binary in a trace-free manner. Our solution depends on two insights:

Bounding symbolic state growth during CFG construction. PUSHAN recovers the CFG by *symbolically emulating* the program from the entry point and incrementally constructing the CFG. Each time symbolic emulation encounters a control-flow transfer, a corresponding node and edge are added to the graph. Traditional DSE must preserve a distinct symbolic state for every execution path reaching a program location, since path feasibility depends on how that location was reached. This leads to path explosion due to branches and loops.

In contrast, we observe that CFG recovery is a *structural* problem. Once a basic block, identified by both its block address and VPC, has been discovered and emulated, re-executing that basic block along different paths does not reveal new control-flow structures. Accordingly, PUSHAN symbolically emulates each block at most once. If additional incoming edges to an already-discovered block are later encountered, these edges are recorded in the CFG, but the block itself is not re-emulated. By collapsing all incoming paths at block boundaries, PUSHAN bounds the number of symbolic states and avoids the path explosion inherent to traditional symbolic execution.

Eliminating reliance on constraint solving. In non-obfuscated binaries, CFG recovery can often be performed syntactically by inspecting branch instructions and extracting explicit targets. Virtualization-based obfuscation breaks this assumption by replacing direct and conditional jumps with indirect jumps whose targets are computed through complex expressions, often involving MBA transformations and values defined far earlier in the program. Some form of symbolic reasoning is necessary to propagate values to jump sites. However, traditional DSE accumulates path constraints and queries an SMT solver to determine whether a branch is feasible. This reliance on satisfiability checking introduces significant overhead and hinders scalability, especially in obfuscated code.

PUSHAN eliminates this dependency by observing that CFG recovery does not require path feasibility. Our goal is not to determine whether a jump can occur, but to enumerate the set of control-flow targets it may resolve to. Accordingly, when resolving indirect jumps or opaque predicates, PUSHAN queries an SMT solver on the recovered symbolic expression *without any accumulated path constraints*. The solver is used purely as an expression simplifier and value enumerator, not as an input feasibility checker. This constraint-free usage avoids solver blowups while still recovering all CFG edges.

PUSHAN decouples CFG recovery from path feasibility. As Figure 3c shows, PUSHAN recovers both feasible and infeasible paths of the `hash` example, yielding a complete CFG (and subsequently, decompiled and simplified pseudocode).

A. Overview

PUSHAN utilizes a three-stage pipeline (Figure 4) to recover the original control flow and produce decompiled code.

Stage 1. VPC-sensitive CFG recovery (Section V). PUSHAN reconstructs the CFG by emulating the obfuscated binary using an abstract state over registers and memory. By uniquely

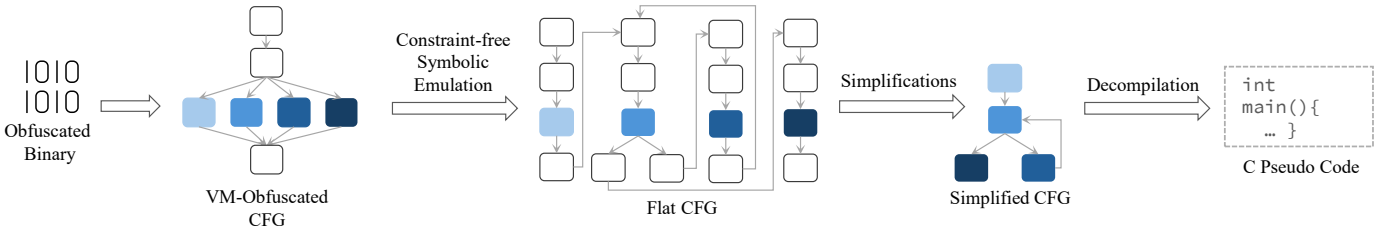


Fig. 4: Overview of the analysis pipeline of PUSHAN.

identifying blocks by both their native address and virtual program counter (VPC), PUSHAN achieves context-sensitive reconstruction. It propagates values through VM instructions to simplify jump-target expressions, enabling the resolution of indirect jumps and the pruning of constant opaque-predicate branches. It then uses symbolization to recover edges missed during emulation. The output of this stage is a CFG with both VM interpreter logic and the original, unobfuscated program's logic. We call this CFG a *flat CFG*.

Stage 2. Semantics-preserving simplifications (Section VI). A flat CFG contains redundant low-level logic inherent to the VM logic. In this stage, PUSHAN applies a series of semantics-preserving transformations on the flat CFG to remove unnecessary artifacts, simplify control flow, and reveal higher-level operations. PUSHAN iteratively applies these simplifications until convergence.

Stage 3. Decompilation (Section VII). Finally, the simplified CFG is processed by an enhanced decompiler to generate C-like pseudocode that preserves the recovered logic. Our enhancements include rewriting node identifiers into a decompiler-compatible format, improving function boundary detection despite obfuscated calls, and extending stack pointer tracking to handle non-standard arithmetic.

B. Threat Model

PUSHAN targets function-level virtualization-based obfuscation in which the virtualized execution maintains a recoverable virtual program counter (VPC). The core CFG-recovery approach is generic to this setting: Given the protected function entry and a trackable VPC, PUSHAN symbolically emulates the virtualized code and reconstructs the function's control-flow structure. We do not claim robustness against arbitrary combinations of binary protections. PUSHAN's simplification and decompilation stages rely on patterns observed in our evaluated obfuscators; other obfuscators or different protection configurations may introduce opaque predicates, pointer-aliasing patterns, conditional-jump sequences, nested virtualization, or other transformations that require additional preprocessing or simplification rules.

Protections orthogonal to virtualization, including packing, MBA, and nested virtualization, are out of scope. Packed binaries can be analyzed only after they are unpacked or dumped from memory. When a binary contains multiple virtualized functions, PUSHAN analyzes each function separately. Recovering the CFG of a caller does not require analyzing callees; We model callees with stubs while PUSHAN continues recovering the caller's control-flow structure.

V. VPC-Sensitive CFG Recovery

Given an obfuscated binary, PUSHAN begins by performing VPC-sensitive CFG recovery. While general-purpose tools for automated VPC identification are generally unavailable, the underlying principles are studied by prior work and not our contribution [35], [36]. PUSHAN implements a set of targeted VPC identification heuristics inspired by existing work. We first describe these VPC identification methods for completeness, then detail our methodology for achieving a complete, VPC-sensitive flat CFG reconstruction.

A. Heuristic VPC Identification

In virtualization-obfuscated binaries, the VPC typically resides in a register or memory location and serves to index the VM bytecode stream. The VPC's location varies by obfuscator. It may be volatile, updated via multi-instruction arithmetic, and transferred between registers (as seen in VMProtect and VMProtect Ultra), or stay in a stable location throughout execution (Themida and Tigress).

To handle volatile instances, PUSHAN re-evaluates candidate VPC locations at the start of each basic block during symbolic emulation. Following the methodology in VM-Doctor [36], we identify the VPC by monitoring registers pointing to memory regions with high entropy (a characteristic of VM bytecode regions) and sequential evolution, where the VPC value increases or decreases within some region. For virtualizers with stable location VPCs, we monitor unique behaviors (e.g., the first memory load from a non-conventional section of a binary, which apply for both Themida and Tigress) and use them to infer VPC locations. By combining these strategies, PUSHAN could successfully identify the VPCs for all evaluated samples across VMProtect, Themida, and Tigress.

For custom CTF virtual machines, PUSHAN extends the entropy-based VPC identification strategy beyond VMProtect-style register VPCs. The shared strategy is to first identify VM data regions whose byte distribution differs from native code, then look for program values that consistently index, point into, or advance through those regions during the deobfuscation window. For register-index VMs, the finder scores registers that are used in memory operands into high-entropy bytecode or VM-state regions and whose values move like a VM program counter. For stack-based VMs, the finder monitors stack locations that are repeatedly updated with

bytecode offsets or pointers, and locks onto a location once it consistently exhibits the same VPC-like update behavior across VM steps. For file-backed bytecode VMs, the finder tracks the bytecode stream cursor and distinguishes repeated decode-loop visits so repeated native decoder blocks produce distinct VM instruction contexts.

While these heuristics are tailored to commercial-strength virtualization-based obfuscators, they are grounded in fundamental properties of virtualization-based obfuscation: Bytecode indexing and predictable execution flows. We expect the VPCs of other virtualizers to remain similarly identifiable with minimal manual effort.

B. Constraint-Free Symbolic Emulation

During flat CFG recovery, PUSHAN emulates each instruction from the beginning of the obfuscated code and updates an abstract state. This abstract state includes storage for registers and memory. PUSHAN uses a custom value domain that includes concrete values (e.g., bitvectors and floating points), symbolic values, and a special abstract value `TOP`, which is created as the result of the union of two or more values when paths merge.

At the beginning of the emulation, we treat all uninitialized values, e.g., environment variables or user inputs, as symbolic. When encountering external function calls, including standard C library functions and OS APIs, we avoid emulating them. Instead, we simulate the function's behavior using a function summary when available or return a symbolic value that adheres to the expected calling convention. Function summaries make this modeling more precise by letting PUSHAN model common input-producing APIs, but they are not required for CFG recovery; when no summary is available, the current implementation returns a symbolic value that adheres to the expected calling convention. This fallback can be made more conservative by treating all function arguments as memory pointers to locations that may receive user-controlled input. We also hook any called unobfuscated functions in the binary to return symbolic values. This prevents unnecessary analysis and ensures symbolic return values flow correctly into the obfuscated logic. Note that existing techniques like VMHunt [6] can be applied to automatically identify the boundaries of the virtualized region and automate the hooking process if desired.

During emulation, PUSHAN creates a new node for each newly discovered basic block and adds a new edge between the source node and the new node in the CFG when a control flow transfer happens. To implement VPC sensitivity, we uniquely label each node using a tuple of (block address, VPC). We refer to this tuple as the ID of this node. After the initial VPC-sensitive CFG exploration, our graph looks like Figure 5c.

During each exploration round, PUSHAN visits each VPC-sensitive block at most once; exploration terminates when no new (address, VPC) pairs are discovered. After each exploration round, symbolization (Section V-C) may introduce additional symbolic values, which enables the discovery of previously missed control-flow edges in subsequent rounds. This iterative process gradually expands the recovered CFG.

Pruning branches for direct conditional jumps. Virtualization-based obfuscators frequently apply opaque predicates to thwart static analysis effort, and a common use case for opaque predicates is to convert an unconditional jump into a conditional jump where one of its branches is always satisfiable [37]. The end result is an excessive number of redundant control-flow transfers in the obfuscated code. PUSHAN simplifies the CFG and conservatively prunes only the branches whose infeasibility follows from the recovered branch expression itself:

1) For each branch guard g, PUSHAN queries the SMT solver without accumulated path constraints to check whether g is satisfiable and whether $\neg g$ is satisfiable. If g is unsatisfiable, PUSHAN prunes the true successor; if $\neg g$ is unsatisfiable, PUSHAN prunes the false successor.
2) If both sides are satisfiable, or if the solver returns unknown or times out, PUSHAN keeps both successors.

Thus, PUSHAN handles constant opaque predicates and obfuscated predicates that simplify to constants under the current symbolic state. It does not perform general MBA simplification or use path constraints to prove path feasibility. MBA-heavy predicates or predicates whose infeasibility depends on earlier path conditions may therefore remain in the CFG and increase analysis cost. Existing MBA simplification methods can be integrated to reduce such expressions before the solver query [38]–[40].

After pruning, our graph looks like Figure 5d, where all the flattened handler copies unrelated to the current VPC are eliminated along with fake branches that depend on opaque predicates (not shown in the graph).

Resolving obfuscated indirect jump targets. In traditional binary disassembly, determining the target of a conditional jump is straightforward, as the target address is embedded directly in the instruction itself (e.g., `jz $+5`). In contrast, obfuscated binaries often replace direct jumps with indirect jumps, where the target address is computed. We use symbolic emulation to recover the intended jump targets by executing the relevant instructions and tracking how the jump target is calculated. Obfuscators often apply MBA transformations to make these expressions harder to interpret. Listing 1 shows an example. To resolve such indirect jumps, PUSHAN applies the same strategy used for pruning opaque branches: It evaluates the recovered expression in isolation, without applying any path constraints. We stress that the goal is not to determine which input would trigger the jump, but to enumerate all possible outputs of the expression—that is, the set of target addresses it may evaluate to. This mirrors traditional disassembling techniques, which recovers jump destinations rather than reasons about the conditions under which they are taken.

[b]

```
1 ~(~(0xfffffffff + (0x0 .. ~(if ~(0x5 + ~input) == 0x0
      ↪ then 1 else 0))) | -5369364481)
2 + ~((0xfffffffff + (0x0 .. (if ~(0x5 + ~input) == 0x0
      ↪ then 1 else 0))) | -5369367216)
```

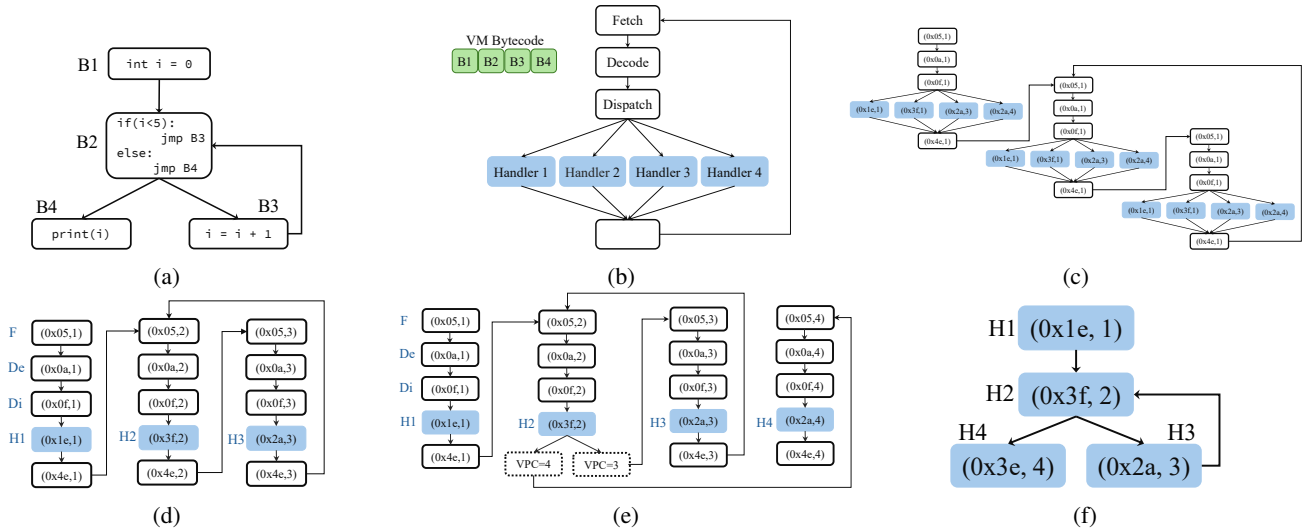


Fig. 5: The PUSHAN deobfuscation workflow applied to a virtualized example. (a) The original CFG of the target program. (b) The VM-obfuscated CFG, where individual handlers encapsulate the semantics of original basic blocks (B_1–B_4). (c) Initial VPC-sensitive CFG recovery. (d) CFG refinement after eliminating infeasible branches and unreachable paths. (e) Recovery of the loop exit condition following the symbolization of variable i. (f) Final CFG after simplifications.

Listing 1: Example of an obfuscated jump target calculation using symbolic variable `input`.

C. Symbolization

Pruning may lead to missing branches during the first analysis pass of a basic block. Consider the exit branch of a loop that iterates for a constant number of iterations. The obfuscated guard condition of the exit branch will only evaluate to one value (going to the beginning of the loop) unless the execution reaches the last iteration. PUSHAN would incorrectly prune the exit branch, leading to an incomplete flat CFG. To address this problem, PUSHAN introduces *symbolization* to discover and symbolize non-constant variables.

1) When emulating each instruction, PUSHAN keeps the values in registers and memory in an abstract state. PUSHAN also resolves the addresses of load and store instructions.
2) When two paths merge during emulation (e.g., a back edge reaching the loop header), PUSHAN merges the abstract states for the two paths, where constant but distinct values at the same location are merged into TOP (an unconstrained symbolic value). For dealing with opaque predicates, we use an SMT solver (without constraints) to check if an expression evaluates to a constant. We also save the information about which locations have been symbolized and at which block ID.
3) Symbolization terminates once the set of symbolic locations reaches a fixed point (i.e., no new symbolic locations are introduced through state merging).
4) When performing CFG recovery (Section V-B) again, PUSHAN can use the saved information regarding symbolized locations.

Symbolization allows for symbolic values to flow into the obfuscated conditions of conditional jumps and discover both branches. Symbolization allows PUSHAN to retrieve all possible jump targets for these types of indirect jumps by symbolizing the constant value that the condition checks for and exploring both branches.

Figure 5d examplifies symbolization, where initializing `i=0` makes the loop-exit guard false and prunes the edge. After symbolization, `i` becomes symbolic, and both successors are explored (Figure 5e).

Increasing scalability by limiting symbolic solves. Many obfuscated expressions can be simplified to constants using an SMT solver, but it is time-consuming and does not scale. We choose a granularity where we resolve the constants by deciding to not solve for all possible expressions, and only solving for certain types of expressions, e.g., the addresses for loads and stores, because these expressions are usually for loading VM bytecode from the bytecode region. This way, we reduce the time spent in solving and still get the constants that are essential for simplifying away the VM machinery.

Mitigating the NP-hardness of path satisfiability. Existing approaches [4]–[6], [28] first collect an execution trace, then accumulate path constraints, and finally query the solver for inputs that satisfy yet-unexplored paths. In contrast, PUSHAN sidesteps full path satisfiability by adopting three lightweight techniques:

(i) **Constraint-free simplification.** To enumerate jump targets, we pass the SMT solver only the target expression, omitting all accumulated path constraints. This keeps each query lightweight and tractable.
(ii) **Symbolization.** Whenever two paths merge, any differing concrete value is replaced with a fresh symbolic variable

(e.g., the initial value 0 of `hash` in Listing 2a), dramatically shrinking the expressions handed to the solver.

(iii) **Single-iteration loop handling.** Each loop is executed at most once during symbolic emulation, preventing complex expressions from forming—which would otherwise be harder for the solver to handle. For instance, in Listing 2a the symbolized `hash` variable is computed over only one iteration; although this yields an approximated expression, it is sufficient for recovering all jump targets, as we do not care about the actual feasibility.

These three tactics suffice in practice, but they can be combined with modern MBA simplifiers [38]–[40] to reduce complex symbolic expressions into forms from which the jump targets can directly be inferred.

VI. Semantic-preserving Simplifications

A flat CFG is complete but insufficient for decompilation (or many other downstream analyses), because it contains many redundant instructions that correspond to VM interpreters. PUSHAN iteratively applies a series of semantic-preserving simplifications on the flat CFG until reaching a fixpoint and producing a simplified CFG that resembles the one in the original unprotected program.

A. List of Simplifications

PUSHAN uses the following simplifications.

S1. Standard simplifications. PUSHAN applies constant propagation across the flat CFG. PUSHAN treats VM bytecode regions as constant (read-only), allowing bytecode values to propagate. This propagation renders much of the VM interpreter machinery redundant (e.g., handler dispatch logic that loads the handler address from the bytecode region) and eliminatable by subsequent simplifications.

PUSHAN then applies dead assignment elimination to remove redundant bytecode-processing logic. This step also eliminates opaque predicates by resolving their guard conditions.

Lastly, PUSHAN performs basic arithmetic simplifications, e.g., grouping together the arithmetic operations performed on the same register. This simplification pass reduces the arithmetic operations left after dead assignment elimination. These transformations follow the standard correctness argument for constant propagation and dead-assignment elimination [41]. PUSHAN performs a constant replacement only when data-flow analysis establishes a unique value at that program point; if merged predecessors disagree or the value is unknown, PUSHAN leaves the expression unchanged. Dead assignments are removed using reaching-definitions and definition-use analysis only when the definition has no live use.

S2. Redundant stack variable removal. A common pattern in simplified code is a value being stored at a stack location and immediately loaded into a register (and the stack location is never loaded after). PUSHAN simplifies this pattern by removing the redundant stack write. To avoid issues with pointer aliasing, we only apply this simplification within a basic block. This is a local copy-propagation rule: the load is replaced by the stored value only when data-flow analysis reports a single reaching store, the load and store widths match, and the location is a stack slot or concrete address. If there is an intervening write, a partial-width access, or an unresolved alias, PUSHAN does not apply the rewrite. Figure 9 in the appendix shows an example.

S3. Removing self-defining variables in loops. PUSHAN further removes self-referential stack and global variables that persist after dead assignment elimination. These variables, often remnants of loop counters, are pruned if their only use is self-incrementing or self-referencing. To maintain semantic integrity, PUSHAN preserves any variable that influences control-flow guards or serves as a function argument. The definition-use graph must show that the variable has no path to a branch guard, call argument, return value, or other live architectural output. Thus, PUSHAN removes only VM-internal bookkeeping that is unobservable in the recovered function semantics.

S4. Obfuscator-specific simplifications. Both VMProtect and Themida use transformations that are not simplified by previous techniques. For example, Themida converts every conditional jump into two conditional jumps. The first jump computes the check and sets a global variable. The second conditional jump checks the value of the global variable and chooses the correct branch. PUSHAN includes obfuscator-specific simplifications to simplify such patterns.

B. Differences to Prior Work

While prior work [4] has implemented some simplifications (S1), we note some key differences below.

- Unlike prior work that operates on individual traces [4], PUSHAN analyzes a complete CFG. This global perspective mitigates the risk of over-simplification inherent in trace-based approaches, which rely on precise, end-to-end taint tracking of user input to maintain correctness. PUSHAN remains input-agnostic and does not require a-priori knowledge of input sources or their propagation.
- Prior work may yield different simplification results when simplifying multiple traces, which makes merging traces into a CFG difficult. PUSHAN simplifies the CFG directly, so it does not need to reconstruct it.
- PUSHAN implements more simplifications (S2, S3), as well as obfuscator-specific simplifications (S4).

VII. Decompilation

PUSHAN takes the simplified CFG from the previous stage and decompiles the entire CFG using a customized open-source binary decompiler. We discuss key steps in this section.

Identifying function boundaries. Most binary decompilers only decompile a function at a time. However, the function boundaries are unclear in obfuscated binaries, especially the function return sites, as there are many fake call and return statements that do not jump to or return from any functions. Additionally, some obfuscators will rewrite `call` instructions

```
not(not(rsp)+0x18) ==> rsp-0x18
```

Fig. 6: An example of using an unconventional operation, Negation, when computing stack pointer offsets.

into `jmp` instructions to further obfuscate the control flow. While most bogus call and return instructions are removed during the simplification stage, there still remain some jump or return statements that actually transfer control flows. To address this, PUSHAN rewrites return statements and jump statements that go to functions inside the unobfuscated code of the binary into call statements.

Enhancing stack pointer tracking. Decompilers usually assume that specific registers (e.g., `rsp` on X86-64) are always stack pointers. This assumption does not always hold with obfuscated binaries, where `rsp` can be repurposed to hold generic values, causing the decompiler to fail to track or recover stack variables. Obfuscators may also insert code to conduct unconventional arithmetic operations, such as Negation, Xor, or MBA expressions, on stack pointer registers. This also curtails the stack pointer tracking that decompilers perform. Figure 6 shows such an example. We enhanced the open-source decompiler to discover the actual stack pointer registers and added support in stack pointer tracking for unconventional arithmetic operations and simplification rules.

VIII. EVALUATION

We first evaluate the effectiveness and correctness of PUSHAN on a diverse set of binaries for which we have the ground-truth (original source code or CFG). Our experiments include analyzing the deobfuscation and decompilation output (Section VIII-C), comparing the deobfuscated CFGs to the original CFGs and quantifying their similarity (Section VIII-D), and validating the semantic correctness of the deobfuscated code (Section VIII-E). We then show with a case study that PUSHAN can achieve logic recovery using deobfuscated and decompiled code from a malware sample (Section VIII-F). Lastly, we evaluate the effectiveness of PUSHAN on custom virtualization-based obfuscation implementations in binaries from Capture-the-Flag (CTF) competitions, for which we do not have ground truth (Section VIII-G).

A. Dataset

To thoroughly evaluate PUSHAN, we constructed a dataset comprising multiple program categories. It includes six Windows malware samples used in prior work [4], four open-source projects, four hand-crafted binaries, five challenges from Capture the Flag (CTF) competitions with custom VM implementations, 1,000 Tigress-generated virtualized binaries, and a real-world VMProtect-obfuscated executable. We distinguish four units in the evaluation. A *program* is an original source project, malware sample, synthetic sample, or challenge. A *protected function* is the function selected for virtualization and analysis. A *binary* is one concrete whole-program executable produced under a particular obfuscator configuration, where the selected protected function is virtualized. A *VM setup* is the obfuscator and protection configuration used to produce that binary. For the malware, open-source, and synthetic samples in Table III, each row corresponds to one function selected for protection inside a whole-program binary; PUSHAN analyzes that function at its entry in the obfuscated binary. For each selected protected-function entry, we build one original whole-program binary and three obfuscated whole-program binaries, using three configurations: VMProtect, VMProtect Ultra, and Themida. We detail these configurations in Table VII in the Appendix for interested readers. The VMProtect-packing experiment adds two packed binaries derived from the synthetic samples and is reported separately as a robustness experiment. We selected functions that contain nontrivial control-flow or data-flow behavior, such as loops, conditionals, and function calls, while also covering realistic functionality including decompression, URL parsing, event handling, and malware-specific backdoor or spreading logic.

Real malware. Our dataset includes six Windows malware samples (source code available) used in prior work [4]. The malware contain typical malicious behaviors, e.g., backdoors and spreaders, demonstrating PUSHAN's intended use case of malware analysis. Across all malware, open-source, and synthetic samples in Table III, the original versions totaled 1,800 instructions, which expanded to 1,659,751 instructions under VMProtect, 3,150,532 instructions under VMProtect Ultra, and 9,654,038 instructions under Themida. For comparison, the 1993 remake of *Doom* contains under 4 million instructions.

Open-source software. We selected one function each from four widely used libraries—zlib (`gz_read`), curl (`glob_url`), libpng (`png_decompress_chunk`), and SDL (`ClosePhysicalCamera`). These cover reading bytes from a file, URL parsing, decompressing data, and device event handling, respectively.

Synthetic samples. We created four synthetic test binaries to evaluate obfuscation scenarios not covered by the malware samples. These include: 1) *if-cond*, which contains a simple input-dependent conditional jump, 2) *hash*, which implements the motivating example in Listing 2a, 3) *const-loop*, which features complex loops that challenge DSE techniques, and 4) *huffman* for comparability to prior work.

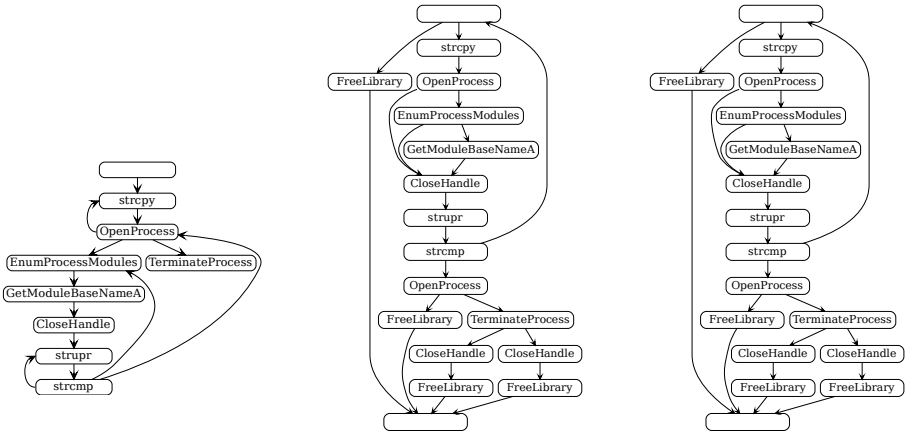
Custom VMs from CTFs. We included five CTF challenges with bespoke virtualization-based obfuscation to test PUSHAN's generalizability. These challenges all implement *custom* VM architectures and do not provide original CFG ground truth, so we validate them using API-trace and public-writeup checks in Section VIII-G.

Large-scale correctness dataset. For Tigress, we used the random function generator to produce $1,000$ hash functions, which we then obfuscated using virtualization along with the following configurations:

- **VM-1**: Stack-based VM with switch-case dispatch.
- **VM-2**: Opaque predicates, duplicate opcodes, direct dispatch, super operators, and additional obfuscations.

TABLE II: VPC identification techniques used for custom CTF virtual machines.

CTF Challenge	VPC Pattern	VPC Finding Mechanism
Discount VMProtect / RCTF	Register/index-like VPC into VM data	Entropy-guided register/index finder with source locking
Highly Optimized	Register/index-like VPC into VM data	Entropy-guided register/index finder
simple-vm / x-mas	Stack slot stores raw bytecode offset	Entropy-guided stack raw-offset finder
vmwhere1	Stack slot stores pointer/position for VPC	Stack stored-pointer finder
reduced-reduced-instruction-set-1	File-backed bytecode cursor	File-cursor VPC tracker with decode-loop namespacing



(a) Prior work's deobfuscated call graph of Netsky [4], with missing blocks and API calls.

(b) Original call graph of Netsky.

(c) Deobfuscated call graph by PUSHAN.

Fig. 7: Comparison of Netsky's original function call graph (middle), prior work's deobfuscated function call graph (left), and the deobfuscated, complete call graph by PUSHAN (right).

- **VM-3**: Virtualization with bogus functions, implicit flows, and opaque predicates.

Tigress operates at the source level, so we compiled the obfuscated source code to generate the final obfuscated binaries.

A real VM-obfuscated sample. Lastly, we evaluated PUSHAN on a real-world VMProtect-obfuscated binary. The original unobfuscated binary and source code are unavailable, so we use this sample for a qualitative logic-recovery case study rather than CFG-similarity measurement.

B. Experiment Environment and Design

We ran all experiments on a Kubernetes cluster with 2.30GHz Intel Xeon CPUs, allocating up to 80GB of RAM per sample. We selected two popular commercial binary obfuscators: VMProtect 3.5.0 [1] and Themida 3.1.1.0. For the standard VMProtect and Themida configurations, we enabled virtualization-based obfuscation and disabled non-virtualization protections to isolate PUSHAN's core contribution: scalable CFG recovery for virtualization-protected functions. We separately evaluate two more realistic VMProtect settings: VMProtect Ultra, a vendor-defined configuration described as combining mutation with virtualization, and VMProtect virtualization combined with output packing.

To evaluate VMProtect virtualization combined with VMProtect's output-packing option, we tested two synthetic samples, *if-cond* and *const-loop*. Since automated unpacking is outside PUSHAN's scope, we first executed each packed binary until VMProtect unpacked it in memory, then dumped the unpacked image and used that dump as input to PUSHAN. PUSHAN recovered both protected functions from these dumps. The recovered CFGs matched the original CFGs with 100% similarity after excluding a VMProtect-inserted protected prelude introduced by packing. This prelude fingerprints the runtime environment and transfers control into the virtualized program logic only if the checks succeed. Thus, for these samples, packing did not prevent recovery of the virtualized program logic after unpacking.

All our Themida-obfuscated samples use the Fish VM variant. To assess generalizability across different VM configurations, we also tested two samples each with the Tiger and Dolphin VM variants. In all cases, PUSHAN achieved the same CFG similarity scores across variants.

Each VMProtect and Themida sample in our dataset was compiled with and without the respective obfuscator SDKs, resulting in four binaries per sample: one original (used as ground truth), one VMProtect-obfuscated binary, one VMProtect Ultra-obfuscated binary, and one Themida-obfuscated binary.

We implement a prototype of PUSHAN on angr [11]. Our prototype mainly works with VEX IR during CFG recovery and simplification, then transitions to angr IL (AIL) for decompilation. Our heuristic-based VPC finder automatically identifies the VPC for VMProtect, Themida, Tigress, and the

[1]The authors of this paper cannot purchase newer versions of VMProtect for compliance reasons.

CTF challenges. For CTF challenges, we use the VPC identification modes summarized in Table II. We will release the tooling and all evaluation artifacts upon publication, including a documented end-to-end case study for an actual binary.

Comparison to prior work. Yadegari et al. [4] did not fully released their tool. Upon contacting them, we learned that the concolic execution component of their tool is not publicly available. Instead, they only released the final CFGs for some of the samples. These CFGs consist solely of the graph structure, without any accompanying deobfuscated code, which is insufficient for accurately comparing similarities. VMHunt [6] provides source code but lacks runnable samples, test binaries, or documentation needed for reproducing their results. Salwan's framework [5] targeting Tigress binaries is available and functional. We use it as a baseline for evaluating our approach on the Tigress binaries of our dataset.

C. Effectiveness

To show that PUSHAN handles real-world malware and aids reverse engineering, we analyzed multiple aspects of decompilation.

Comparison against original source code. Table III compares lines of decompiled code against source code for programs whose original code is available. Some binaries, such as *huffman*, show significant line expansion due to obfuscated conditional branch representation. VMProtect, for instance, replaces conditional jumps (*jz*, *jle*) with flag bit calculations. This spreads condition checks across multiple instructions that our semantics-preserving transformations cannot simplify to the original form. Resolving this requires mapping instruction sequences back to original conditional jumps. We leave it as future work as it does not impact our core technique.

For Tigress samples (Table V), the line count increase stems from limited code-level simplifications rather than control-flow complexity. PUSHAN outputs one arithmetic operation per line for Tigress, while the original source combines multiple operations per line. This increases line count despite equivalent underlying logic.

Runtime performance. PUSHAN's performance scales with binary size and the number of constant-guarded nested loops, requiring one symbolization run per such condition. The timing results show that VMProtect Ultra and Themida generally take longer than standard VMProtect on these samples. Figure 8 in the appendix shows the runtime breakdown by stage for VMProtect, VMProtect Ultra, and Themida.

D. Correctness: CFG Similarity

We first measure the graph edit distance (GED) of our deobfuscated CFG to the original CFG. Previous work [4] utilized the Hu GED algorithm [42] to measure the approximate similarity of their graphs. We found this GED estimation algorithm inapplicable for two reasons. First, the Hu algorithm relies on the presence of x86 instructions to improve its results by matching opcodes—PUSHAN does not produce x86 instructions. Second, the Hu algorithm often incorrectly reports identical graphs as different. For example, Hu's GED algorithm reported a non-zero distance between the original CFG and PUSHAN's recovered CFG for the Themida protected *hunatcha* sample, despite the two 16-node graphs being structurally identical.

For more accurate CFG measurement, we extended the CFGED algorithm by Basque *et al.* [43] to handle addressless graphs. The CFGED algorithm relies on a partial mapping of nodes across two CFGs, commonly collected by matching the addresses of node pairs. After obfuscation and subsequent deobfuscation, this address association is lost. Instead, we automatically created a mapping using information from the CFG by extending the discovRE [44] block-similarity algorithm (see Algorithm 1 in the appendix). First, we matched all nodes with the same function calls. We disambiguated candidates by matching node degrees, then distance from the graph root. Then we ran the discovRE algorithm to match any nodes at least 60% similar. We manually validated that all of these mappings were correct on all of our samples, and will release our tooling. We further evaluated the effectiveness of our extended CFGED algorithm on CFGs used in the original CFGED work [43]. This set consisted of 491 unique function pairs across optimized and unoptimized binaries from Coreutils. Of the 491 pairs, our algorithm had the exact score on 389 pairs (79%), while the Hu algorithm was exact on 36 pairs (7%). Additionally, our algorithm introduced an average similarity error of 3% while the Hu algorithm introduced 22%.

For CTF challenges lacking ground truth, we cannot perform graph edit distance matching. Instead, we approximate similarity by collecting an API trace from a single execution and verifying whether our recovered CFG can produce the same API sequence through DFS traversal.

Results. Table III shows the normalized similarity score produced by our extended CFGED algorithm. We normalized our GED score as described in prior work [4].

Interpreting non-isomorphic CFGs. CFG isomorphism requires the recovered and original CFGs to have the same graph structure under the validated node mapping. Therefore, a recovered CFG can be complete but non-isomorphic. We investigated the non-isomorphic cases observed in our evaluation and found the following causes. First, PUSHAN's simplifications are conservative and may retain obfuscator-introduced nodes that do not affect the original semantics. Second, one original node may appear as multiple semantically equivalent nodes in the recovered CFG when the observed VPC update occurs before the corresponding original basic-block boundary, which is an obfuscator-specific implementation detail. Third, the original unoptimized binary may contain semantically infeasible branches, such as branches guarded by constant-false predicates; PUSHAN may remove these edges during simplification. Although symbolization may in principle introduce jump targets that are never taken, we did not observe this case in our evaluation.

We evaluated the artifacts from prior work [4] using our extended CFGED algorithm. As Table III shows, PUSHAN

TABLE III: Evaluation results for open source software, malware and synthetic programs showing the impact of virtualization-based obfuscation. VMProtect, VMProtect Ultra, and Themida expand binaries by several orders of magnitude. CFG similarity is measured using our enhanced graph edit distance metric, showing high CFG similarity across most samples.

Program	Function	Instructions				LoC	DLoC			Orig. CFG		Similarity on VMP				Similarity on VMP-U				Similarity on TH			
		Orig	VMP	VMP-U	TH	Orig	VMP	VMP-U	TH	N	E	Pushan	Yadegari	ΔN	ΔE	Pushan	Yadegari	ΔN	ΔE	Pushan	Yadegari	ΔN	ΔE
zlib	gz_read	188	267,387	464,622	669,613	70	551	571	121	31	46	100%	–	0	0	100%	–	0	0	100%	–	0	0
curl	glob_url	133	152,889	320,732	704,548	42	182	316	52	15	22	74%	–	0	-1	85%	–	1	0	74%	–	0	-1
libpng	png_decompress_chunk	274	351,459	639,658	1,069,308	134	670	894	158	41	59	100%	–	0	0	45%	–	3	4	100%	–	0	0
sdl	ClosePhysicalCamera	139	177,967	346,495	829,215	50	460	430	220	16	22	82%	–	1	0	88%	–	1	0	100%	–	0	0
netsky	main	150	144,233	233,062	679,412	33	148	179	105	24	35	96%	33%	1	1	96%	–	1	1	96%	19%	1	1
hunatcha	InfectDrives	99	47,722	122,359	791,110	22	68	706	21	6	7	100%	30%	0	0	86%	–	1	1	100%	23%	0	0
cairuh	kazaa_spread	186	131,410	250,636	1,466,294	28	318	318	108	20	30	100%	–	0	0	100%	–	0	0	100%	–	0	0
blaster	blaster_spreader	140	91,250	150,006	773,783	64	117	122	89	11	16	100%	49%	0	0	93%	–	1	1	100%	–	0	0
newstar	InfectExes	81	30,957	75,668	489,129	18	45	45	20	6	6	100%	53%	0	0	100%	–	0	0	100%	18%	0	0
newstar	Backdoor	116	40,088	150,833	848,281	31	25	25	45	4	4	100%	–	0	0	100%	–	0	0	100%	–	0	0
huffman	create_huffman_codes	205	97,868	206,816	834,083	30	327	987	96	21	28	90%	33%	0	-1	91%	–	0	0	81%	31%	1	0
hash	main	51	54,374	100,441	273,375	12	107	113	26	7	8	100%	–	0	0	100%	–	0	0	100%	–	0	0
if-cond	main	15	33,935	42,013	163,944	6	7	107	11	4	4	100%	–	0	0	100%	–	0	0	100%	–	0	0
const-loop	main	23	38,212	47,191	61,943	5	78	83	7	4	4	100%	–	0	0	100%	–	0	0	100%	–	0	0

DLoC: Decompiled Lines of Code (LoC) after Pushan's deobfuscation. **Similarity:** CFG similarity score computed using enhanced GED. ΔN**:** signed node difference, computed as deobfuscated CFG nodes minus original CFG nodes. ΔE**:** signed edge difference, computed as deobfuscated CFG edges minus original CFG edges. **Orig. CFG:** nodes (N) and edges (E) in the original binary's CFG. **Orig:** Instructions / Lines of Code (LoC) in the *original*, unobfuscated binary. **VMP:** VMProtect. **VMP-U:** VMProtect Ultra. **TH:** Themida.

TABLE IV: CTF Challenge Analysis. DLoC = lines of decompiled code. "API" indicates whether the recovered CFG contained the API trace from execution. "Writeup" indicates whether the recovered CFG matched functionality described in publicly available CTF writeups.

Challenge	DLoC	API	Writeup
reduced-reduced-instruction-set-1	132	✓	✓
Discount VMProtect	55	✓	✓
simple-vm	84	✓	✓
Highly Optimized	157	✓	✓
vmwhere1	2108	✓	✓

outperforms Yadegari's across all released artifacts. The best similarity score Yadegari et al. achieved was 53%, whereas PUSHAN consistently achieves significantly higher scores.

Open-source programs: PUSHAN generated isomorphic CFGs for *zlib* across all three commercial configurations, for *libpng* under standard VMProtect and Themida, and for *sdl* under Themida. The remaining open-source cases are non-isomorphic but remain explainable by the causes above.

Tigress: PUSHAN generated isomorphic graphs for 982 of 1,000 Tigress samples, as detailed in Table V. When evaluating Salwan's tool [5] on the same dataset, only 647 samples completed within three days. When comparing CFG similarity on these 647 common samples, PUSHAN significantly outperforms Salwan's tool, where PUSHAN achieves 643 isomorphic graphs compared to Salwan's 68.

For the main CFG-similarity benchmarks, PUSHAN restored isomorphic CFGs for 1,010 of 1,042 VM-obfuscated targets.

E. Correctness: Semantic Equivalence

We evaluate PUSHAN's correctness using Tigress-generated programs because they compute hash functions, which provide a simple, deterministic mapping from inputs to outputs. This enables automated validation through output comparison: If the deobfuscated program produces the same hash value as the original for a given input, we guarantee equivalent behaviors.

We evaluated semantic correctness on Tigress samples using input/output testing on the recovered VEX IR. Since Tigress programs compute hash functions, we tested each sample with 100 distinct inputs and compared the deobfuscated VEX IR outputs against the original binary.

TABLE V: Tigress Deobfuscation and Decompilation Results across three VM configurations. LoC = lines of code. "Verified" indicates successful semantic validation.

Metric	VM-1	VM-2	VM-3
Deobfuscation Results			
Total Files	334	333	333
Succeeded	330	329	329
Failed	4	4	4
Success Rate	98.80%	98.80%	98.80%
Total Time	1397.94 hr	2523.97 hr	1525.01 hr
Average Time	4.19 hr	7.58 hr	4.58 hr
Verification Results			
Files Processed	330	329	328
Succeeded	318	320	318
Failed	12	9	10
Success Rate	96.36%	97.26%	96.95%
Total Time	1430.73 hr	1335.43 hr	1717.03 hr
Average Time	4.34 hr	4.06 hr	5.23 hr
Decompilation Quality			
Decompiled Files	330	329	329
Avg. LoC (Original)	187	185	188
Avg. LoC (Obfuscated)	4824	4686	5166
Avg. LoC (Decompiled)	692	739	698
Max Lines (Original)	555	604	516
Max Lines (Obfuscated)	13619	13246	12827
Max Lines (Decompiled)	2172	2506	2273
CFG Similarity			
100% Similarity (Count/Total)	328 / 330	326 / 329	328 / 329

We present our results in Table V. The recovered IR produced correct outputs in nearly all successful deobfuscation cases. Specifically, 956 of 1,000 Tigress samples passed all 100 input/output tests, demonstrating that PUSHAN reliably preserves functional behavior. In some cases, decompilation failed while CFG-level verification succeeded.

F. Analyzing A Real Binary via Code Reuse

We analyzed a VMProtect-obfuscated malware sample obtained in the wild. According to VirusTotal, 28 out of 72

antivirus engines flagged this sample as malicious [45], revealing a wide disagreement among them. From reversing the non-obfuscated logic in this binary, we found that it uses a VM-obfuscated function to decrypt an embedded AutoIt script. However, attackers may have also implemented malicious behaviors in the obfuscated function. We must fully deobfuscate this function to determine whether the binary is malicious or not.

PUSHAN took a total of 85 minutes to deobfuscate and decompile the obfuscated function. The final pseudo-code (476 lines) correctly reflected the complete control flow, revealing no additional malicious behaviors other than conducting integer operations inside two loops.

We provided the pseudocode output to Claude (Opus 4.6) with instructions to produce a simplified, recompilable, and semantically equivalent C implementation. Notably, we gave the model only PUSHAN's decompiled pseudo-code, without access to the original binary, disassembly, or any auxiliary analysis tools. Claude reduced the 476-line function to approximately 30 lines of recompilable C, inferring that the function is "a pseudo-random number generator wrapper with hash-based seeding." This reduced function allowed us to extract the embedded AutoIt script. We then realized that the main feature of this script was patching another commercial utility, DiskGenius, to potentially bypass its license checks if the host machine is not using Simplified or Traditional Chinese codepages. This allows us to conclude that the binary is a crack, which is not malicious but may be used for piracy.

We note that while it is possible to extract the the embedded AutoIt script by debugging the binary and bypassing potential runtime protections (e.g., anti-debugging checks), dynamic analysis cannot reveal hidden behaviors in the VM-obfuscated function. This highlights the necessity of static deobfuscation and decompilation of PUSHAN.

G. Extensibility: Decompiling Custom VMs

Without ground-truth source code for CTF challenges, we validated correctness through API trace matching and comparison with public write-ups. All five challenges matched both API traces and write-up descriptions (Table IV). In one challenge (Listing 3, appendix), the decompiled output directly revealed embedded flag values in conditional branches.

CFG similarity. For all CTF challenges, we successfully reproduced the observed API call sequence by traversing the recovered CFG, confirming at least one correct execution path. We also manually verified that each recovered CFG contained the functionality described in public write-ups.

H. Decompilation Quality

We evaluate PUSHAN's decompilation quality using the DecBench pipeline [46], which assesses three dimensions: structural correctness, type correctness, and recompilation correctness. For each dimension, DecBench measures the edit distance from the original, unobfuscated source code, with lower values indicating higher quality. We apply DecBench to all 42 obfuscated binaries in the main evaluation set (Table III), including those used in the ablation study. Across these samples, PUSHAN achieves median edit distances of 20.50 for structure and 5.0 for types. By comparison, Hex-Rays 9.2 achieves median edit distances of 38.00 for structure and 5.0 for types. We omit recompilation correctness because it requires the original compilation toolchain for every sample, which is unavailable for parts of our dataset.

PUSHAN therefore reduces the median structural edit distance by 46% relative to Hex-Rays, largely because PUSHAN focuses on control-flow recovery. Future work should investigate obfuscation-aware type recovery and recompilability techniques to improve decompilation quality further.

IX. DISCUSSION

PUSHAN does not yet simplify complex or obfuscated arithmetic expressions such as MBA expressions. Without simplifying away obfuscated expressions, the decompilation output of PUSHAN may have a simplified control flow, but its semantics can still be hard to understand. Existing solutions (e.g., MBA-Blast) simplify many types of MBA expressions. LOKI [15] primarily uses MBA expressions to obfuscate the semantics of handlers, without focusing on obfuscating the VM structure itself. As PUSHAN does not support MBA simplification, we do not compare against these works.

Obfuscators can attack the symbolization process of PUSHAN by adding malformed control flows that result in the incorrect symbolization of actual constants. This could cause PUSHAN to generate a deobfuscated CFG with missing branches or spurious branches that are not present in the original program. We can improve the PUSHAN prototype by tracking the constant values more accurately: Avoiding merging symbolic variables to `TOP` and instead introducing a value domain where values are guarded by their path predicates. This way PUSHAN can split these merged values back to their original values and prevent incorrect symbolization.

Although PUSHAN is more scalable than deobfuscation solutions that rely on full-trace DSE, the current prototype of PUSHAN is still too slow due to the reliance on theorem provers to simplify MBA expressions to constants. Integrating existing MBA simplification techniques would greatly speed up PUSHAN by avoiding excessive use of the solver.

X. RELATED WORK

Virtualization deobfuscation. Kochberger et al. [37] provide a systematization of various VM deobfuscation techniques based on extracted artifacts, analysis efforts, degree of automation, and generalizability. Beyond the works on automated deobfuscation studied in Section II, Liang et al. [47] propose to use symbolic execution to identify handler semantics. However, this comes with the pitfalls of using symbolic execution for deobfuscation that PUSHAN avoids. Similar to other works, SEEAD [48] proposes dynamic taint analysis to identify control dependencies in obfuscated code, combined with aggressive path pruning. This increases performance but comes with the risk of missing paths.

MBA Deobfuscation. Orthogonal to our approach, various techniques focus on the simplification of MBA expressions, including techniques such as: (a) Pattern matching [49], [50], which relies on existing knowledge, limiting their scope; (b) Program synthesis [8], [9], [51], [52] generates simpler equivalent expressions but doesn't guarantee correctness; (c) Algebraic methods [38]–[40] that are based on the property that two semantically equivalent n-bit input variables are aligned for 1-bit input variables, which is effective for linear MBA expressions; (d) Deep learning [53], which takes a novel approach by training models to deobfuscate MBA expressions. Additionally, Arybo [54] employs the Bit-Blast method, simplifying arithmetic expressions to bit-level symbolic expressions, best suited for small expressions because of high-performance cost.

XI. Conclusion

PUSHAN is a scalable technique for deobfuscating and decompiling virtualization-obfuscated binaries. Building on the concept of VPC-sensitive CFG recovery, PUSHAN allows for the recovery of complete, pre-obfuscation CFGs without relying on user input that covers all branches of the obfuscated program or performing DSE on a sufficient number of traces and finding user input. The experiments show that PUSHAN can recover the CFGs of the original, unobfuscated programs with a high level of similarity and output decompiled code that can enable advanced malware analysis with LLMs.

Ethical Considerations

This work aims to advance defensive security capabilities through improved malware analysis.

Stakeholders. We identify three stakeholders, malware analysts, companies relying on virtualization-obfuscation for legitimate purposes, and society in general.

Impact. Our work has both a positive and negative impact. *Positive Impact:* Our work helps malware analysts to analyze protected malware at scale, as manual reverse engineering is time-consuming, error-prone, and does not scale to the volume of obfuscated malware encountered in practice. Helping to identify malware enables timely mitigation, thus benefitting society in general.
Negative Impact: Malicious actors could likewise use our code to target benign code obfuscated by companies to protect their intellectual property, prevent cheating in online games, or use it for Digital Rights Management (DRM) applications.

Mitigations. To avoid harm, we conducted all experiments on publicly available datasets and malware samples from public sources. Our analysis of the VMProtect-obfuscated sample revealed it to be a software crack rather than intentional malware. No human subjects or sensitive data were involved in our research, ensuring we respect persons. Ultimately, we cannot prevent misuse of our tool by malicious actors.

Justification for Research. We believe the benefits to defenders significantly outweigh potential harms. Virtualization-based obfuscation is widely used for legitimate purposes such as protecting intellectual property and preventing game piracy, but is also heavily employed by malware authors. Our work primarily benefits security analysts who need to analyze protected malware at scale, as manual reverse engineering is time-consuming, error-prone, and does not scale to the volume of obfuscated malware encountered in practice. We believe that publication of our research benefits the security community by enabling more effective analysis of protected malware, which is essential for developing defenses against evolving threats.

Acknowledgments

This work was supported by the Advanced Research Projects Agency for Health (ARPA-H) under Contract No. SP4701-23-C-0074, the National Science Foundation (NSF) under Grants No. 2232915 and 2146568, the Office of Naval Research (ONR) under Grant No. N00014-23-1-2563, and the Air Force Research Laboratory (AFRL) under Grant No. FA9550-24-1-0204. We acknowledge support from the HARDEN, SENSEI, STEEDDEMON, and Tools and Techniques for Characterizing Integrity in DevOps Technologies projects. We also gratefully acknowledge the generous support of the U.S. Department of Defense.

References

[1] Kaspersky, "The number of the year: Kaspersky detected half a million malicious files daily in 2025," 2026, https://www.kaspersky.com/about/press-releases/the-number-of-the-year-kaspersky-detected-half-a-million-malicious-files-daily-in-2025.

[2] C. R. Assis, R. S. Miani, M. G. Carneiro, and K. J. Park, "A Comparative Analysis of Classifiers in the Recognition of Packed Executables," in *IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2019.

[3] A. Mantovani, S. Aonzo, Y. Fratantonio, and D. Balzarotti, "Re-mind: a first look inside the mind of a reverse engineer," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2727–2745.

[4] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A Generic Approach to Automatic Deobfuscation of Executable Code," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2015.

[5] J. Salwan, S. Bardin, and M.-L. Potet, "Symbolic Deobfuscation: From Virtualized Code Back to the Original," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 2018, pp. 372–392.

[6] D. Xu, J. Ming, Y. Fu, and D. Wu, "VMHunt: A Verifiable Approach to Partially-virtualized Binary Code Simplification," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[7] S. Bardin, R. David, and J.-Y. Marion, "Backward-bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2017.

[8] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, "Syntia: Synthesizing the Semantics of Obfuscated Code," in *USENIX Security Symposium*, 2017.

[9] G. Menguy, S. Bardin, R. Bonichon, and C. d. S. Lima, "Search-based Local Black-box Deobfuscation: Understand, Improve and Mitigate," in *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[10] M. Balliu, M. Dam, and G. Le Guernic, "Encover: Symbolic Exploration for Information Flow Security," in *IEEE Computer Security Foundations Symposium (CSF)*, 2012.

[11] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "SoK:(State of) the Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2016.

[12] T. Seto, A. Monden, Z. Yücel, and Y. Kanzaki, "On Preventing Symbolic Execution Attacks by Low Cost Obfuscation," in *IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. IEEE, 2019.

[13] Z. Wang, J. Ming, C. Jia, and D. Gao, "Linear Obfuscation to Combat Symbolic Execution," in *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2011.

[14] M. Ollivier, S. Bardin, R. Bonichon, and J.-Y. Marion, "How to Kill Symbolic Deobfuscation for Free (or: Unleashing the Potential of Path-oriented Protections)," in *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[15] M. Schloegel, T. Blazytko, M. Contag, C. Aschermann, J. Basler, T. Holz, and A. Abbasi, "Loki: Hardening Code Obfuscation against Automated Attacks," in *USENIX Security Symposium*, 2022.

[16] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. r. Pretschner, "Code Obfuscation against Symbolic Execution Attacks," in *Annual Computer Security Applications Conference (ACSAC)*, 2016.

[17] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, "Helping Johnny to Analyze Malware: A Usability-optimized Decompiler and Malware Analysis User Study," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2016.

[18] M. Botacin, L. Galante, P. de Geus, and A. Grégio, "RevEngE is a Dish Served Cold: Debug-oriented Malware Decompilation and Reassembly," in *Reversing and Offensive-oriented Trends Symposium*, 2019.

[19] D. Votipka, M. N. Punzalan, S. M. Rabin, Y. Tausczik, and M. L. Mazurek, "An investigation of online reverse engineering community discussions in the context of ghidra," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2021, pp. 1–20.

[20] Z. L. Basque, S. Doria, A. Soneji, W. Gibbs, A. Doupé, Y. Shoshitaishvili, E. Losiouk, R. Wang, and S. Aonzo, "Decompiling the synergy: An empirical study of human–llm teaming in software reverse engineering," in *The Proceedings of the Network and Distributed System Security Symposium (NDSS 26)*, 2026.

[21] D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, "An observational investigation of reverse {Engineers'} processes," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1875–1892.

[22] E.-M. C. Behner, S. Enders, and E. Padilla, "Sok: No goto, no cry? the fairy tale of flawless control-flow structuring," in *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2025, pp. 411–431.

[23] J. Kinder, "Towards Static Analysis of Virtualization-obfuscated Binaries," in *Working Conference on Reverse Engineering (WCRE)*. IEEE, 2012.

[24] S. Enders, E.-M. C. Behner, N. Bergmann, M. Rybalka, E. Padilla, E. X. Hui, H. Low, and N. Sim, "dewolf: Improving decompilation by leveraging user surveys," in *Proceedings 2023 Workshop on Binary Analysis Research*, ser. BAR 2023. Internet Society, 2023. [Online]. Available: http://dx.doi.org/10.14722/bar.2023.23001

[25] VMProtect Software, "VMProtect Software," 2024, https://vmpsoft.com.

[26] Oreans Technologies, 2024, https://www.oreans.com/Themida.php.

[27] SEFCOM, "Pushan," 2026. [Online]. Available: https://github.com/sefcom/pushan

[28] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of Virtualization-obfuscated Software: a Semantics-based Approach," in *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[29] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson, "Information Hiding in Software with Mixed Boolean-Arithmetic Transforms," in *International Workshop on Information Security Applications*. Springer, 2007.

[30] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[31] ——, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," in *ACM Symposium on Principles of Programming Languages (POPL)*, 1998.

[32] R. B. Dewar, "Indirect Threaded Code," *Communications of the ACM*, vol. 18, no. 6, pp. 330–331, 1975.

[33] I. Piumarta and F. Riccardi, "Optimizing Direct Threaded Code by Selective Inlining," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1998.

[34] H. Xu, Y. Zhou, Y. Kang, F. Tu, and M. Lyu, "Manufacturing Resilient Bi-Opaque Predicates against Symbolic Execution," in *Conference on Dependable Systems and Networks (DSN)*, 2018.

[35] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic Reverse Engineering of Malware Emulators," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2009.

[36] N. Zhang, D. Xu, J. Ming, J. Xu, and Q. Yu, "Inspecting Virtual Machine Diversification Inside Virtualization Obfuscation," in *IEEE Symposium on Security and Privacy (S&P)*, 2025.

[37] P. Kochberger, S. Schrittwieser, S. Schweighofer, P. Kieseberg, and E. Weippl, "Sok: Automatic Deobfuscation of Virtualization-protected Applications," in *International Conference on Availability, Reliability and Security (ARES)*, 2021.

[38] B. Liu, J. Shen, J. Ming, Q. Zheng, J. Li, and D. Xu, "MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation," in *USENIX Security Symposium*, 2021.

[39] B. Reichenwallner and P. Meerwald-Stadler, "Efficient Deobfuscation of Linear Mixed Boolean-Arithmetic Expressions," in *ACM Workshop on Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks*, 2022.

[40] ——, "Simplification of General Mixed Boolean-Arithmetic Expressions: GAMBA," in *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2023.

[41] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[42] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale Malware Indexing using Function-Call Graphs," in *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[43] Z. L. Basque, A. P. Bajaj, W. Gibbs, J. O'Kain, D. Miao, T. Bao, A. Doupé, Y. Shoshitaishvili, and R. Wang, "Ahoy SAILR! There is no Need to DREAM of C: A Compiler-aware Structuring Algorithm for Binary Decompilation," in *USENIX Security Symposium*, 2024.

[44] S. Eschweiler, K. Yakdan, E. Gerhards-Padilla *et al.*, "Discovre: Efficient Cross-architecture Identification of Bugs in Binary Code," in *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[45] VirusTotal, 2026, https://www.virustotal.com/gui/file/47ae162c83cfc2302f795774aa19fb5090817362781643224157e285244eb295.

[46] Noelo Lab, "DecBench: A benchmarking suite for evaluating perfect decompilation," 2026. [Online]. Available: https://github.com/Noelo-Lab/decbench

[47] M. Liang, Z. Li, Q. Zeng, and Z. Fang, "Deobfuscation of Virtualization-obfuscated Code through Symbolic Execution and Compilation Optimization," in *International Conference on Information and Communications Security (ICICS)*. Springer, 2018.

[48] Z. Tang, K. Kuang, L. Wang, C. Xue, X. Gong, X. Chen, D. Fang, J. Liu, and Z. Wang, "Seead: A Semantic-based Approach for Automatic Binary Code De-obfuscation," in *IEEE Trustcom/BigDataSE/ICESS*. IEEE, 2017.

[49] N. Eyrolles, L. Goubin, and M. Videau, "Defeating MBA-based Obfuscation," in *International Workshop on Software PROtection (SPRO)*, 2016.

[50] F. Biondi, S. Josse, A. Legay, and T. Sirvent, "Effectiveness of Synthesis in Concolic Deobfuscation," *Computers & Security*, vol. 70, pp. 500–515, 2017.

[51] R. David, L. Coniglio, and M. Ceccato, "Qsynth–A Program Synthesis based Approach for Binary Code Deobfuscation," in *Workshop on Binary Analysis Research (BAR)*, 2020.

[52] J. Lee and W. Lee, "Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting," in *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[53] W. Feng, B. Liu, D. Xu, Q. Zheng, and Y. Xu, "Neureduce: Reducing Mixed Boolean-Arithmetic Expressions by Recurrent Neural Network," in *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020.

[54] A. Guinet, N. Eyrolles, and M. Videau, "Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions," in *GreHack*, 2016.

[55] VirusTotal, "VirusTotal File Report for PCHunter," 2026, https://www.virustotal.com/gui/file/c82ecb56a7ebd461c99e9c292861105eb7c49c88dac5d3ea853035654be88a62.

[56] ——, "VirusTotal File Report for MultiKey," 2026, https://www.virustotal.com/gui/file/0c7806ff710d34d12075d8d0b728a58723fbcb3b3f9ef94ef5a247eeed94f0d6.

[57] ——, "VirusTotal File Report for CNICDriver," 2026, https://www.virustotal.com/gui/file/b97fee512e8f5611aa23a86bbbad3844556ddfe191c331e93cccef2603825d5e.

APPENDIX

A. Ablation Study

To evaluate the contribution of PUSHAN's main components, we ran four ablation experiments on two representative protected binaries from the main evaluation: *zlib*/`gz_read` protected by VMProtect and *sdl*/`ClosePhysicalCamera` protected by Themida. Full PUSHAN recovers 100% CFG similarity for both cases (Table III). All ablations use the same binaries, entry points, and an 18-hour time limit. These ablations measure the impact of the PUSHAN-specific components that extend prior VPC-sensitive analysis toward scalable CFG recovery.

First, in the no-symbolizer ablation, we disable the iterative symbolization phase; this tests whether CFG recovery remains complete without symbolization-guided refinement. Second, in the path-constraints ablation, we replace PUSHAN's constraint-free CFG exploration with standard symbolic execution over accumulated path constraints. Third, in the no-false-branch-pruning ablation, we disable pruning of infeasible VM branches, preserving false successors during CFG recovery. Fourth, in the no-simplification ablation, we disable the semantic-preserving simplification stage to measure whether the recovered CFG remains tractable. Table VI summarizes the ablation results. Disabling symbolization can make the analysis finish quickly, but it produces incomplete recovered structure. Both the path-constraints and no-false-branch-pruning ablations time out while still attempting CFG recovery, before reaching downstream simplification or decompilation. The path-constraints ablation becomes infeasible due to path explosion, while the no-false-branch-pruning ablation suffers substantial graph blow-up from retained infeasible VM-introduced branches. The no-simplification ablation reaches the 80 GB memory cap on both samples, showing that simplification is also necessary for keeping the recovered representation tractable. These ablations support that PUSHAN's scalability and graph quality depend jointly on symbolization, constraint-free CFG exploration, pruning of semantically false VM branches, and semantic-preserving simplifications.

B. Obfuscation Configurations

Table VII summarizes the VM setups for the samples that we used in our evaluation.

C. Additional VMProtect Samples

In addition to the real-world VMProtect case study discussed in Section VIII-F, we also ran PUSHAN on three more wild binaries that include VM-obfuscated code by VMProtect: *PCHunter* (SHA-256 prefix `c82ecb56`), *MultiKey* (prefix `0c7806ff`), and *CNICDriver* (prefix `b97fee51`) [55]–[57]. PCHunter is a user-space program while the other two are kernel-mode drivers. We ran PUSHAN on each of these samples with the same configuration as the real-world VMProtect case study and successfully deobfuscated and decompiled at least one function from each sample. Based on the decompiled output and manual reversing of the original binaries, we did not find anything in PCHunter or the MultiKey driver that would indicate any malicious behaviors. The deobfuscated function of PCHunter (starting at 0x140881960, see Listing 2) is a file probing function: It uses `CreateFileW` to detect if a file (or in this case, the PCHunter driver exists and is available for opening). The deobfuscated function of MultiKey implements some cryptographic algorithm.

CNICDriver is more interesting. An AI agent (Claude Code) with Opus 5 analyzed the binary and concluded that it "is malicious" based on the APIs it imports and the features it implements, without deobfuscating the VMProtect-obfuscated code. We deobfuscated and decompiled all obfuscated functions in CNICDriver, which made us believe that the driver does not implement any malicious behaviors.

It is worth noting that according to VirusTotal, out of 70 antivirus engines, more have flagged PCHunter (41) and the MultiKey driver (27) as malicious than the CNICDriver (13). However, the threat names that antivirus engines report are generic ("Unsafe" or "PossibleThreat"), about the obfuscator ("VMProtect"), or simply flagging the software itself as malicious ("RiskWare.PcHunter" or "Trojan.MultiKey"). This suggests that antivirus companies may not have inspected the actual logic or behaviors that are under the protection of VM-obfuscation in these binaries, and PUSHAN can help security analysts inspect their actual behaviors.

```
void VM_1(void)
{
    unsigned long long v6;  // 5216
    unsigned long v7;  // 5358
    unsigned long v8;  // rbx
    unsigned long v9;  // rdi
    unsigned long v0;  // [bp-0x238]
    unsigned long v1;  // [bp-0xd8]
    unsigned long v2;  // [bp-0xd0]
    unsigned long v4;  // [bp+0x10]
    unsigned long v5;  // [bp+0x18]

    v1 = &g_140881989;
    v0 = CreateFileW(&g_14074b090, 0x80000000, FILE_SHARE_NONE, NULL,
      ↪ OPEN_EXISTING, SECURITY_ANONYMOUS, NULL);
    v6 = (38505 >> (((char)v0 + 1 ^ (char)v0 + 1 >> 4) & 15) & 1) * 4 | (v0 ==
      ↪ 0xffffffffffffffff) * 64 | (v0 + 1 < 0) * 128;
    v7 = *((long long *)(((v6 & 64) >> 3) + (char *)&v1));
    if (!(((*((int *)(v7 - 4)) ^ (unsigned int)v7) >> 1 ^ 2006124373) - 1) == 0
      ↪ xffffffffffb310d)
    {
        v2 = v8;
        v4 = v8;
        v2 = v9;
        v5 = v9;
        CloseHandle(v0);
    }
    return;
}
```

Listing 2: The deobfuscated and decompiled function in the PCHunter sample, starting at 0x140881960.

D. Timing and Simplification Examples

Figure 8 shows the time taken by different stages of PUSHAN for the VMProtect, VMProtect Ultra, and Themida configurations.

E. CTF Challenge

```
cmp = 1885566054 - input;  // "pctf"
if (!cmp) return;
// ... omitted code ...
cmp = 2071358815 - input;  // "vm_{"
if (!cmp) return;
// ... omitted code ...
cmp = 1915975269 - input;  // "r3ve"
if (!cmp) return;
// ... omitted code ...
cmp = 1920152425 - input;  // "rs3i"
if (!cmp) return;
// ... omitted code ...
cmp = 1850171185 - input;  // "nG_1"
```

TABLE VI: Ablation study on representative commercial-obfuscator samples.

Ablation	***zlib*/gz_read (VMProtect)**	***sdl*/ClosePhysicalCamera (Themida)**
No symbolizer	• Similarity: 7%; $\Delta N = -29$, $\Delta E = -44$	• Similarity: 19%; $\Delta N = -14$, $\Delta E = -20$
Path constraints	• Time out ($>$ 18 hr)	• Time out ($>$ 18 hr)
No false-branch pruning	• Time out ($>$ 18 hr)	• Time out ($>$ 18 hr)
No simplification	• Memory cap (80 GB)	• Memory cap (80 GB)

TABLE VII: VM setups, sample counts, automation boundaries, and validation. The function address is the default manual input; counts follow each row's evaluation unit.

VM setup and protections	**Samples**	**Automatic VPC ID**	**Extra manual input**	**Validation**
VMProtect virtualization (only virtualization enabled)	14	✓	–	CFG similarity
VMProtect Ultra (mutation plus virtualization)	14	✓	–	CFG similarity
VMProtect virtualization with packing (output packing enabled)	2	✓	Unpacked memory dump	CFG similarity
Themida Fish VM (only virtualization enabled)	14	✓	–	CFG similarity
Themida Tiger VM (only virtualization enabled)	2	✓	–	CFG similarity
Themida Dolphin VM (only virtualization enabled)	2	✓	–	CFG similarity
Tigress VM-1 (stack VM, switch-case dispatch)	334	✓	–	CFG similarity + I/O testing
Tigress VM-2 (opaque predicates, duplicate opcodes, direct dispatch, super operators)	333	✓	–	CFG similarity + I/O testing
Tigress VM-3 (bogus functions, implicit flows, opaque predicates)	333	✓	–	CFG similarity + I/O testing
CTF custom VMs (bespoke challenge VMs)	5	✓	–	API trace + writeup checks (no GED)
Real-world VMProtect sample (vendor options unknown)	1	✓	–	Qualitative case study

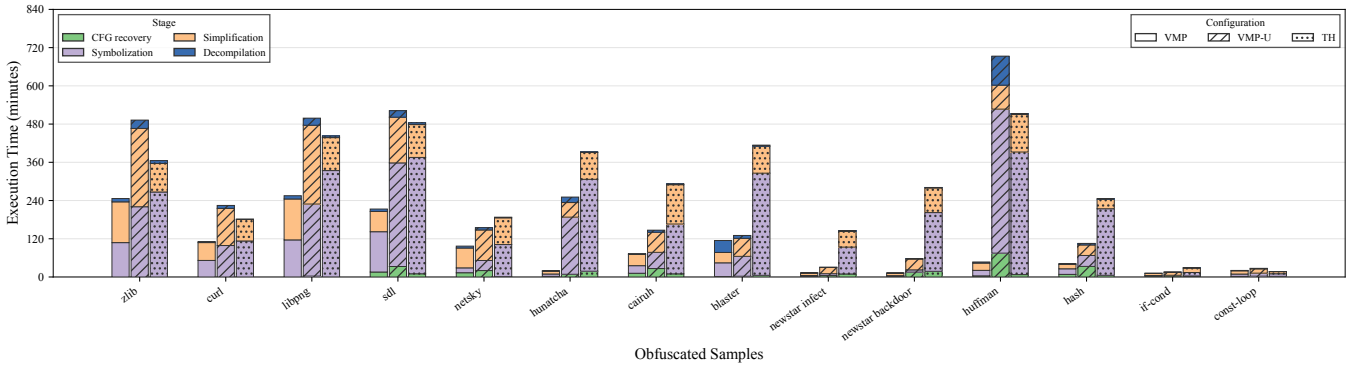


Fig. 8: Time taken by different stages of PUSHAN for the VMProtect, VMProtect Ultra, and Themida configurations.

```
t29 = Add64(t12,0xfffffffffffffe80)
STle(t29) = t27
...
t31 = Add64(t23,0xfffffffffffffe80)
t33 = LDle:I64(t31)
```

(a) Redundant stack-based data movement.

```
t29 = Add64(t12,0xfffffffffffffe80)
STle(t29) = t27
...
t31 = Add64(t23,0xfffffffffffffe80)
t33 = t27
```

(b) After eliminating redundant stack-based data movement.

Fig. 9: Examples of semantics-preserving simplifications.

```
if (!cmp) return;
// ... omitted code ...
cmp = 1935635570 - input;  // "s_tr"
if (!cmp) return;
// ... omitted code ...
cmp = 828599161 - input;   // "lcky"
if (!cmp) return;
// ... omitted code ...
cmp = 2100310064 - input;  // "}000"
if (!cmp) return;
```

Listing 3: Decompiled flag checks from a custom-VM CTF challenge. Each constant encodes a four-character flag segment.

F. PUSHAN *Deobfuscation Output of The Real-world Sample*

```
extern char g_7ff74db5f1e5;
extern char g_7ff74db5f705;
extern char g_7ff74db5ff1a;

long VM_1(unsigned long a0, unsigned int a1)
{
    .
    .
    .
    .

    v44 = 0;
    v43 = v25;
    v7 = *((int *)&v43);
    v8 = 0;
    v42 = 8 + v9;
    v19 = v42;
    v15 = v19;
```

```
    v24 = 0x7ff60da70000;
    v6 = tmp_1035;
    v5 = v9;
    v9 = v19;
    v19 = 17;
    v2 = *((long long *)&v7);
    v23 = v10;
    v7 = v15;
    v17 = tmp_1070;
    v10 = v22;
    v47 = 435308053;
    .
    .
    .
    .
    v19 = v36;
    v36 = v24 + 68;
    v13 = v36;
    v14 = 0;
    memmove(v19, v18, *((long long *)&v13));
    v13 = v20;
    v19 = v18;
    v18 = 0;
    v6 = 144 + v13;
    v91 = 1;
    v92 = (v18 >> 21 == 1 ? 1 : 0);
    v93 = (v18 >> 18 == 1 ? 1 : 0);
    memmove(v6, v19, 68);
    v5 = 0;
    v40 = v13;
    v33 = v5;
    v18 = 9;
    do
    {
        v50 = 4 | v91 & 0x400 | v92 * 0x200000 & 0x200000 | v93 * 0x40000 & 0x40000
          ↪ ;
        v35 = v18;
        v34 = v33;
        v91 = (v34 >> 10 == 1 ? -1 : 1);
        v92 = (v34 >> 21 == 1 ? 1 : 0);
        v93 = (v34 >> 18 == 1 ? 1 : 0);
        v94 = prng(v40);
        v21 = 0;
        v12 = v94;
        v18 = v40;
        v19 = 0;
        v20 = -1 + v35;
        v24 = (1 & v21) + (-2 & v19);
        v21 = ~(~(v24) & -17) & ~(16 & ~(~(v24) & ~((1 & v21) + (-2 & v19))));
        v50 = *((long long *)((char *)&<0x4e6f6e6531343037303031333639333939393230[
          ↪ is_189]|Stack bp-16, 1 B> + ((~(v21) & 64) >> 3)));
        v24 = v50;
        v49 = v50;
        v49 = ~(*((long long *)&v49));
        v50 = *((int *)((char *)&v49 + 4)) & (int)v49;
        v49 = 247274402;
        v49 = ~(*((long long *)&v49));
        v50 = *((int *)((char *)&v49 + 4)) & -247274403;
        v49 = -247274403;
        v48 = v24;
        v48 = ~(*((long long *)&v48));
        v49 = 247274402 & (int)v48;
        v49 = ~(*((long long *)&v49));
        v50 = *((int *)((char *)&v49 + 4)) & (int)v49;
        v13 = v91 & 0x400 | v92 * 0x200000 & 0x200000 | v93 * 0x40000 & 0x40000;
        v33 = v21;
        v32 = v13;
        v27 = v50;
        v95 = (int)v27 + 0x100000000;
        v96 = *((char *)((int)v27 + 140699062697985));
        .
        .
        .
        .

    } while ((((((((&g_7ff74db5ff1a)[__ROR__(__ROL__(*((char *)((int)v27 +
      ↪ 140699062698017)) - ((char)v0 - 133 - (char)v102), 5) - 33, 4)] & ~0
      ↪ xff00ff00ff0100) >> 8 | (&g_7ff74db5ff1a)[__ROR__(__ROL__(*((char *)((int)
      ↪ v27 + 140699062698017)) - ((char)v0 - 133 - (char)v102), 5) - 33, 4)] << 8
      ↪  & ~0xff00ff00ff0100) & ~0xffff00010000) >> 16 | 0x400e000000000000) & ~0
      ↪ x100000000) >> 32 | 0x100000000) + 0x7ff60da70000 == 140700137420549);
    v50 = 0;
    *((unsigned long *)&v50[88]) = v12;
    return (long long)v50[88];
}
```

Listing 4: PUSHAN's deobfuscated output (truncated) for the real-world sample [45].

G. Claude's Simplified Output of the Real-world Sample

```
#include <string.h>
#include <stdint.h>

// External PRNG function (provided elsewhere in the binary)
extern uint64_t prng(uint64_t state);

long VM_1(unsigned long a0, unsigned int a1) {
    uint8_t *buf = (uint8_t *)a0;
    uint32_t count = a1;

    // Phase 1: Fill buffer with multiplicative hash sequence
    uint32_t *out = (uint32_t *)buf;
    uint32_t val = count;
    for (int i = 0; i < 17; i++) {
        val = 1403630842u * val - 2;
        // The NOT/AND pattern collapses to: val = val (with potential truncation)
        // effectively: val = (uint32_t)(~(uint64_t)val) & high32 => just keeps val
        val = val; // obfuscated identity after simplification
        *out++ = val;
    }

    // Phase 2: memmove 68 bytes within the buffer
    memmove(buf + 76, buf, 68);
    memmove(buf + 144, buf, 68);

    // Phase 3: PRNG loop
    uint64_t state = (uint64_t)(uintptr_t)buf;
    uint64_t result = 0;
    for (int i = 9; i > 0; i--) {
        result = prng(state);
    }

    // Store and return final PRNG result
    *(uint64_t *)(buf + 88) = result;
    return (long)result;
}
```

Listing 5: Claude's simplified version of PUSHAN's output for the real-world sample.

H. The Extended CFGED Algorithm for Similarity Measurement

Algorithm 1: Extended CFGED

Input: deobfuscated CFG G_d and original CFG G_o
Output: normalized similarity σ

$(G_d, G_o) \leftarrow \textsc{Normalize}(G_d, G_o)$;
$F \leftarrow \textsc{GetBlockFeatures}(G_d, G_o)$;
$M \leftarrow \emptyset$;
foreach $u \in V_d$ *containing a named call* **do**
  $C \leftarrow \{v \in V_o \mid \text{CallNames}(u) = \text{CallNames}(v)\}$;
  $v \leftarrow \textsc{ResolveByDegreeThenDepth}(u, C)$;
  if $v \neq \perp$ **then** $M[u] \leftarrow v$;
$M[\text{entry}(G_d)] \leftarrow \text{entry}(G_o)$;
if $|\text{exits}(G_d)| = |\text{exits}(G_o)| = 1$ **then**
  $M[\text{only}(\text{exits}(G_d))] \leftarrow \text{only}(\text{exits}(G_o))$;
foreach $u \in V_d \setminus \text{dom}(M)$ **do**
  $C \leftarrow V_o \setminus \text{ran}(M)$;
  if $C = \emptyset$ **then break**;
  $s \leftarrow \max_{x \in C} \textsc{DiscovRE}(F(u), F(x))$;
  $B \leftarrow \{x \in C \mid \textsc{DiscovRE}(F(u), F(x)) = s\}$;
  $v \leftarrow \arg\min_{x \in B} |\text{depth}_{G_d}(u) - \text{depth}_{G_o}(x)|$;
  if $s \geq 0.6$ **then** $M[u] \leftarrow v$;
changed $\leftarrow$ true;
while *changed* **do**
  changed $\leftarrow$ false;
  foreach $(u, v) \in M$ **do**
    $P_d \leftarrow \text{pred}_{G_d}(u) \setminus \text{dom}(M)$;
    $P_o \leftarrow \text{pred}_{G_o}(v) \setminus \text{ran}(M)$;
    if $|P_d| = |P_o| = 1$ **then**
      $M[\text{only}(P_d)] \leftarrow \text{only}(P_o)$;
      changed $\leftarrow$ true; **break**;
    $S_d \leftarrow \text{succ}_{G_d}(u) \setminus \text{dom}(M)$;
    $S_o \leftarrow \text{succ}_{G_o}(v) \setminus \text{ran}(M)$;
    if $|S_d| = |S_o| = 1$ **then**
      $M[\text{only}(S_d)] \leftarrow \text{only}(S_o)$;
      changed $\leftarrow$ true; **break**;
$d \leftarrow \text{CFGED}(G_d, G_o; M)$;
return $1 - d/(|V_d| + |E_d| + |V_o| + |E_o|)$;