DEBRA: A Real-World Benchmark For Evaluating Deobfuscation Methods

Zheyun Feng
University of New Hampshire
Durham, New Hampshire, USA
Zheyun.Feng@unh.edu

Dongpeng Xu
University of New Hampshire
Durham, New Hampshire, USA
Dongpeng.Xu@unh.edu

Abstract

Software obfuscation is a broadly adopted protection method that hides representative information by transforming it into a highly opaque but semantic-equivalent form. To date, a variety of deobfuscation methods have been developed to peel off the obfuscation and expose the original program semantics. However, nearly all deobfuscation tools are merely tested and evaluated on small toy programs with ad-hoc configurations, leading to a fundamental gap between the deobfuscation research and real-world practice. We discover the key obstacle is due to the absence of a real-world, large-scale testing benchmark that can systematically evaluate the deobfuscation methods.

To fill this gap, we propose DEBRA, a comprehensive, large-scale obfuscation benchmark crafted with a diverse range of real-world programs for evaluating deobfuscation methods. First, we collect a set of real-world open-source programs representing diverse obfuscation scenarios. Second, we design a metric-driven approach to determine the crucial or sensitive functions to be obfuscated, because in real-world practice, only the critical parts of a program are obfuscated to balance security and execution overhead. Instead of blindly and arbitrarily obfuscating a program, this design makes our obfuscation benchmark closely mirror the real-world practice. Next, we obfuscate the selected areas in these programs with state-of-the-art obfuscators and obfuscation techniques, resulting in DEBRA. During our evaluation, the samples from DEBRA crashed the target deobfuscators and exposed limitations that were not shown during their original evaluation. With the hope of driving advancements in deobfuscation research, DEBRA serves as a pioneering standard benchmark for evaluating and comparing different deobfuscation methods.

ACM Reference Format:
Zheyun Feng and Dongpeng Xu. 2025. DEBRA: A Real-World Benchmark For Evaluating Deobfuscation Methods. In *Proceedings of the 2025 Workshop on Software Understanding and Reverse Engineering (SURE '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3733822.3764674

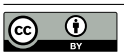
1 Introduction

Over the last decade, software obfuscation has seen a surge in adoption across both benign and malicious domains. Commercial software developers increasingly rely on obfuscation to safeguard intellectual properties and proprietary algorithms [23, 26], specifically targeting functions such as licensing checks, DRM enforcement, password validation and key generation. On the other hand, malware authors exploit it to elude security detection and analysis techniques [21, 27, 34, 45]. Software obfuscation transforms the original program into a highly intricate, yet functionally equivalent form, thereby camouflaging the underlying intent. The intricacy introduced by various obfuscation methods poses formidable hurdles for security practitioners in program analysis and reverse engineering.

Recognizing the urgent need to address this issue, numerous deobfuscation methods have been proposed and developed to unravel the intricate layers of obfuscated code [10, 22, 25, 35, 41, 43]. Unfortunately, these automatic deobfuscation methods have seen limited evaluation in real-world scenarios [6, 20]. One notable challenge is the absence of an open standard benchmark that accurately reflects the complexity of real-world situations for evaluating deobfuscation methods. Existing evaluations are often conducted on toy programs with simplistic structures, a limited selection of samples sharing similar functionalities, or ad hoc settings without enough reproducible details, making it difficult to validate and compare the efficacy of different deobfuscation methods comprehensively and impartially. The deficiency in public, standard, and practical benchmarking has long hindered the advancement of effective countermeasures against obfuscated malware.

In an effort to bridge the gap, we establish DEBRA, **D**eobfuscation **E**valuation **B**enchmark for **R**esearch and **A**nalysis. DEBRA comprises 1,917 programs derived from obfuscating 224 function-level targets selected by our metric-driven approach with three state-of-the-art obfuscators and four prevalent obfuscation transformations under tens of parametric configurations across 15 real-world, open-source, and complex programs. Our metric-driven approach based on calculating the sensitivity and centrality of a function identifies key functions within the codebase of a program, simulating the obfuscation strategies in real-world scenarios. This real-world benchmark accurately reflects the situations in practical deobfuscation tasks.

In our evaluation, we use DEBRA to re-assess the deobfuscation method proposed in [31] and Xyntia [24]. The results turned out that both deobfuscators had limited capacity to process the real-world samples from DEBRA. Furthermore, we investigate the root causes of those failures and attribute them to the overfitting of both deobfuscation methods to overly simplistic and synthetic benchmarks. While they demonstrated promising results in their own evaluation, DEBRA exposes their limitations in handling real-world programs. In doing so, DEBRA not only facilitates the refinement of existing deobfuscation methods but also supports the development of more robust and adaptive cybersecurity strategies.



This work is licensed under a Creative Commons Attribution 4.0 International License.
SURE '25, Taipei, Taiwan
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1910-3/25/10
https://doi.org/10.1145/3733822.3764674

In a nutshell, our paper delivers three pivotal contributions as follows.

- We build a real-world benchmark to facilitate comprehensive and reproducible deobfuscation method evaluations in practical settings. We discern and highlight the gap in current deobfuscation method evaluations and practices, and use real-world programs and prevalent obfuscation transformations to establish DEBRA, serving as the new standard for deobfuscation method evaluations.
- We design a generic and metric-driven approach to identify key functions that are prime candidates for obfuscation within a program's codebase. This approach scores functions based on two core dimensions: sensitivity and centrality.
- We prove the efficacy of our benchmark by re-assessing the performance of two advanced deobfuscation methods. The process unveiled their limitations that were not reported before.

We organize the paper as follows: Section 2 provides a summary of the limitations we discerned in current deobfuscation method evaluations and Section 3 describes our approach to address these limitations. We elaborate the process of constructing DEBRA in Section 4. An in-depth reassessment of two representative deobfuscators using DEBRA is presented in Section 5. We discuss the future plan and pertinent works in Section 6 and Section 7 respectively. Finally, we conclude the paper in Section 8.

2 Background and Motivation

In this section, we first delineate the general concepts of software obfuscation and deobfuscation. Next, we point out the limitations buried in deobfuscation method evaluations and motivate the need to develop a real-world benchmark to address the constraints.

2.1 Software Obfuscation and Deobfuscation

Software obfuscation is an act of transformation that preserves the semantics of a program while altering its syntactic form to impede direct understanding and analysis. This technique has been extensively utilized in the protection of digital assets against piracy and malicious reverse engineering across a variety of programming languages and platforms.

Banescu et al. [9] published a comprehensive review of the taxonomies of software obfuscation techniques. Obfuscation can target different abstraction levels: source code level [4], intermediate representation level [17], and binary machine code level [1] as well as different granularity levels: instruction, basic block, loop, function, program, and system. Obfuscation is allowed to be applied across multiple levels based on the implementation decisions by the developers. For instance, Control Flow Flattening [38] can be employed both at the source code level [4] and the intermediate representation level [17] to flatten a program's natural control flow into a more complex, less predictable form.

Software deobfuscation is the process of reversing obfuscated applications, transforming an obscured part back into a form that is more approachable to understand and analyze. Deobfuscation techniques are continually evolving to counteract the abuse of obfuscation especially in malware and viruses, which often employ such tactics to camouflage their malicious purposes and evade detection. Deobfuscation is often considered more difficult than obfuscation because of the asymmetric nature between the two tasks.

As obfuscation techniques continue to evolve and become more sophisticated, substantial efforts have been put into the development of potent deobfuscation strategies to unveil the authentic semantics of an obfuscated program. Strategies such as pattern matching [12], symbolic execution [7], and program synthesis [10] are widely adopted in deobfuscation research. Among these, symbolic execution and program synthesis have emerged as the state-of-the-art techniques. Symbolic execution systematically explores program paths and simulates program execution by treating concrete program inputs as symbolic ones. It excels in discarding bogus and infeasible branches introduced by obfuscation techniques such as opaque predicates but falls short when the number of possible execution paths becomes huge, leading to path explosion, or when the program interacts heavily with system APIs or the network, resulting in constraint-solving difficulties. By contrast, program synthesis examines input-output samples and generates a simpler but semantically equivalent version of the obfuscated program. It is obfuscation-agnostic and less hindered by syntactic complexities. Its strength lies in reconstructing semantically equivalent expressions or functions that are hard to analyze symbolically. For example, a synthesizer examines the input/output samples of the expression $X + Y + 1 + (\sim X \mid \sim Y)$ and the expression $X \mid Y$ and finds they are equivalent, thus it considers $X \mid Y$ as the simplified result of the former. However, when the search space of possible candidates becomes very large, synthesis may time out or yield imprecise results.

Meanwhile, obfuscation detection is a critical precursor of deobfuscation, focusing on the identification of obfuscated segments within an application. It typically includes the steps of scanning, analyzing, and reporting. However, obfuscation detection has often been neglected as Xu et al. [41] pointed out that many analyses operate under the presumption that the scope of obfuscated code is known, thereby bypassing the crucial step of detection. A handful of works recognize the unrealistic assumption and place the pinpointing of obfuscation areas as an initial step for their deobfuscation tools. VMHunt [41] identifies the boundaries of virtualization-obfuscated code from an execution trace before initiating its simplification process. IDA Pro plugin gooMBA [2] locates before simplifying potential Mixed Boolean-Arithmetic (MBA) expressions at the intermediate representation when inputting a program to IDA Pro.

2.2 Motivation

The development of deobfuscation methods is inherently guided by the specific types of obfuscation they are designed to counteract. A common strategy for evaluating a deobfuscation method starts with a selection of previously used sample sets by peers or custom-created sample sets depending on the focus of the deobfuscation method (e.g., malware, binaries), which are subsequently obfuscated with the obfuscators that implement the pertinent type(s) of obfuscation at the authors' choices, thereby fabricating a controlled environment to test their deobfuscation method. A comparative

analysis is conducted afterward between the vanilla and the deobfuscated version of the sample sets to showcase the capability of the deobfuscation method which can be further quantified based on experimental results. We carefully study the evaluation patterns of existing deobfuscation methods and find that they seriously suffer from the following limitations, which form the major motivation of this work.

Toy Programs. Nearly all existing deobfuscation methods are evaluated on toy programs. In this context, "toy programs" refer to tiny programs with simple structures and functions that are not used in real-world projects. Some common examples are implementations of algorithms from college assignments such as bubble sort and binary search, simple cryptographic algorithms, as well as synthesized programs produced by automated generators such as Csmith [44] and USmith [28]. Tofighi-Shirazi et al. [36] evaluated their deobfuscation approach partially with toy programs sourced from GitHub. Similarly, Blazytko et al. [10] employed custom-built toy programs to test the efficacy of their method. In particular, a recent study [19] reported that toy programs are used in a strikingly high number of evaluations for deobfuscation works. An online repository, Obfuscation-Benchmarks [3] proposed by Banescu et al. [8] steadily increases appearance in the evaluation of deobfuscation works. Nonetheless, this benchmark predominantly incorporates simple toy programs which often only have one main function.

The heavy reliance on toy programs poses a severe threat to the validity of evaluation results, omitting feedback on the performance of deobfuscation methods under complex scenarios encountered in practical environments. Even though the evaluation results look promising on toy programs, due to significantly higher complexities, the same level of deobfuscation effectiveness cannot be reproduced and guaranteed on real-world programs.

Arbitrary Obfuscation Region. In practice, to avoid unnecessary performance overhead, obfuscations are only applied to the sensitive or critical parts of a target program. Thus modern obfuscators are equipped with regional obfuscation feature, i.e., an explicit specification of where and what the obfuscation is. For example, Tigress necessitates the function name(s) as a required parameter for execution, Code Virtualizer expects marked code regions with built-in macros when obfuscating an ELF executable.

Nevertheless, we observe that *random selection* and *whole program obfuscation* are the prevalent strategies in the existing deobfuscation evaluation. The authors rarely describe the exact regions where obfuscation has been applied. For instance, DiANa [18] was evaluated with a C/C++ set in which every program was fully obfuscated by O-LLVM. LOOP [25] was tested with programs obfuscated by random insertion of opaque predicates.

Evaluations conducted on arbitrarily obfuscated targets lead to unexpected exceptions and unsatisfied results. For instance, obfuscating a driver function may be both futile and tedious as sensitive data or proprietary algorithms typically are not located in a driver function in real-world programs. Whole program obfuscation is hardly used in practice, yielding biased results when evaluating a deobfuscation method in such setting. Moreover, real-world obfuscators cannot handle all data/control structures, so obfuscation could lead to incompatible errors or exit unexpectedly when obfuscating a function with features incompatible with an obfuscator.

Table 1: A complete list view of all testbed programs.

Name	Version	Description	LoC
bash	5.2.15	Unix shell and command language	137.5K
chmod	9.4	Utility to change access permissions	36.6K
cp	9.4	Utility to copy files and directories	41.1K
curl	8.3	Toolkit for transferring data with URLs	316.4K
find	4.9	Utility to search files and directories	192.8K
gcc	11.4	Compiler system	8.6M
grep	3.11	Utility to search specific patterns	170.0K
gzip	1.13	Data compressor	92.3K
httpd	2.4.57	HTTP server	217.0K
ls	9.4	Utility to display files/directories	42.4K
nano	7.2	Text editor	85.5K
OpenSSL	3.1.3	Cryptography toolkit	702.5K
QEMU	8.11	Machine emulator and virtualizer	7.1M
SQLite	3.43	SQL database engine	200.8K
tar	1.35	File archiver	231.2K

Absence of Standards. Existing deobfuscation experiments are conducted on custom-created micro sample sets or ad hoc case studies without sufficient details documented [13, 29], which hinders the comparison evaluation with peer tools. This lack of standard undermines the completeness, validity, and reproducibility of the evaluation results [37]. Another study [19] also concludes that a lack of standardized, representative benchmarks exists in software protection research.

Our Insight. Motivated to address these limitations, we take the first step to create DEBRA, a benchmark that incorporates curated and marked obfuscation targets at function-level within real-world and open-source programs to facilitate comprehensive deobfuscation method evaluations. Our goal is to provide an environment where deobfuscation methods can be evaluated in scenarios that closely mirror real-world conditions. Researchers and developers can measure the efficacy of their methods in real-world settings, the evaluation results can be directly used to compare with peer tools to gain insights into the strengths and weaknesses of their methods.

3 Method

To address the aforementioned limitations, our design of DEBRA involves three core aspects:

(1) **Real-world Testbed.** We have carefully curated real-world open-source programs as the testbed programs. Our benchmark suite excludes toy programs to deliver realistic scenarios of obfuscating programs in practice.
(2) **Pinpoint Obfuscation.** We design a novel metric-driven method to automatically pinpoint the sensitive or critical parts of a program to be obfuscated.
(3) **Standardization.** We obfuscate selected areas with standard, reproducible obfuscation methods from state-of-the-art obfuscators. The entire obfuscation process is well documented. Moreover, DEBRA is open to the public and free to use, establishing it as a robust standard for comparing different deobfuscation techniques.

The rest of this section elaborates the design of DEBRA.

3.1 Real-world Testbed

To ensure that each program in our benchmark closely mirrors real-world scenarios, we meticulously select a range of programs based on the following criteria.

- **Unique Functionality.** Ensure a spread of unique functionality across programs is crucial for providing a broad spectrum of real-world workloads. Different applications exhibit varying performance characteristics, thereby supporting a comprehensive testbed for evaluating deobfuscation techniques.
- **Code Complexity.** Applications with a degree of code complexity are indispensable for a realistic evaluation of deobfuscation methods. Complex code structures emulate real-world scenarios, providing a robust platform for assessing the efficacy and performance of deobfuscation methods.
- **Open-source.** We focus on open-source programs to ensure transparency in implementation details, facilitating deeper analysis of deobfuscation method evaluations. At the same time, it upholds reproducibility allowing the evaluations to be both verifiable and comparable.
- **Recognition and Maintenance.** All programs selected are actively maintained and well-recognized within the software security research community. Their widespread recognition reflects a high level of trust, while active maintenance guarantees reliability and completeness.

Overall, the selection process totals a list of 15 real-world programs in Table 1. Each testbed program represents a distinct general category in software applications according to its functional purpose. Column *LoC* approximates the code complexity of each testbed program by counting the lines of code for C/C++ code in codebases with `scc` (version 3.1.0). As opposed to toy programs, which usually include less than 100 lines of code, our testbed programs contain 100,000 lines of code on average. Many of the testbed programs debuted decades ago, yet they continue to be actively maintained and remain prominent nowadays. All testbed programs are open-source programs and were publicly accessible at the time of writing this paper.

We restrict our scope to programs written in C/C++ language as Kochberger et al. [19] note that C/C++ languages are the most targeted languages in recent software security studies whereas a severe shortage of suitable benchmarks exists.

3.2 Pinpoint Obfuscation

In real-world practice, typically only a select few critical functions within a software require obfuscation to safeguard sensitive information as obfuscation notably impacts the performance [32]. To reflect this trait, we develop a novel metric-driven method to automatically pinpoint target functions to apply obfuscation.

Function-level obfuscation has gained prominence in both academic research and industrial applications. In a broad examination of obfuscation techniques, 10 out of 31 obfuscation transformations are aimed at function level, underscoring the importance of this category in the field [9]. Given the prevalence of function-level obfuscation, our benchmark is designed to focus specifically on this granularity.

```
1 static bool
2 set_owner(..., struct stat const *src_sb,
      ...)
3 {
4   uid_t uid = src_sb->st_uid;
5   gid_t gid = src_sb->st_gid;
6   ...
7 }
```

Figure 1: An example candidate for obfuscation is the *set_owner* function in the *cp* program, where the user ID and group ID are assigned directly to variable *uid* and *gid* without any safeguarding measures to conceal semantics.

However, not all functions within a program are equally susceptible or attractive to reverse engineering attacks. Functions that perform critical tasks, manipulate sensitive data, or carry proprietary assets are often the prime candidates for obfuscation, given their inner values to security risks. These considerations drive the needs for an analytical selection process to identify a subset of functions within the programs in our benchmark, optimized for a balanced evaluation of the potency, stealth, cost, and resilience of an obfuscation technique [11].

To fulfill the needs and automate the selection process, we develop a static source code analyzer that inspects the source code files, evaluating various function-level attributes against pre-defined criteria and filtering out suitable functions as obfuscation targets within testbed programs. The criteria we employ in the automated selection are based on the following two aspects.

Function Sensitivity. Sensitivity, in this context, refers to a function's possession of sensitive information that could be the target an attacker is hunting for, making it a prime candidate for obfuscation protection.

To investigate the sensitivity of a function, we examine its input and output behaviors. Functions that engage (e.g., read from or write to) with external environments (e.g., text files, databases) can be considered sensitive, given their direct impact on data confidentiality. Sensitive information (e,g., user input password, calculated results) is integrated into or detached from the data flow in a bare state without safeguarding measures during these interactions. At the same time, we look for functions that interact with sensitive variables, especially those with names like "password", "key", "id", or other similar variants that would typically imply the presence and storage of sensitive information. Codebases are often constructed rationally and developers often adopt such naming conventions to maintain code readability, making these variables easy to identify.

To quantify the sensitivity, our analyzer will specifically monitor the count of variables with sensitive naming conventions and the frequencies of I/O operations with external environment. Together, these metrics provide a solid estimate of a function's sensitivity.

For instance, the *set_owner* function is a sensitive function identified by our source code analyzer within the source file *copy.c* of the *cp* program where it deals with user ID and group ID to set the ownership of a destination file or directory during a file copy operation. As shown in Figure 1, the values of user ID and group

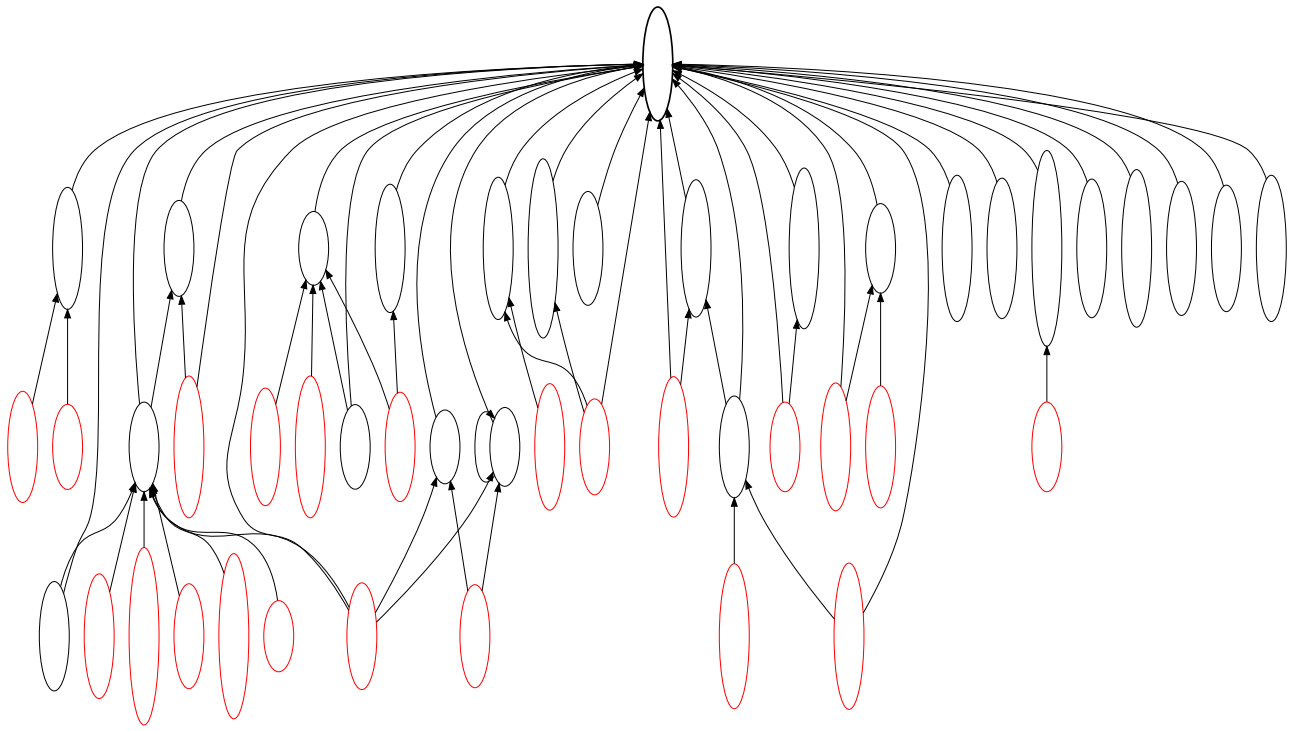


Figure 2: The call graph of the *find_next_block* function. All nodes are functions that directly or indirectly call *find_next_block*. The bold node at the top represents the *find_next_block* function, and the nodes with red borders are recursive functions. For readability, function names are omitted.

ID are directly stored in the variable *uid* and *gid* without any safeguarding measures to conceal their semantics. The bare sensitive data is exposed to an attacker who could easily tamper with the two variables to alter the ownership in an unauthorized way. The sensitive nature makes the function *set_owner* an ideal candidate for extra protection.

Function Centrality. Centrality signifies the functional importance of a function within the application's workflow. Functions with higher centrality will be more central to the logic of an application, making it a more worthwhile target for obfuscation.

More specifically, from a static perspective, we identify *critical path functions*, which are functions that sit at the crux of the application's control flow graph. These functions often serve as the decision-making points. From a dynamic perspective, we gauge the execution frequency of a function under different inputs, measuring how often a function is invoked during the application's runtime. Functions with higher execution frequencies typically play a more pivotal role in the application's operational flow and can be considered as centrality functions.

In assessing a function's centrality, we compute the function's invocation dynamics (e.g., how many times it calls other functions and how many times it is called by other functions), and the execution frequency of a function across various runs. Leveraging ChatGPT's exceptional generation power [40], execution instances are generated with prompts similar to "*Can you provide me with diverse and practical usages of the program ls? Please include a variety of examples covering different flags, patterns and scenarios.*" and manually verified thereafter. Our analyzer will analyze the two metrics, providing a comprehensive quantifiable basis for assessing a function's centrality.

Figure 2 shows an example from the source file *buffer.c* of the *tar* program. The *find_next_block* function is a central function identified by our source code analyzer. It plays an important role in deciding and managing the subsequent data block to be read from or written to an archive. Its centrality is underscored by the substantial amount of functions that are dependent on its output. Any malicious modifications of its underlying logic would crash the entire program in the worst scenario. Therefore, the pivotal nature makes the *find_next_block* function a reasonable target for obfuscation.

Nevertheless, our approach may fail to capture functions whose sensitivity is only evident through code reasoning. For example, functions that implicitly handle encryption keys without "sensitive" variables declared and used could be overlooked. Conversely, our approach may surface functions that are not realistic protection targets. For instance, utility functions that frequently perform I/O operations could be mistakenly flagged with high centrality. That said, due to the characteristics of our testbed programs, overlooked

Table 2: Obfuscators and obfuscation transformations with parametric values and ranges featured in DEBRA.

Obfuscator	Version	Obfuscation	Option	Value
Tigress	3.3.3	Virtualization	VirtualizeConditionalKinds	branch
			VirtualizeDispatch	switch, call, ifnest
			VirtualizeOperands	stack
			VirtualizeSuperOpsRatio	[0.0, 1.5]
			VirtualizeMaxMergeLength	[0, 30]
		Opaque Predicates	InitOpaqueStructs	list, array
			InitOpaqueCount	[4, 20]
			AddOpaqueKinds	true, bug, junk
			AddOpaqueCount	[4, 20]
		Control Flow Flattening	FlattenDispatch	switch
			FlattenRandomizeBlocks	true
			FlattenSplitBasicBlocks	true
			FlattenConditionalKinds	branch
Code Virtualizer	3.1.2	Virtualization	Custom Virtual Machine	Fish White, Dolphin Red, Eagle Black
Loki	None	Encode Arithmetic	Add	Depth: [1, 25]
			Subtract	Depth: [1, 25]

functions tend to be a small minority and misidentified candidates can be filtered out by human verification.

3.3 Standardization

We select an array of obfuscation methods from state-of-the-art obfuscators, as listed in Table 2 to obfuscate function targets determined by the source code analyzer. We release our benchmark for public use. To the best of our knowledge, DEBRA is the first open benchmark built upon real-world programs for deobfuscation method evaluations, this lays a standardization for deobfuscation method evaluations so that future work can have a standardized set to evaluate their works, ensuring the validity, comparability, and reproducibility of the evaluation results.

We identified a recurring set of obfuscation techniques through a review of the seminal works on the taxonomy and categorization of obfuscation techniques [9, 11, 30, 33, 42]. The recurrence of certain obfuscation techniques underscores their prevalence and significance in the field. Therefore, we use those obfuscation techniques to generate the obfuscated executables for our benchmark, ensuring alignment with real-world scenarios. It is noteworthy that certain obfuscation techniques exhibit overlapping functionalities or are essentially identical, are known by different names due to varied naming conventions. For instance, *Garbage Insertion* [9] and *Junk codes* [42] essentially refer to the same obfuscation technique aimed at thwarting attackers by adding extraneous instructions to the code. To eliminate any ambiguities, we have adhered to the nomenclature outlined in [9].

The *Obfuscation* column in Table 2 presents a full list of selected obfuscation techniques featured in DEBRA. Control Flow Flattening conceals a program's natural control flow by consolidating it into a flat dispatch structure inside a loop. Encode Arithmetic replaces simple arithmetic expressions with more complex, yet equivalent representations. Opaque Predicate introduces deceptive conditions with truth values predetermined at obfuscation to mislead static analyzers. Encode Literal transforms integer or string literals into equivalent but intricate expressions to hide their straightforward representations. Virtualization obfuscation transforms native code into bytecode, which is represented in a custom instruction set architecture. An accompanying virtual machine or emulator is tasked with interpreting and emulating the bytecode during runtime.

The *Obfuscator* and *Version* columns in Table 2 list the specifications of the obfuscators. Tigress, one of the most extensively used obfuscators in academia and industry, supports an array of obfuscation techniques with user-configurable settings. Code Virtualizer specializes in virtualization, offering a range of custom virtual machines varying in complexity, size, and speed that allow users to tailor the obfuscation strength. Both obfuscators appear at the top list of popular tools used in software protection research [19]. Loki, an academic obfuscator prototype has the ability to synthesize diverse and robust Mixed Boolean-Arithmetic expressions to harden VM handlers in virtualization obfuscation.

The *Option* and *Value* columns in Table 2 detail the configurations for each obfuscation transformation. When obfuscating *replacers* with Control Flow Flattening using Tigress, we opt for the traditional *switch* dispatch method to conceal original control flows. For the incorporation of opaque predicates into *replacers*, we tune the *AddOpaqueCount* parameter to adjust the number of opaque predicates, ranging from 4 to 20. Additionally, we set the *AddOpaqueKinds* parameter to introduce opaque predicates of types *true*, *bug*, *junk* into *replacers*' function bodies. For virtualization with Tigress, we primarily adjust the types of dispatch methods, the types of operands allowed in the ISA, the numbers of super operators, and the length of the longest sequence of instructions to be merged to increase diversities.

In the settings of Code Virtualizer for virtualization, we select three distinct virtual machines: Fish White, Dolphin Red, and Eagle Black to virtualize the target functions. Custom virtual machines

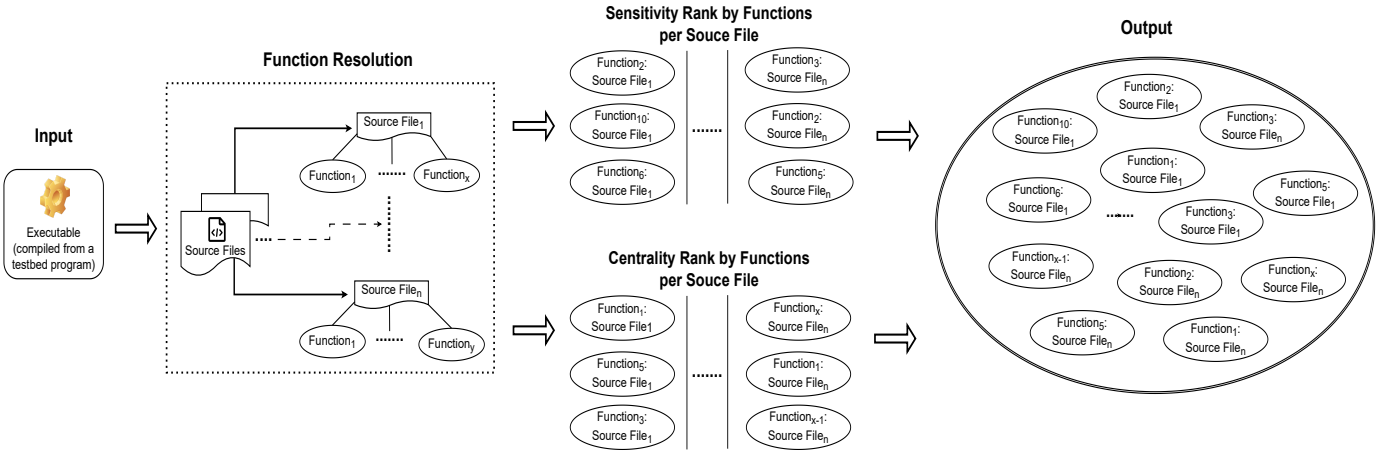


Figure 3: The workflow of the source code analyzer.

provided by Code Virtualizer are uniquely denoted by a combination of an animal and a color (white, red, and black). An animal refers to a set of VM architecture and color represents the degree of complexity. For our purpose, we choose the three VMs to cover complete complexity ranges. The Fish White VM represents the lowest complexity configuration, Dolphin Red represents the medium complexity configuration and Eagle Black represents the highest complexity configuration.

For Encode Arithmetic, we select the MBA expressions synthesized by Loki to substitute add and subtract expressions in *replacers*. We do not use the internal MBA expressions in Tigress as it has been proved that MBA expressions featured in Tigress are limited by hand-generated rules and not robust enough to thwart attacks [32].

4 Implementation

In this section, we first present the implementation of our source code analyzer in Section 4.1 and then we elaborate the generation of DEBRA in Section 4.2.

4.1 Source Code Analyzer

Figure 3 depicts the workflow of our source code analyzer. First, it reads an executable compiled with both debugging information and profiling enabled. Next, the function resolution stage extracts functions from the source files associated with the executable. Then the extracted functions within each source file are ranked based on sensitivity and centrality respectively in descending order. The top $\mathcal{M}$ functions with the greatest sensitivity and the top $\mathcal{N}$ functions with the greatest centrality from each source file are stored in a result set as the targets for further obfuscation. $\mathcal{M}$ and $\mathcal{N}$ are user-defined thresholds, we set $\mathcal{M}$ and $\mathcal{N}$ to three for demonstration purposes in Figure 3 and adopt the same values for building DEBRA. Any dependent functions defined in external libraries or third-party dependencies are excluded as they are not a native part of a testbed program.

The source code analyzer is built on top of `pycparser` (version 2.21) to parse source code files and ascertain the sensitivity for functions, `Cally` and `gprof` (version 2.30) to reason and profile centrality for functions. In total, our source code analyzer is written in approximately 1,200 lines of Python code.

4.2 Generation of *DEBRA*

The automated selection process, followed by human verification, filtered out 224 qualified function-level targets within 15 testbed programs. These targets are subsequently processed with a selection of obfuscation techniques to generate the benchmark. The generation process took about 25 hours on a desktop equipped with AMD Threadripper 1900X 8-Core CPU, 64 GB RAM, NVIDIA 2080Ti GPU, and Ubuntu 18.04 operating system, resulting in a total of 1,917 obfuscated executables.

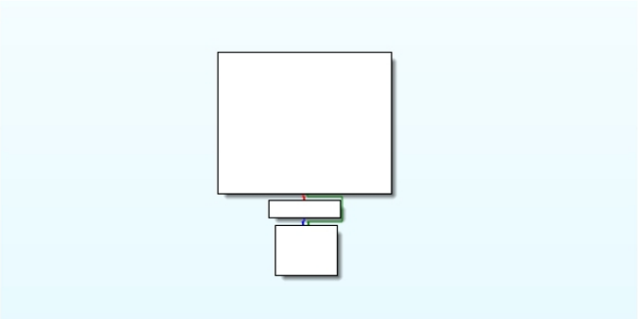
Tigress provides command line options that enable users to specify obfuscation transformations and tune obfuscation parameters. However, a known limitation of Tigress is its inability to fully support obfuscating code that contains features beyond the C99 standard[1]. This poses a challenge for our purpose as all of the testbed programs integrate modern features that are not recognizable to Tigress, hindering its direct application. Previous works have reported similar findings [16, 46]. To the best of our knowledge, adding "-D" flags to define macros that coarsely resemble the unrecognized built-in types and functions during the obfuscation with Tigress could be a partially viable solution. Although theoretically feasible, this approach is labor-intensive and risks altering the original semantics of a program. To address this issue, we take the first step and develop a workaround that indirectly integrates Tigress obfuscation into the testbed programs.

Our approach draws inspiration from the concept of Encoding Literals. More specifically, we replace integer literals within target functions with calls to crafted functions. We call these crafted functions *replacers*. A *replacer* takes an integer literal as an input parameter and is bloated with junk code. The output of a *replacer* is an integer that matches the value of the input literal, thus preserving the target function's semantics. We ensure that a *replacer* does not contain any features that are not processable to Tigress. After that, we obfuscate *replacers* with Tigress and inline them within

[1] https://tigress.wtf/bugs.html

```
1 static bool
2 initialize_wd_for_exec (...)
3 {
4   ...
5   if (execp->wd_for_exec->desc < 0)
6   ...
7 }
```

(a) Integer literal *0* at line 5 in the code snippet of the *initialize_wd_for_exec* function is a target for *replacer*.



(b) Control flow graph for the original *initialize_wd_for_exec* function.

Figure 4: Simplified source code and control flow graph for the *initialize_wd_for_exec* function in the *find* program.

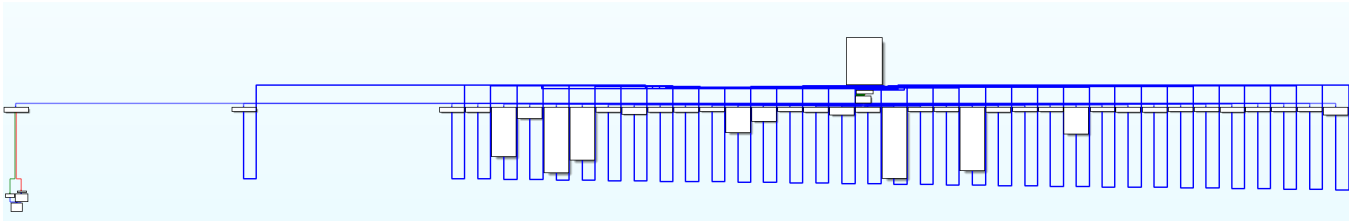


Figure 5: Control flow graph for the *initialize_wd_for_exec* function in the *find* program after replacing the integer literal 0 with a Tigress-Flatten-transformed *replacer*.

the target functions. Prior to the inlining, we run 1,000 test cases to verify the equivalence of inputs and outputs to a *replacer*. This strategy retains the obfuscation effects within the target functions and effectively eradicates the risks of semantics distortions. Figure **??** and Figure 5 display the control flow graphs for the function *initialize_wd_for_exec* before and after the *replacer* is in place. As we can observe, the effect of control flow flattening transformation is evident.

Theoretically, we can produce an unlimited number of obfuscated samples because of the stochastic generation process of *replacers*. Consequently, a user will have access to a new batch of obfuscated samples whenever she executes the generation script we provided. However, for the demonstration in this paper, we present all analyses based on a pre-generated and plan-to-release batch.

5 Experiment and Evaluation

To showcase the strengths and usability of our benchmark, we first conduct a comparative analysis between DEBRA and three peer benchmarks. Next, we reevaluate the deobfuscation method proposed in [31] and Xyntia [24] with DEBRA. Both deobfuscators reported impressive results but without systematic examination of their adaptability to real-world programs in their original evaluations.

5.1 Peer Benchmarks

SPEC CINT2006 [15], MiBench [14] and Obfuscation Benchmarks [8] are the top three prevalent peer benchmarks identified by a recent survey [19].

SPEC CINT2006. SPEC CINT2006 benchmark is the integer component of the SPEC CPU benchmarks developed by SPEC in 2006 to provide performance measurements that can be used to compare CPUs on different computer systems. SPEC CINT2006 includes a collection of 12 real-world applications written in C/C++ with distinct application areas.

MiBench. MiBench is an embedded benchmark suite developed by Guthaus et al. in 2001 [14]. MiBench contains 35 applications written in C, spanning six categories: Automotive and Industrial Control, Consumer Devices, Office Automation, Networking, Security, and Telecommunications.

Obfuscation Benchmarks. Obfuscation Benchmarks was introduced in 2016 by Banescu et al. [8] to evaluate the strength of different obfuscation techniques. It primarily comprises university-level basic algorithm implementations, non-cryptographic hash function implementations, as well as synthesized programs created by Tigress's Random Function transformation. All programs are written or generated in C.

As depicted in Table 3, DEBRA outperforms the peer benchmarks across multiple dimensions. It enhances diversities by deriving obfuscated variants from 15 independent real-world programs spanning distinct functional categories, in contrast to 12 in SPEC CINT 2006, 6 in MiBench, and 3 in Obfuscation Benchmarks respectively. Figure 6 presents a visualization of the count of non-toy programs involved in each benchmark. Both DEBRA and SPEC CINT2006 exclude any toy programs, with DEBRA having three more non-toy programs compared to SPEC CINT2006. While MiBench includes

Table 3: Comparison between DEBRA and the peer benchmarks.

Benchmark	Number of Category	Count of Non-toy Programs	Accessibility	Released
DEBRA	**15**	**15**	**Open access**	**2025**
SPEC CINT2006	12	12	Restricted access	2006
MiBench	6	13	**Open access**	2001
Obfuscation Benchmarks	3	0	**Open access**	2016

35 applications in total, more than half are toy programs. Additionally, all programs in Obfuscation Benchmarks fall under the toy program category. Unlike MiBench and Obfuscation Benchmarks, DEBRA avoids the inclusion of toy or synthetic programs. Instead, we only integrate real-world programs of varying complexities to provide a more robust and realistic evaluation landscape.

SPEC CINT2006 and MiBench are not specifically designed for software protection research. As a consequence, programs provided in these two benchmarks are raw and unobfuscated executables. Users often have two options if they decide to reform the two benchmarks for their software obfuscation and deobfuscation related evaluation. One option involves downloading the codebase of a program from a trusted source, determining specific section(s) for obfuscation, injecting obfuscation transformations within the codebase and subsequently recompiling the program. Alternatively, users can choose to lift a compiled program to an intermediate representation (IR), applying obfuscation transformations at IR level and then recompiling the program. However, both approaches can lead to biases that heavily affect reproducibility, such as variations in obfuscation settings and compiler configurations. On the other hand, Obfuscation Benchmarks provides source code files instead of executables. The majority of the source files contain only a main function, encapsulating the entire program logic. Additionally, it offers an automated obfuscation script that applies selected Tigress obfuscation transformations to the main function. However, the reliance on toy programs limits its ability to provide a comprehensive evaluation of deobfuscation methods. We address these limitations by providing readily available executables that are pre-obfuscated using real-world programs. In addition, we include manifests to document the obfuscated functions and the source files they originate from, the applied obfuscation techniques, and the corresponding obfuscator for each sample in DEBRA.

DEBRA, MiBench, and Obfuscation Benchmarks are freely accessible, whereas accessing SPEC CINT 2006 incurs an inevitable cost, albeit with an educational discount. Programs in DEBRA are at their recent versions at the time of writing this paper. On the other hand, SPEC CINT2006, released in 2007 and retired in 2018, as well as MiBench, which ceased major active maintenance since its release in 2001, potentially lacking in reflecting the current state and trend in modern programming development.

5.2 Deobfuscator Reevaluation

We reevaluate deobfuscation method proposed in [31] and Xyntia [24] with DEBRA and report our findings in this section.

5.2.1 Deobfuscation Method Selection. The selection of considered deobfuscation methods is based on three factors. First, the deobfuscation method must be publicly available and accessible

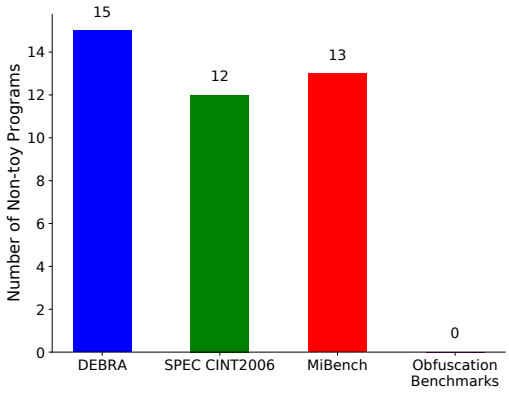


Figure 6: Number of non-toy programs in each benchmark.

to users. Second, the deobfuscation method should come with a working prototype and an adequate guide to facilitate correct usage and thorough evaluation. Third, the allowed input for the prototype needs to include executables or subsidiaries such as execution traces and source code files rather than being restricted to only accepting modular entities such as functions, basic blocks or a set of instructions.

Guided by the criteria, we select the deobfuscation method developed by Salwan et al. [31] and Xyntia [24] as the objects for our evaluation. Since Salwan et al. [31] did not assign an official name to their method in the paper, we will refer to it as "Deobfuscator S" for ease of reference in subsequent context. Deobfuscator S is designed to break virtualization obfuscation introduced by Tigress, it takes a Tigress-virtualization-obfuscated executable as the input, removes VM's instructions and structures buried in it through taint analysis, and returns a restored executable as output. A Python script[2] implementing the entire process has been made available on GitHub. Xyntia [24] is developed to break general obfuscation schemes through a search-based, blackbox deobfuscation approach. Xyntia exhibits the capability to process an input executable, synthesizing simplified, yet semantically equivalent expressions for individual basic blocks retrieved from execution traces based on I/O sampling. The authors released relevant Bash and Python scripts[3] to facilitate the deobfuscation process in an automated manner. Technically speaking, Deobfuscator S can be characterized as a symbolic execution based method, whereas Xyntia belongs to the program synthesis category.

[2]https://github.com/JonathanSalwan/Tigress_protection/blob/master/solve-vm.py
[3]https://github.com/binsec/xyntia/tree/2a9ff5/scripts

Table 4: A side-by-side comparison between obfuscated hash functions within the custom dataset used in the initial evaluation of Deobfuscator S and obfuscated samples in DEBRA subset generated by the *Virtualization* transformation of Tigress. We present min, median, and max values for each metric respective to the number of samples within each set.

	Custom Dataset (460 samples)		**DEBRA Subset** (230 samples)	
	[Min, Max]	Med	[Min, Max]	Med
Functions #	[6, 46]	9	[132, 23,693]	1339
Instructions #	[491, 7,364]	1145	[9,571, 1,468,035]	83,741.5
Size (KB)	[13, 77]	18	[305, 76,049]	2,608.5

5.2.2 Experiment Settings. We test Deobfuscator S with a subset of DEBRA totaling 230 samples generated by the *Virtualize* transformation of Tigress and Xyntia with the complete set of samples in DEBRA respectively. Despite the variance in options and values for using Tigress to generate samples in DEBRA as opposed to those in the custom datasets used in the initial evaluation of Deobfuscator S and Xyntia, our primary focus is to measure the competence of such deobfuscators, whose performance in real-world settings remains unknown, in handling obfuscated samples derived from real-world programs rather than repeatedly analyzing their capabilities across varying protection settings.

As a comparison, Deobfuscator S's original experiments were conducted using a custom dataset of 920 obfuscated samples. Each of those obfuscated samples was generated with the *Virtualize* transformation of Tigress configured with 46 distinct values across 13 options. These samples were derived from two sources: 10 hash function implementations and 10 programs from the Tigress Challenge. Deobfuscator S was able to restore all of the 920 obfuscated samples to their functionally equivalent versions prior to obfuscation as reported by the authors. Xyntia unveils obfuscated code, irrelevant to the applied obfuscation techniques, by synthesizing syntactically simplified expressions that mirror the I/O behaviors of the original code. Its efficacy was mainly tested with two expression datasets containing Boolean, Arithmetic, and Mixed Boolean-Arithmetic expressions with up to 5 variables. Xyntia achieved an average synthesis rate of 96.21% in delivering qualified alternatives across multiple experimental runs.

Table 4 presents a side-by-side comparison between the obfuscated hash function samples from the custom dataset used in the original evaluation of Deobfuscator S (metrics are calculated based on the obfuscated hash function samples because of the unavailability of the Tigress Challenge samples, but we anticipate the metrics would be very close to the real situation based on the open data released in their paper) and those in DEBRA subset. We can observe that substantial differences exist in each metric, most notably in the maximum number of instructions where the difference reaches 200 times. A similar trend is also observed in the number of functions and sizes where the maximum number of functions and the maximum value of size in the custom dataset are not even close to half of their counterparts in DEBRA subset. This reveals how different real-world programs are from toy programs in terms of complexities especially when obfuscations are applied.

Table 5: The amount of instructions that Deobfuscator S can process in each testbed program.

Testbed	**Total**	**Processed**	**Percentage**
bash	192,134	32	0.017%
chmod	17,431	41	0.230%
cp	24,396	40	0.164%
curl	28,653	53	0.185%
find	56,696	33	0.058%
gcc	49,098	100	0.204%
grep	42,076	43	0.102%
gzip	21,141	213	1.008%
httpd	125,606	53	0.042%
ls	22,719	40	0.176%
nano	64,750	256	0.395%
OpenSSL	1,140,830	172	0.015%
QEMU	1,446,341	2560	0.177%
SQLite	322,458	34	0.011%
tar	86,712	28	0.032%

5.2.3 Result Analysis. We ran Deobfuscator S on DEBRA subset and the results are reported in Table 5. We reported the quantity of instructions that could be processed by Deobfuscator S instead of the number of samples being successfully restored by it. The reason is that Deobfuscator S was not able to deobfuscate any of the samples in DEBRA subset and crashed right away for nearly all samples. The median number of instructions Deobfuscator S managed to process is 43 instructions per sample, with the highest 2560 instructions for QEMU samples and the lowest 28 instructions for tar samples. In terms of the processing rate, gzip samples top with just over 1% whereas the processing rate on SQLite samples is as low as 0.011%.

Deobfuscator S. After carefully looking into the implementations, we identify two major limitations that contribute to the undesired outcomes. First, Deobfuscator S struggles with dynamically linked executables because it relies on a pre-defined list (18 functions) for managing transfers to external functions. For external functions within the list, Deobfuscator S sets the register *rip* to the function's return address, safely bypassing the call. However, an external function outside the list leaves Deobfuscator S at a loss, triggering an emulation error that mistakenly zeroes out the register *rip*. The emulation is disrupted immediately and leads to the crash. Second, Deobfuscator S explicitly searches for a set of hard-coded entrance and exit points within the codebase to start analysis. The entrance point is indicated by the *strtoul* function, and the function's return value serves as the source of Deobfuscator S's taint analysis. The *printf* function acts as the indicator of an exit to terminate the analysis and the function's second parameter (e.g., *sink* in *printf("%d", sink);*) serves as the sink of Deobfuscator

```
.....
35678: cmp DWORD PTR [rip+0x304179], 0x0
                # 3397f8 <pretty_print_mode>
.....
35694: call 36b00 <pretty_print_loop>
.....

356d7: mov DWORD PTR [rip+0x304117], 0x0
                # 3397f8 <pretty_print_mode>
.....
```

Figure 7: Snippet from the disassembled *main* function of the *bash* program. Lines containing the substring "ret" are erroneously identified as the locations of *ret* instruction by Xyntia.

S's taint analysis. In the absence of either function, Deobfuscator S is incapable of performing its taint analysis. Moreover, we discovered that Deobfuscator S requires an exact textual match between the name of an external function and those in the predefined list. For instance, Deobfuscator S differentiates between calls to function *__printf_check* and *printf*, although both originate from the function *printf* in the source code. It's worth noting that such strict conditions are rarely true in real-world scenarios. As a result, Deobfuscator S stumbled with the real-world samples in DEBRA.

Interestingly, Kochberger et al. [20] noticed that the presence of the function *strtoul* and the function *printf* are required as well when evaluating Deobfuscator S, referred to in their work as *Tigress DeObf* with a custom dataset primarily consisting of toy programs. Deobfuscator S started to work as expected and managed to restore most of the samples after Kochberger et al. [20] patched their samples derived from toy programs to include the *strtoul* and the *print* functions within the *main* function. Upon comparing the datasets used for evaluation in Salwan et al. [31] and Kochberger et al. [20], we find significant similarities. Kochberger et al. [20] selected hash function implementations from Obfuscation Benchmarks [3], and generated four distinct sets of obfuscated samples from them. The datasets in both cases heavily consist of hash function implementations. These implementations share the same naive structures, with a main function and another custom function containing the hash algorithm. Additionally, there are no features that can be commonly found in real-world programs such as complex data structures included in these implementations. The similar natures of these datasets allow Deobfuscator S to achieve excellent performance in both cases, and the hand-crafted rules are enough to cover all situations in these samples so that other limitations we discovered were not encountered and reported in their work [20].

Xyntia. Xyntia was evaluated across all 1,917 samples in our benchmark with the proven optimal search strategy, Iterated Local Search, synthesis timeout, 60s, and objective function, logarith to maximize the efficacy. However, Xyntia was limited to processing 205 samples derived from 4/15 of the testbed programs (*chmod*, *cp*, *grep*, and *ls*). The failures can be attributed to its tracing engine. Primarily, the engine uses Python's *in* keyword to search for the *ret* instruction within the *main* function of an executable to define the boundaries of tracing. This strategy is flawed in instances where

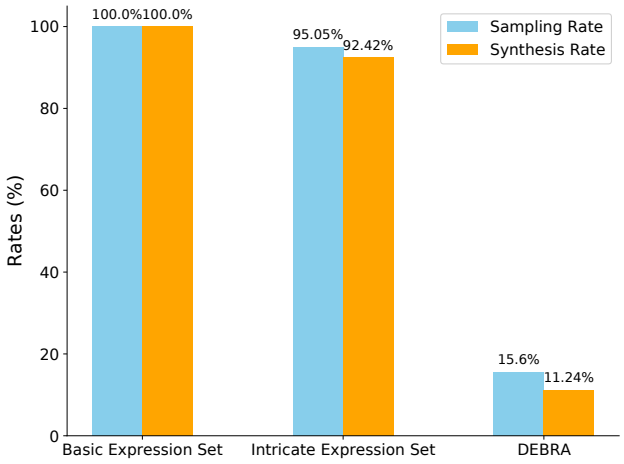


Figure 8: Comparing average sampling and synthesis rates Xyntia achieved on DEBRA with those reported in the original evaluation.

the *ret* instruction is absent, as observed in the *main* function of the *nano* program, or when false positives occur, as depicted in Figure 7. Under both scenarios, the tracing engine shuts itself down. Additionally, even when the *ret* instruction's location is correctly captured, the engine's inadequate handling of recursions within external functions can trap itself forever.

In Figure 8, we present the average sampling and synthesis rates Xyntia achieved on DEBRA versus those reported in the original evaluation. The sampling rate represents the proportion of retrieved basic blocks for which Xyntia learned I/O behaviors, while the synthesis rate reflects the percentage of basic blocks from the pool of sampled blocks for which Xyntia synthesized semantically equivalent, yet simplified expressions. On average, Xyntia sampled 15.6% of the retrieved basic blocks per program in DEBRA. However, **none** of the target basic blocks containing obfuscated code segments were in that category. A closer examination of the basic blocks sampled by Xyntia revealed that a non-negligible portion of them were trivial cases with one-to-one I/O behaviors. Xyntia struggled with complex basic blocks due to the internal SMT expression builder's limitations in efficiently handling such blocks. However, such basic blocks are common in real-world scenarios especially when obfuscation is applied. On the other hand, Xyntia achieved an average synthesis rate of 11.24%, a notable drop from over 90% synthesis rate reported in the original evaluation. This discrepancy could stem from the complexity of our samples, which include a greater number of I/O variables compared to the maximum of 5 variables considered in the initial evaluation.

The evaluation results demonstrate that DEBRA can simulate real-world deobfuscation tasks and thus expose the problems derived from a real-world point of view. Please note that the evaluation results should not be interpreted as deficiencies of the evaluated tools, because deobfuscation technique design is often inherently a target and obfuscation technique specific task rather than a generic problem. The observed failures should be attributed to mismatches in context between the design expectations of the evaluated tools

and those in our experiments. Therefore, DEBRA's contribution is not meant to challenge the validity of existing deobfuscation tools, but to provide a ground for comparative evaluation and to motivate future deobfuscation techniques to better handle real-world obfuscated programs.

6 Limitation

We selectively incorporate obfuscation techniques and obfuscators to construct DEBRA based on their frequency of occurrence in recent works. While we do not exhaust obfuscation techniques and obfuscators, DEBRA is designed to be extensible. Users can freely augment it based on their needs as our source code analyzer pinpoints candidate locations for obfuscation. We consider expanding DEBRA is necessary and as part of our future work.

Another unsupported aspect is multi-layer obfuscation, which involves one or several obfuscation techniques applied once or multiple times to a program in an attacker-agnostic order. Obfuscation techniques are often found to be deployed in combination to enhance the holistic resilience against reverse engineering in common practice. The synergy effects of different obfuscation techniques have been validated in multiple works [32, 39, 42]. Also, the order of obfuscation proves to be a key factor to yield optimal obfuscation strength in Wang et al. [39]. However, we are not aware of widely adopted, well-established solutions to systematically break layered obfuscation. Therefore, we consider the investigation of how to build a practical layered obfuscation upon DEBRA as part of our future work as well.

While we seek to mirror real-world practices as closely as possible, the current obfuscation target searching risks overlooking potential candidates. As a promising direction moving forward, we will incorporate more static features such as Halstead Complexity Measures and Cyclomatic Complexity to augment the source code analyzer.

7 Related Work

High-quality benchmark datasets are invaluable yet scarce resources for deobfuscation technique evaluations, malware datasets serve as an ideal option for such evaluations. Malware is rich with real-world instances of sophisticated obfuscation techniques injected by attackers to evade detection [45]. Notable malware datasets such as Drebin [5] and MalGenome [48] include a wide array of malware captured in the wild, are highly regarded within the malware research community. However, the lack of ground truth concerning the precise obfuscation locations undermines their usability as standardized evaluation targets for deobfuscation techniques.

Zhao et al. [47] constructed a large-scale obfuscation dataset to evaluate their research work on obfuscation scheme prediction. The dataset includes over 240,000 obfuscated samples by obfuscating gcc version 7.4 and GNU Tookits without explicit specifications such as versions and programs with a selection of 8 obfuscation transformations from two open-source obfuscators, O-LLVM and Tigress. However, the authors have not made the dataset publicly available, and certain details concerning the generation process are omitted, which poses a limitation due to a lack of standardization. Moreover, all obfuscated samples are further processed to facilitate their specific evaluation purposes, rendering the dataset unsuitable for generic deobfuscation technique evaluations.

Kochberger et al. [20] obfuscated hash function implementations collected from Obfuscation Benchmarks to build datasets for evaluating four deobfuscation methods specializing in breaking virtualization. While the complete datasets are not made publicly available, the authors detailed the process of generating the datasets. However, the components of the datasets are limited to toy programs. As a result, their datasets cannot serve as a standard benchmark for generic deobfuscation technique evaluations.

8 Conclusion

We identify the limitations in current deobfuscation method evaluation patterns and point out the long-standing deficiency of a comprehensive and standardized benchmark to support systematic and impartial evaluation of deobfuscation methods. In this paper, we present DEBRA, a real-world benchmark built to address the current limitations. We applied a selection of obfuscation techniques to function-level targets reported by a metric-driven source code analyzer within 15 real-world programs to generate the benchmark. DEBRA is a publicly available set and we hope our work boosts both the advancement and comparative evaluation of deobfuscation methods.

Acknowledgments

We sincerely thank our shepherd for the valuable guidance during the revision process and the anonymous reviewers for their constructive feedback during the review process. This research was supported by NSF grants 2211905, 2022279, and 2154606.

References

[1] [n. d.]. Code Virtualizer. https://www.oreans.com/CodeVirtualizer.php.
[2] [n. d.]. gooMBA. https://github.com/HexRaysSA/goomba.
[3] [n. d.]. Obfuscation Benchmarks. https://github.com/tum-i4/obfuscation-benchmarks.
[4] [n. d.]. The Tigress C Obfuscator. https://tigress.wtf.
[5] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of android malware in your pocket.. In *Ndss*, Vol. 14. 23–26. doi:10.14722/ndss.2014.23247
[6] Alessandro Bacci, Alberto Bartoli, Fabio Martinelli, Eric Medvet, Francesco Mercaldo, Corrado Aaron Visaggio, et al. 2018. Impact of Code Obfuscation on Android Malware Detection based on Static and Dynamic Analysis.. In *ICISSP*. 379–385. doi:10.5220/0006642503790385
[7] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 189–200. doi:10.1145/2991079.2991114
[8] Sebastian Banescu, Christian Collberg, and Alexander Pretschner. 2017. Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning. In *26th USENIX Security Symposium (USENIX Security 17)*.
[9] Sebastian Banescu and Alexander Pretschner. 2018. A tutorial on software obfuscation. *Advances in Computers* 108 (2018), 283–353. doi:10.1016/bs.adcom.2017.09.004
[10] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 643–659. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko
[11] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. *A taxonomy of obfuscating transformations*. Technical Report. Department of Computer Science, The University of Auckland, New Zealand.
[12] Ninon Eyrolles, Louis Goubin, and Marion Videau. 2016. Defeating mba-based obfuscation. In *Proceedings of the 2016 ACM Workshop on Software PROtection*. 27–38. doi:10.1145/2995306.2995308
[13] Yoann Guillot and Alexandre Gazet. 2010. Automatic binary deobfuscation. *Journal in computer virology* 6, 3 (2010), 261–276. doi:10.1007/s11416-009-0126-4

[14] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 3–14. doi:10.1109/WWC.2001.990739

[15] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17. doi:10.1145/1186736.1186737

[16] Harshvardhan P Joshi, Aravindhan Dhanasekaran, and Rudra Dutta. 2015. Trading off a vulnerability: does software obfuscation increase the risk of rop attacks. *Journal of Cyber Security and Mobility* (2015), 305–324. doi:10.13052/jcsm2245-1439.444

[17] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*. doi:10.1109/SPRO.2015.10

[18] Zeliang Kan, Haoyu Wang, Lei Wu, Yao Guo, and Daniel Xiapu Luo. 2019. Automated deobfuscation of Android native binary code. *arXiv preprint arXiv:1907.06828* (2019). doi:10.48550/arXiv.1907.06828

[19] Patrick Kochberger, Sebastian Schrittwieser, Bart Coppens, and Bjorn De Sutter. 2023. Evaluation Methodologies in Software Protection Research. *arXiv preprint arXiv:2307.07300* (2023). doi:10.48550/arXiv.2307.07300

[20] Patrick Kochberger, Sebastian Schrittwieser, Stefan Schweighofer, Peter Kieseberg, and Edgar Weippl. 2021. Sok: Automatic deobfuscation of virtualization-protected applications. In *Proceedings of the 16th International Conference on Availability, Reliability and Security*. 1–15. doi:10.1145/3465481.3465772

[21] Minerva Labs. 2020. Egregor Ransomware – An In-Depth Analysis. https://minerva-labs.com/blog/egregor-ransomware-an-in-depth-analysis/.

[22] Binbin Liu, Junfu Shen, Jiang Ming, Qilong Zheng, Jing Li, and Dongpeng Xu. 2021. {MBA-Blast}: Unveiling and Simplifying Mixed {Boolean-Arithmetic} Obfuscation. In *30th USENIX Security Symposium (USENIX Security 21)*. 1701–1718.

[23] Haoyu Ma, Chunfu Jia, Shijia Li, Wantong Zheng, and Dinghao Wu. 2019. Xmark: dynamic software watermarking using Collatz conjecture. *IEEE Transactions on Information Forensics and Security* 14, 11 (2019), 2859–2874.

[24] Grégoire Menguy, Sébastien Bardin, Richard Bonichon, and Cauim de Souza Lima. 2021. Search-based local black-box deobfuscation: understand, improve and mitigate. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2513–2525. doi:10.1145/3460120.3485250

[25] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. 2015. Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. 757–768. doi:10.1145/2810103.2813617

[26] Camille Mougey and Francis Gabriel. 2014. DRM obfuscation versus auxiliary attacks. In *Recon conference*.

[27] Philip OKane, Sakir Sezer, and Kieran McLaughlin. 2011. Obfuscation: The Hidden Malware. *IEEE Security and Privacy* (2011). doi:10.1109/MSP.2011.98

[28] Chengbin Pang, Tiantai Zhang, Xuelan Xu, Linzhang Wang, and Bing Mao. 2023. OCFI: Make Function Entry Identification Hard Again. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 804–815. doi:10.1145/3597926.3598097

[29] Rolf Rolles. 2009. Unpacking virtualization obfuscators. In *3rd USENIX Workshop on Offensive Technologies.(WOOT)*.

[30] Kevin A Roundy and Barton P Miller. 2013. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys (CSUR)* 46, 1 (2013), 1–32. doi:10.1145/2522968.2522972

[31] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. 2018. Symbolic deobfuscation: From virtualized code back to the original. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 372–392. doi:10.1007/978-3-319-93411-2_17

[32] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. 2022. Loki: Hardening code obfuscation against automated attacks. In *31st USENIX Security Symposium (USENIX Security 22)*. 3055–3073.

[33] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys (CSUR)* 49, 1 (2016), 1–37. doi:10.1145/2886012

[34] Vít Šembera, Masarah Paquet-Clouston, Sebastian Garcia, and Maria Jose Erquiaga. 2021. Cybercrime specialization: An exposé of a malicious Android Obfuscation-as-a-Service. In *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. doi:10.1109/EuroSPW54576.2021.00029

[35] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2009. Automatic Reverse Engineering of Malware Emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P'09)*. doi:10.1109/SP.2009.27

[36] Ramtine Tofighi-Shirazi, Irina-Mariuca Asavoae, Philippe Elbaz-Vincent, and Thanh-Ha Le. 2019. Defeating opaque predicates statically through machine learning and binary analysis. In *Proceedings of the 3rd ACM Workshop on Software Protection*. 3–14. doi:10.1145/3338503.3357719

[37] Erik van der Kouwe, Gernot Heiser, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2019. SoK: Benchmarking flaws in systems security. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 310–325. doi:10.1109/EuroSP.2019.00031

[38] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. Software tamper resistance: Obstructing static analysis of programs. (2000).

[39] Huaijin Wang, Shuai Wang, Dongpeng Xu, Xiangyu Zhang, and Xiao Liu. 2020. Generating effective software obfuscation sequences with reinforcement learning. *IEEE Transactions on Dependable and Secure Computing* 19, 3 (2020), 1900–1917. doi:10.1109/TDSC.2020.3041655

[40] Tianyu Wu, Shizhu He, Jingping Liu, Siqi Sun, Kang Liu, Qing-Long Han, and Yang Tang. 2023. A brief overview of ChatGPT: The history, status quo and potential future development. *IEEE/CAA Journal of Automatica Sinica* 10, 5 (2023), 1122–1136. doi:10.1109/JAS.2023.123618

[41] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. 2018. VMHunt: A verifiable approach to partially-virtualized binary code simplification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 442–458. doi:10.1145/3243734.3243827

[42] Hui Xu, Yangfan Zhou, Jiang Ming, and Michael Lyu. 2020. Layered obfuscation: a taxonomy of software obfuscation techniques for layered security. *Cybersecurity* 3, 1 (2020), 1–18. doi:10.1186/s42400-020-00049-3

[43] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*. doi:10.1109/SP.2015.47

[44] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294. doi:10.1145/1993316.1993532

[45] Ilsun You and Kangbin Yim. 2010. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*. IEEE, 297–300. doi:10.1109/BWCCA.2010.85

[46] Naiqian Zhang, Daroc Alden, Dongpeng Xu, Shuai Wang, Trent Jaeger, and Wheeler Ruml. 2023. No Free Lunch: On the Increased Code Reuse Attack Surface of Obfuscated Programs. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 313–326. doi:10.1109/DSN58367.2023.00039

[47] Yujie Zhao, Zhanyong Tang, Guixin Ye, Dongxu Peng, Dingyi Fang, Xiaojiang Chen, and Zheng Wang. 2020. Semantics-aware obfuscation scheme prediction for binary. *Computers & Security* 99 (2020), 102072. doi:10.1016/j.cose.2020.102072

[48] Yajin Zhou and Xuxian Jiang. 2012. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*. IEEE, 95–109. doi:10.1109/SP.2012.16