VMorph: A Virtualization/Metamorphic Framework for Binary Obfuscation and Intellectual Property Protection

Pierciro Caliandro[1], Matteo Ciccaglione[1], Andrea Pepe[1], Giuseppe Bianchi[1] and Alessandro Pellegrini[1,*]

[1] *Tor Vergata University of Rome, Rome, Italy*

Abstract

In this paper, we analyse the effectiveness of combining obfuscation and metamorphism techniques to evade antivirus detection and protect intellectual property. We do so by introducing a new framework called VMorph, which jointly utilises emulation and metamorphic techniques to thwart attempts at reconstructing the application's behaviour and accessing internal details or secrets. We assess the performance of VMorph to determine its suitability for safeguarding the intellectual property of the application. The findings indicate a (bounded) decrease in performance, which is still acceptable for securing applications. Additionally, we investigate the stealth capabilities of the proposed technique, which enhances its ability to bypass common static analysis techniques. Based on the results, we also suggest detection techniques that can be employed to mitigate the risk that this technique is used to evade antivirus detection.

Keywords

Binary obfuscation, Metamorphic techniques, Virtualization-based obfuscation, Intellectual property protection

1. Introduction

Binary obfuscation encompasses a range of methodologies aimed at manipulating applications to conceal their internal logic by means of *code confusion* [1, 2]. The ultimate goal of binary obfuscation is to make the analysis process more difficult and time-consuming [3]. These techniques serve a dual purpose: they can be employed to secure applications against reverse engineering [4, 5, 6], or they can be abused for malicious intents, such as evading antivirus detection by malware [7, 8].

Several attempts have been made to provide techniques that enable *automatic* obfuscation of generic applications. *Binary code permutation* is a simple reorganisation of the control flow graph of the application [9], while *encryption* and *packing* are strong obfuscation techniques largely diffused in malicious application [10]. Encryption is based on the use of cryptography to hide the representation of the program [11], and packing [12] is the art of embedding the application into some container that cannot be executed directly but needs the support of another piece of code, the *unpacker*, which brings the payload into executable form and activates it.

Binary code virtualization [13, 14, 15] is essentially the conversion of executable code from a well-known (typically native) Instruction Set Architecture (ISA) into a new one, which is often hand-crafted and undocumented, and that can be interpreted by an underlying interpreter or Virtual Machine (VM). This technique results in a file that cannot be executed directly on the CPU but must be loaded as a guest on the VM and then interpreted and emulated by it.

All of these techniques suffer from a common pitfall: they are inherently static in that, once generated, the program's binary image is not subject to any change. This means that static analysis and debugging techniques can allow uncovering the code's original version or providing insights on the program's logic [16, 17, 18]. In some cases, the process can also be automated [15].

Joint National Conference on Cybersecurity (ITASEC & SERICS 2025), February 03-8, 2025, Bologna, IT

[*] Corresponding author.

p.caliandro@ing.uniroma2.it (P. Caliandro); m.ciccaglione@ing.uniroma2.it (M. Ciccaglione); pepe.andmj@gmail.com (A. Pepe); giuseppe.bianchi@uniroma2.it (G. Bianchi); a.pellegrini@ing.uniroma2.it (A. Pellegrini)

0009-0001-3549-9353 (P. Caliandro); 0009-0005-7348-2674 (M. Ciccaglione); 0009-0001-6981-3259 (A. Pepe); 0000-0001-7277-7423 (G. Bianchi); 0000-0002-0179-9868 (A. Pellegrini)

In contrast, *metamorphism* [19, 20] is a technique that exceeds this limitation by changing the binary form of the program at each generation without impacting the program's semantics. Therefore, any static or automatic analysis will face different versions of the same program whenever the same application is started, making it more difficult to get insights into what the program is doing. Both static and dynamic analysis techniques face significant challenges. However, more advanced detection methods, particularly those that analyse similarities between different generation of programs, may effectively identify metamorphic applications [21, 22].

Jointly using obfuscation and metamorphic techniques may offer a more covert approach to hiding application internals. To the best of our knowledge, the joint use of these techniques has never been explored. In this paper, we analyse the effectiveness of combining these techniques by introducing VMorph, a novel framework for code obfuscation that employs both virtualization and metamorphism.

VMorph allows the automatic building of an executable obfuscated using emulation, either starting from the application's sources or the compiled version, thanks to binary transformation techniques. Metamorphism is applied to the virtualised ISA used by the emulator and the emulator itself. This approach results in a final binary that mutates itself entirely upon each execution. Thus, the metamorphic transformations are concealed, making it difficult to correlate the different versions.

We evaluated the stealth capabilities of this approach by exercising existing malware samples transformed with VMorph against off-the-shelf antiviral solutions. Based on the results, we propose possible detection approaches that can be studied to reduce the impact of this kind of threat. Also, we study the performance impact of VMorph on transformed applications to understand the viability of the approach to protect benign software from reverse engineering for intellectual property (IP) protection purposes.

The remainder of this paper is structured as follows. In Section 2, we discuss related work. Our metamorphic virtual machine is presented in Section 3. We present an experimental assessment in Section 4.

2. Related Work

Obfuscation techniques are widely discussed in the literature, both from the evasion and detection point of view. The panorama is vast, with techniques such as ROP-based obfuscation [23], opcode hiding [24], code virtualization [25, 26], control flow redirection based on exceptions [27] and mimimorphic techniques [28] based on attacking both semantic and statistical analysis. Another type of hiding technique is based on the use of packers, i.e. compressing or encrypting binary code in the executable file, adding an unpacking stub that, when activated, reverses the packing operation before giving control to the entry point of the original code [29, 30, 31, 29].

All of these obfuscation techniques share the same problem: they are static and do not change executable files over time. Therefore, once reverse engineers have recognised the underlying pattern (e.g. how to unpack or how the code is virtualised), they can construct a deobfuscated version of the executable to analyse it without impediments. Our solution to this problem does not involve including additional levels of complexity in the binary representation, as many of the above contributions did. Instead, we switched focus, allowing the obfuscated application to modify itself over time. This is done through a metamorphic engine that can drastically rewrite itself, based on some source of randomness, to prevent reverse engineers from observing two identical versions of the same program.

Malware writers have applied metamorphism to protect their code against detection activities. The most relevant example is Metaphor [32, 33]. Indeed, the author showed how to build a metamorphic engine to effectively change the binary code of malware. The techniques reported in [32] were a starting point for this work. VMorph extends current literature results by combining virtualization and metamorphism, allowing developers to obfuscate and metamorphose their applications even if they were not built natively to do so. The result is that VMorph's main improvement to the state of the art introduces an obfuscation technique which can be resilient to static analysis by converting the original binary code into a self-modifying ISA.

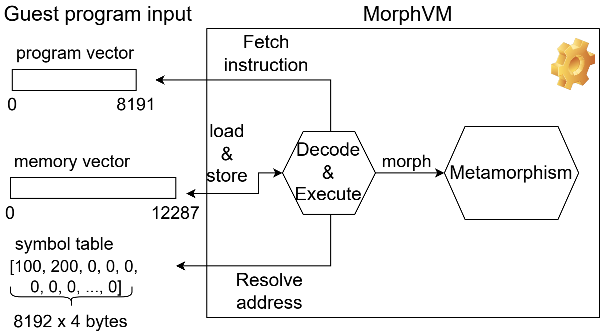


Figure 1: VM Architecture.

In the field of binary manipulation, various proposals in the literature offer the capability to manipulate executable files (see, e.g.,[34, 35, 36]) with differing levels of flexibility. These proposals primarily focus on studying or enhancing the non-functional properties of the applications, which limits the scope of modifications they can support. Conversely, to create an effective metamorphic engine, a significantly broader range of alterations to binary files must be supported. This necessitates the development of meticulous solutions to avoid compromising the correctness of the application and to extract information that is not explicitly available in the program, possibly through ad-hoc heuristics. This forms a crucial part of the proposal on which VMorph is based.

The problem of the static structure of virtualised program emulators has already been acknowledged in the literature, and several papers have attempted to address it. In [26], the authors propose a technique that employs diversified dispatchers to decode the program. This approach makes it more complicated to analyse the structure of the emulator, but it still relies on a static structure, which we overcome with VMorph. In [37], the authors enhance the protection of virtualised code by leveraging encryption based on dynamically generated keys, which depend on an anti-tamper code that, if modified during the analysis, no longer generates the correct keys. This approach relies on an additional layer of security, which we consider orthogonal to the mechanism we present in this paper.

3. VMorph

The VMorph metamorphic virtualization framework includes a retargetable virtual machine interpreter and a metamorphic engine. VMorph can be integrated into the source code as a development library, or used on compiled applications. When used as a library, developers can obfuscate specific parts of the application using VMorph's API, while leaving optimised code untouched. In this way, performance-critical parts of the applications can be left native. Binary transformation can be useful for privacy-preserving purposes, obfuscating and protecting applications which were not explicitly designed to natively rely on code protection mechanisms. With this approach, the VMorph framework can also be included in a CI/CD pipeline. We implemented binary transformation by means of a *transpiling approach*, but for the sake of space, we do not discuss this aspect in this paper.

3.1. Interpreter and ISA

The general architecture of the VMorph interpreter is shown in Figure 1. The interpreter relies on two large arrays representing the virtualised `program` and `memory`. The former contains instructions for the virtualised program, which are fetched one by one, decoded, and executed; the latter holds data for the guest program. The interpreter maps to the `memory` array different logical regions necessary to support the correct execution of generic applications. In the initial part, we keep *global data*, *read-only* variables and optional command line *arguments* used by the guest program, in this order. Then we place contiguously the *heap* and the *stack* of the virtualised application.

As mentioned earlier, one of the goals of VMorph is to enable the metamorphism of both the interpreter and the virtualised program. To achieve virtualised program metamorphism, we do not rely on a single ISA; instead, we have defined a generic structure for the ISA, upon which multiple variations can be created. Therefore, the program uses only one of the possible ISAs that the interpreter can run. When a metamorphic transformation is applied to the virtualised program, a new representation of the instructions is randomly generated, making it more difficult to correlate the results of previous iterations of static analysis. Each byte of the `program` can either be part of a legitimate instruction or a *padding* byte. In fact, *padding* is one way in which multiple versions of the ISA are implemented, as there can be several different encodings for each logical instruction while maintaining the same meaning.

The ISA register file is divided into several categories. There are 16 general-purpose registers, namely `r0` through `r15`. The `tgt` register keeps the destination address of every jump the guest program performs. The `v0` register keeps the return value of subroutines, while `ret` keeps the return address. `ip` holds the program counter, while `sp` points to the top of the stack. The `brk` register keeps the program break to identify the valid portion of the virtualised heap.

All registers are 64-bit. They are a mixture of different architectural organisations of real-world CPUs, taking some ideas from MIPS, x86, RISC-V, and ARM architectures [38]. This choice avoids abiding by a single architectural organisation in the attempt to make it less evident what certain logical structures are used for in static analysis.

Registers are used with specific calling conventions. When invoking functions (belonging to the virtualised program, to some library or a syscall), parameters are placed into registers `r0` through `r15` in sequential order. This means that there must be a mapping between the native ISA and the VMorph one. To accommodate the different calling conventions used by libc and syscalls, before each invocation, the value in each register used to pass a function parameter is saved by pushing it onto the stack. After the function returns, the saved values are restored back into their respective registers.

The `libc` register maintains a second-order opcode to select any standard C library function with a single instruction. Dedicated support for the standard library is essential because the interpreter mediates access to the underlying system, ensuring the program remains under the interpreter's control and cannot bypass it by calling the regular standard library directly. Additionally, providing personalised access to the standard library, where most functions are internally reimplemented by VMorph, can reduce the chances of studying a virtualised program through behavioural analysis at the standard library level.

The instructions supported by the interpreter are grouped in classes, such as arithmetic-logic ones or load/store to access guest memory. We provide a `lea` (load effective address) instruction to convert addresses from the guest to the host address space. All jumps are indirect and use the dedicated `tgt` register to keep the destination in the `program` array. Conditional jump operations combine comparison and jump functionalities into one instruction, resulting in the modification of the `ip` register (based on the `tgt` register's value) when the specified condition is satisfied.

The `arit` instruction indexes another second-order opcode table that implements arithmetic operations, both integer and floating point. The `brk` instruction grows the heap in the `memory` array when dealing with memory allocation. `syscall` and `libc` instructions are, respectively, the access points to invoke a system call or a standard library function (relying on the `libc` register mentioned above). The `morph` instruction explicitly activates the VMorph metamorphic engine.

The `brk` register and instruction are used by a custom memory allocator to support dynamic memory allocation and deallocation through `malloc` and `free`. The need for a custom dynamic memory manager is similar to the reasons for having a custom standard library: to prevent the virtualised program from escaping the control of the interpreter.

The basic allocator implemented in VMorph is quite simple: the entire available heap space is divided into chunks, each associated with metadata that includes the chunk's size and a flag indicating whether the chunk is in use. Whenever the virtualised program needs a new block of dynamic memory, the allocator first checks if there is a previously allocated block with a size greater than or equal to the requested size that has been freed. If so, the block is split into two if its size is more than twice

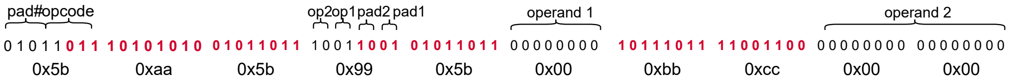


Figure 2: VMorph instruction format.

the requested size. A new block is allocated using the `brk` instruction if no free blocks exist. If the `brk` instruction fails, an ENOMEM error is returned to the virtualised program. When `free` is called, the "in-use" flag is reset, and the memory content is zeroed out before making it available for future allocations[1].

The binary format of the instructions is shown in Figure 2: bold red bits are random padding, while regular black ones are semantically valid. Random padding bytes can appear *in the middle* of instructions to increase their confusion. The meaning of each (non-padding) byte is the following:

- The first byte is the *primary opcode*: its first 5 bits encode the first-level ISA operation, while the last 3 specify how many padding bytes immediately follow.
- The second is the `mod` byte, whose first 4 bits are the size of the source and destination operands (if present), and the last 4 bits tell how many padding bytes will be placed before each operand.
- Then, there are the operands preceded by padding bytes, if any.

Each instruction can have 0 to 2 operands, which come in various types and sizes. The register encodings are 8 bits in size, and their value corresponds to the register number. The size of a memory operand is 16 bits, representing an offset in the `memory` array. The size of the value loaded/written from/to memory depends on the current value of the `size` register, which specified the size of the data transfer in *bits* (up to 64). An immediate also has a size of 16 bits, representing the immediate value itself. We do not support immediate sizes other than 16 bits.[2].

It is noteworthy to emphasize the importance of padding. As depicted in Figure 2, a byte can be used to encode a valid ISA opcode or act as a padding byte. In this sense, the same value can have different meanings in the program's byte stream. Padding is also helpful in creating a CISC-like ISA for VMorph because the size of an instruction can vary depending on the number of padding bytes used to encode it. However, the interpreter's implementation follows a RISC-like approach, which can be confusing to analysts.

3.2. Metamorphic Transformations

The VMorph framework allows for automatic metamorphism of both the virtualised program (from now on, the "guest") and the interpreter (from now on, the "host"). The guest and host undergo different transformations due to their technical nature. Our reference implementation targets Linux systems adhering to the System V ABI [39]. We discuss separately how we handle host and guest metamorphism.

3.2.1. Host Metamorphism

For the host program, metamorphism involves rewriting the binary code to change the order and type of instructions without altering its functionality. This approach takes inspiration from previous work in [32] and divides the metamorphic engine into functional blocks, as shown in Figure 3. The process begins with the *disassembler*, which reads and interprets ELF binaries [39], and outputs an intermediate representation composed of multiple linked data structures, inspired by [34]. This representation includes function information, ELF metadata detailing the position and size of sections and segments, and cross-reference information between instructions and data. Since metamorphic transformations must preserve references, any modifications must respect these relationships. Transformations are

[1] This last step is purposely not compliant with standard operations of the `malloc` library.

[2] Larger values can be constructed in the 64-bit registers using arithmetic/logic instructions. This further complicates the generated virtualised code.

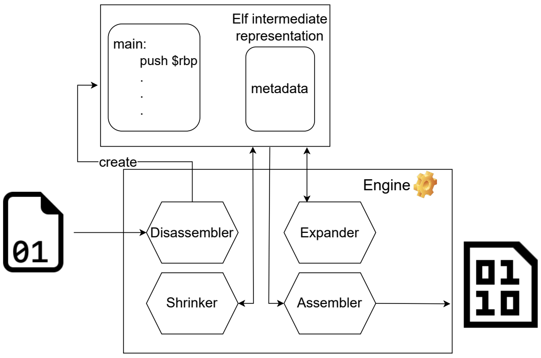


Figure 3: VMorph metamorphism.

always performed on this intermediate representation to ensure that relocated instructions or data remain semantically correct and consistent.

The *expander* reorders the instruction blocks, inserting additional jumps to preserve the logical flow of the program. More importantly, it replaces specific instruction sequences with one or more equivalent statements, guided by a set of predefined rules. For instance, a jump instruction can be transformed into a push followed by a return, maintaining semantic equivalence but altering the structure of the binary. Each type of statement is associated with transformation rules identified by the opcode mnemonic or the behaviour of the instruction. Because certain transformations only apply when particular conditions hold, some rules must be carefully chosen, such as a move↔push/pop replacement that only works when the source operand is a register.

Since applying these transformations repeatedly could cause unlimited binary growth, the *shrinker* component then reverts some of the changes introduced in previous metamorphic steps. By restoring certain instructions to their original form, the shrinker maintains a balance between complexity and size, ensuring that the binary remains manageable while still varying enough to prevent easy pattern recognition.

Finally, the *assembler* reconstructs a functional ELF file from the instructions produced by the expander and the shrinker. It uses Keystone [40] to generate valid machine code and updates the section and program headers, reassigning the entry point and ensuring that all references between instructions and data remain correct. This process yields a binary that preserves its original functionality, but has been rewritten in such a way that it can resist static analysis and code-reconstruction attempts.

3.2.2. Guest Metamorphism

Because the program is interpreted by a custom interpreter, it is possible to manipulate both the ISA and its semantics during guest metamorphism. Here, metamorphism involves modifying not only the program array but also the memory array. Within the program array, the opcode-to-handler mapping can be completely changed. The set of valid opcode bits illustrated in Figure 2 and any padding values are fully remapped, including both the number of padding bytes embedded within each instruction and the values of those bytes. Consequently, the ISA employed by VMorph varies from execution to execution. The reassignment of opcodes obscures the type of each instruction, and variations in padding bytes mask operand types and values.

In the memory array, the objective is to conceal the guest program's data by introducing arbitrary

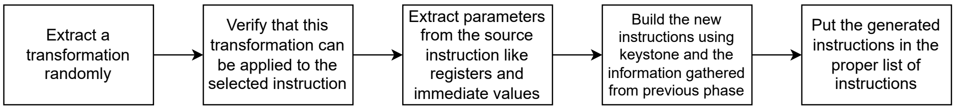

Extract a transformation randomly
Verify that this transformation can be applied to the selected instruction
Extract parameters from the source instruction like registers and immediate values
Build the new instructions using keystone and the information gathered from previous phase
Put the generated instructions in the proper list of instructions

Figure 4: Expansion scheme for host metamorphism.

amounts of padding bytes before each piece of actual data. Although the addresses used by the instructions remain unchanged, it becomes impossible to distinguish real data from garbage. To ensure that the virtual machine can still resolve correct locations, a simple symbol table indexes addresses and variable sizes. This table is an array of pointers to unsigned long values, with one entry per memory address. An entry contains a valid memory area only if a variable starts at that address. In such cases, the first 32 bits indicate the number of padding bytes preceding the variable, and the last 32 bits indicate the variable's size. During interpretation, any memory access instruction is resolved by converting the virtual address to a physical address, adding the padding bytes specified by the symbol table. During metamorphic transformations, this table identifies variables and their lengths, and the amount of padding is adjusted based on how many variables are inserted.

These modifications are not visible while the program is loaded in memory. During metamorphosis, only the program's disk image is modified. The application that uses VMorph must rewrite itself on disk to persist these transformations. This presents a challenge, since the operating system denies write permissions to a running binary. To address this issue, VMorph uses a separate hook executable that rewrites the binary file. The interpreter constructs the new version of the virtualised program in a shared memory buffer. At the end of the metamorphic transformation, the hook is triggered, waits for the interpreter to exit, and then acquires write access to the executable file to finalise the updated version.

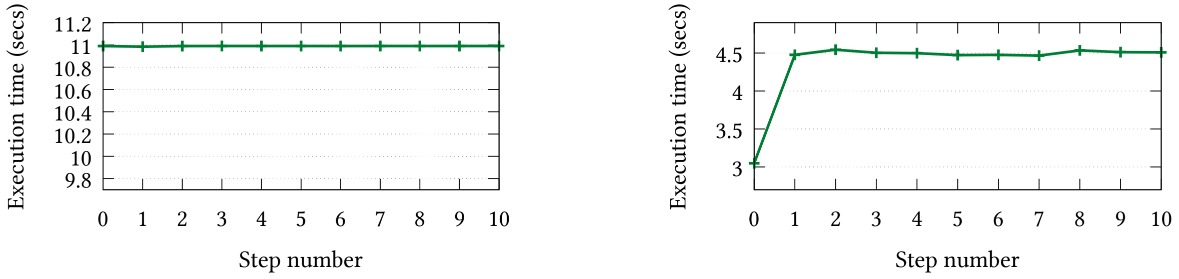
3.3. Detection and Countermeasures

VMorph can be leveraged to obfuscate and metamorphose malicious code, evading both static and dynamic analysis methods. Even the most sophisticated static checks, such as those proposed in [41], are unable to detect the transformed payload produced by VMorph, which alters both its semantics and its representation. Although an adversary may gain advantage by adopting VMorph, modern approaches to counter such threats can rely on dynamic analysis, often employing sandboxing mechanisms. Since a VMorph-obfuscated application executes its logic through an interpreter, heuristic-based measures involving system call tracing or other run-time behaviours can label the running instance as malicious.

A further strategy to detect metamorphosis is to identify recurring patterns that point to transformation activities, perhaps by integrating static and dynamic observations to ascertain the characteristics of the current ISA and the manner in which the emulator is generated before metamorphosis takes place. Several studies have introduced a variety of static analysis approaches, for instance the "Smart Scanning" method presented in [42], which removes redundant instructions to apply signatures more effectively. However, such methods fail against VMorph due to the complexity and breadth of its transformations. Another method described in [43] relies on pattern matching between multiple versions of the same file, deriving signatures from system calls and library functions, and quarantining programs with scores exceeding a given threshold. Similarly, the work presented in [44] proposes static analysis aided by execution traces. Enhancing these latter approaches to include dynamic analysis, possibly before any metamorphosis is performed, could produce detailed insight into the current ISA and the emulator's structure. Statistical [45] and machine learning techniques [46] have also shown promise, allowing classifiers to identify and flag metamorphosed malware samples as malicious.

4. Experimental Assessment

We conducted an experimental assessment of VMorph to evaluate its performance, the correctness of virtualized programs, and its ability to conceal the implementation of the original application to reduce the risk of reverse engineering. We isolated different parts of VMorph in the experiments, to better understand their effects on the overall architecture.



(a) Performance of mthtest2.

(b) Performanc of Pagerank.

Figure 5: Metamorphism Performance Tests

4.1. Metamorphic Transformation Performance

To measure the impact of metamorphic transformations, experiments have been carried out using two compute intensive benchmark applications, namely *mthtest2* [47], which calculates the position of the moon, and a simplified version of Google's *pagerank* algorithm [48]. Both applications were evaluated in their original binary forms and subsequently metamorphosed multiple times to assess changes in their structure after repeated transformations. The tests were run on a machine equipped with an Intel Core i5 11^{th} generation CPU and 16 GB of DDR4 memory, with a Linux operating system using kernel version 6.7.4.

Metamorphic transformations are inherently random and may produce negligible or counterproductive modifications. To ensure meaningful results, a sampling methodology was defined. One hundred metamorphic transformations were applied to each benchmark. Starting from the initial binary, ten different versions were generated, collectively referred to as *step 1*. Each version produced in step 1 was then metamorphosed again to yield ten new variants, known as *step 2*. This operation was repeated until reaching step 10. For each step, a unique seed (yet different from other steps) was used to guide the metamorphic transformations, increasing the probability that the newly generated binaries would differ, even if only slightly, from one another. The execution time for each step was derived by computing the average execution time of five runs for each variant at that step, and then averaging the results across all ten variants. This procedure was followed for each of the ten steps.

Figures 5a and 5b report the results. The *mthtest2* benchmark exhibits negligible impact from metamorphism, likely due to its limited code size, which restricts the scope and complexity of the transformations. In contrast, *pagerank* shows a substantial increase in execution time, nearly doubling after the first set of transformations. In subsequent steps, execution times remain roughly constant, confirming the efficacy of the shrinking module that prevents unchecked code bloat.

4.2. Evaluation of Obfuscation Capabilities

The obfuscation capabilities were evaluated using two benchmark applications, the Prime Count Benchmark (PCB) and an RSA-based test, alongside two malware samples (2d8e89* and 4b060a*) obtained from a publicly available repository [49]. PCB identifies prime numbers in the range $[a, b \cdot a]$, where a and b are provided as command-line parameters; its performance scales with increasing b. The RSA benchmark encrypts and decrypts ten randomly generated integers, relying on a public key (e, N) and a private key (d), with execution time measured while varying N and adjusting the keys accordingly. These tests were compared against other obfuscation tools, including UPX [50], VMProtect [51], and ShiftyLoader [52].

We used the program's entropy to measure their obfuscation level. This metric, a statistical measure of byte distribution, allows quick recognition of packed or encrypted samples [53]. The ENT tool [54] was used to compute two indicators from the generated executables. The first is the binary program entropy, expressed in bits per character. A value near 8 suggests random, high-density information. The second is the arithmetic mean value of the file's bytes, computed by summing all byte values and

Filename	Method	Entropy	Arithmetic mean value
PCB	plain	1.373965	10.6083
PCB	VMorph	5.600081	58.3275
PCB	UPX	7.262627	97.0260
PCB	VMProtect	7.797149	117.7580
PCB	ShiftyLoader	5.613139	74.6742
RSA	plain	1.674976	14.3756
RSA	VMorph	5.603559	58.3330
RSA	UPX	7.235762	99.4116
RSA	VMProtect	7.827722	120.0439
RSA	ShiftyLoader	5.583325	75.1930
2d8e89*	plain	1.607687	12.9348
2d8e89*	VMorph	5.659131	59.2632
2d8e89*	UPX	7.299064	97.9632
2d8e89*	VMProtect	7.772334	116.5230
2d8e89*	ShiftyLoader	5.524642	71.1724
4b060a*	plain	3.369404	31.0812
4b060a*	VMorph	5.301009	54.0427
4b060a*	UPX	6.761425	86.0645
4b060a*	VMProtect	7.761307	115.8346
4b060a*	ShiftyLoader	5.870197	80.2429

Table 1: Entropy and arithmetic mean value.

Sample	# AV Tested	# Detected	Detection Rate
4b060a*	63	34	0.45
4b060a* (VMorph)	63	5	0.07
4b060a* (UPX)	63	10	0.13
4b060a* (VMPROTECT)	63	0	0.00
2d8e89*	63	32	0.42
2d8e89* (VMorph)	63	0	0.00
2d8e89* (UPX)	63	5	0.07
2d8e89* (VMPROTECT)	63	0	0.00

Table 2: *VirusTotal* results.

dividing by the file length. When data are close to random, this value approaches 127.5, reflecting a uniform byte distribution [55].

Table 1 presents the results for the selected benchmarks. The values show that VMorph's transformations do not substantially alter entropy or arithmetic mean. In contrast, binaries protected with UPX or VMProtect reach entropy levels near 8 and display arithmetic means close to 100 or 120, respectively, indicating that obfuscation is indeed taking place. Conversely, VMorph yields values significantly lower than those produced by other obfuscation techniques, thereby preventing a direct correlation between its transformations and the presence of a virtualization layer. ShiftyLoader behaves similarly to VMorph in terms of entropy, but its arithmetic mean remains closer to that produced by UPX. Overall, these observations suggest that VMorph can effectively conceal virtualization and produce obfuscation patterns that are not easily detected by entropy-based analysis.

To complete the evaluation, the detection rates of commonly available static and dynamic detection tools were assessed for both unobfuscated and obfuscated malware samples. This analysis was conducted using *VirusTotal* [56]. For these tests, no samples protected with ShiftyLoader were included due to the authors' restrictions on public disclosure. The comparison involved the original binaries and their obfuscated counterparts produced using different solutions.

Table 2 reports the detection results. The artefact 4b060a* shows a reduction in detection rate of approximately 40% after virtualization through VMorph. This version's detection rate is 0.07, compared to 0.45 for the native binary. Nonetheless, a few antivirus scanners can still recognise the obfuscated sample. This persistence of detection is likely related to the presence of strings unchanged from the original binary, suggesting that certain signatures rely on these strings to identify malicious content. Conversely, detection for the artefact 2d8e89* is reduced to zero once it is processed by VMorph. Since this shellcode-based sample has no explicit strings in memory, virtualization disrupts any signature that might have been anchored to identifiable patterns in the code.

Some antivirus engines remain effective against binaries protected using UPX, probably due to the packer's well-known strategy. This is evident in non-zero detection rates for both artefacts obfuscated with UPX. On the other hand, binaries protected through VMProtect have a detection rate of zero in both cases, presumably because its encryption disrupts string-based detection entirely.

To further highlight these trends, five antivirus solutions were selected from those with the strongest detection capabilities and the lowest decline in detection rates when facing obfuscated samples. Figure 6 shows that most of these solutions fail to identify malware that has been processed with VMorph as malicious. These findings indicate that VMorph's transformations can significantly hinder the automated classification methods employed by modern antivirus systems, thus extending the obfuscation capabilities of transformed binaries beyond those offered by existing solutions.

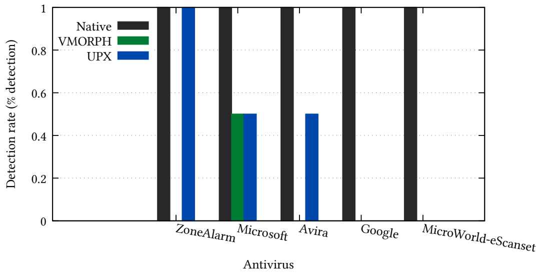


Figure 6: *VirusTotal* Best 5 Antivirus Detection Rate (both Native and Obfuscated).

5. Conclusions and Future Work

We have presented VMorph, a framework that can protect binaries from source code reconstruction using virtualization and metamorphic techniques. Our experiments showed a performance degradation comparable to that of other existing solutions, but improved obfuscation capabilities. Our future plans include developing a back-end LLVM compiler, supporting multithreading, and porting VMorph to other operating systems.

Acknowledgments

This work was partially funded by the project I-Nest (G.A. 101083398—CUP B97H22004950001)—Italian National hub Enabling and enhancing networked applications and Services for digitally Transforming Small, Medium Enterprises and Public Administration.

Declaration on Generative AI

During the preparation of this work, the authors used Writefull in order to: Grammar and spelling check. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] G. Wróblewski, General Method of Program Code Obfuscation, Ph.D. thesis, Wrocław University of Science and Technology, 2002.

[2] S. Drape, Intellectual property protection using obfuscation, Technical Report CS-RR-10-02, Oxford University Computing Laboratory, 2011.

[3] Y. Guillot, A. Gazet, Automatic binary deobfuscation, Journal in computer virology 6 (2010) 261–276. doi:10.1007/s11416-009-0126-4.

[4] PreEmptive Solutions, LLC, The unofficial R8 documentation, https://r8-docs.preemptive.com/, 2018. Accessed: 2023-9-4.

[5] Z. Dong, H. Liu, L. Wang, X. Luo, Y. Guo, G. Xu, X. Xiao, H. Wang, What did you pack in my app? a systematic analysis of commercial android packers, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Association for Computing Machinery, New York, NY, USA, 2022, pp. 1430–1440. doi:10.1145/3540250.3558969.

[6] S. A. Sebastian, S. Malgaonkar, P. Shah, M. Kapoor, T. Parekhji, A study & review on code obfuscation, in: Proceedings of the 2016 World Conference on Futuristic Trends in Research and Innovation for Social Welfare, IEEE, Piscataway, NJ, USA, 2016, pp. 1–6. doi:10.1109/startup.2016.7583913.

[7] D. Javaheri, P. Lalbakhsh, M. Hosseinzadeh, A novel method for detecting future generations of targeted and metamorphic malware based on genetic algorithm, IEEE access: practical innovations, open solutions 9 (2021) 69951–69970. doi:10.1109/access.2021.3077295.

[8] Z. Mumtaz, M. Afzal, W. Iqbal, W. Aman, N. Iltaf, Enhanced metamorphic techniques-a case study against havex malware, IEEE access: practical innovations, open solutions 9 (2021) 112069–112080. doi:10.1109/access.2021.3102073.

[9] M. Sikorski, Practical Malware Analysis: The hands-on guide to dissecting malicious software, No Starch Press, San Francisco, CA, 2012.

[10] K. Kaushik, H. S. Sandhu, N. K. Gupta, N. Sharma, R. Tanwar, A systematic approach for evading antiviruses using malware obfuscation, in: Marriwala N., Tripathi C.C., Jain S., Mathapathi S. (Eds.), International Conference on Emergent Converging Technologies and Biomedical Systems, ETBS 2021, volume 841, Springer Science and Business Media Deutschland GmbH, Berlin, Germany, 2022, pp. 29–37. doi:10.1007/978-981-16-8774-7_3.

[11] P. Ször, The art of computer virus research and defense, Addison-Wesley Professional, Boston, MA, USA, 2005.

[12] M.-J. Kim, J.-Y. Lee, H.-Y. Chang, S. Cho, Y. Park, M. Park, P. A. Wilsey, Design and performance evaluation of binary code packing for protecting embedded software against reverse engineering, in: 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, IEEE, Madeira Island, Portugal, 2010, pp. 80–86.

[13] C. Collberg, C. Thomborson, D. Low, A taxonomy of obfuscating transformations, Technical Report 148, Department of Computer Science, The University of Auckland, 1997.

[14] J. Hwang, T. Han, Identifying Input-Dependent jumps from obfuscated execution using dynamic data flow graphs, in: Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop, number Article 3 in SSPREW-8, Association for Computing Machinery, New York, NY, USA, 2018, pp. 1–12.

[15] M. Sharif, A. Lanzi, J. Giffin, W. Lee, Automatic reverse engineering of malware emulators, in: Proceedings of the 30th IEEE Symposium on Security and Privacy, IEEE, Piscataway, NJ, USA, 2009, pp. 94–109. doi:10.1109/sp.2009.27.

[16] J. H. Anderson, Y.-J. Kim, GDB: The GNU project debugger, Distributed Computing 14 (2001) 17–29. doi:10.1007/PL00008923.

[17] NSA, Ghidra - a software reverse engineering (SRE) suite of tools developed by NSA's research directorate in support of the cybersecurity mission, https://ghidra-sre.org/, 2019.

[18] H. Rays, Hex rays - state-of-the-art binary code analysis solutions, 1997.

[19] J.-M. Borello, L. Mé, Code obfuscation techniques for metamorphic viruses, Journal in Computer Virology 4 (2008) 211–220. doi:10.1007/s11416-008-0084-2.

[20] J.-M. Borello, Étude du métamorphisme viral: modélisation, conception et détection, Ph.D. thesis, Université Rennes 1, Rennes, France, 2011.

[21] Nikitchenko M., Spivakovsky A., Ermolayev V., Bassiliades N., Kharchenko V., Shyshkina M., Peschanenko V., Mayr H.C., Fill H.-G., Yakovyna V. (Eds.), Metamorphic viruses' detection technique based on the equivalent functional block search, volume 1844, CEUR-WS, 2017.

[22] A. G. Kakisim, M. Nar, I. Sogukpinar, Metamorphic malware identification using engine-specific patterns based on co-opcode graphs, Computer standards & interfaces 71 (2020) 103443. doi:10.1016/j.csi.2020.103443.

[23] P. Borrello, E. Coppa, D. C. D'Elia, Hiding in the particles: When return-oriented programming meets program obfuscation, in: Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), DSN '21, IEEE, Piscataway, NJ, USA, 2021, pp. 555–568. doi:10.1109/dsn48987.2021.00064.

[24] W. Mahoney, J. Franco, G. Hoff, J. T. McDonald, Leave it to weaver, in: Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop, SSPREW '18, ACM, New York, NY, USA, 2018, pp. 1–9. doi:10.1145/3289239.3291459.

[25] J. Kinder, Towards static analysis of virtualization-obfuscated binaries, in: Proceedings of the 19th Working Conference on Reverse Engineering, WCRE '12, IEEE, Piscataway, NJ, USA, 2012, pp.

61–70. doi:10.1109/wcre.2012.16.

[26] J. H. Suk, D. H. Lee, VCF: Virtual code folding to enhance virtualization obfuscation, IEEE Access 8 (2020) 139161–139175. doi:10.1109/ACCESS.2020.3012684.

[27] B. Lee, Y. Kim, J. Kim, binOb+: a framework for potent and stealthy binary obfuscation, in: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS '10, Association for Computing Machinery, New York, NY, USA, 2010, pp. 271–281. doi:10.1145/1755688.1755722.

[28] Z. Wu, S. Gianvecchio, M. Xie, H. Wang, Mimimorphism: a new approach to binary code obfuscation, in: Proceedings of the 17th ACM conference on Computer and communications security, CCS '10, Association for Computing Machinery, New York, NY, USA, 2010, pp. 536–546. doi:10.1145/1866307.1866368.

[29] J. Oberheide, M. Bailey, F. Jahanian, PolyPack: an automated online packing service for optimal antivirus evasion, in: Proceedings of the 3rd USENIX conference on Offensive technologies, WOOT'09, USENIX Association, USA, 2009, p. 9.

[30] G. Bernardinetti, D. Di Cristofaro, G. Bianchi, PEzoNG: Advanced packer for automated evasion on windows, Journal of computer virology and hacking techniques 18 (2022) 315–331. doi:10.1007/s11416-022-00417-2.

[31] M. Gaudesi, A. Marcelli, E. Sanchez, G. Squillero, A. Tonda, Malware obfuscation through evolutionary packers, in: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO Companion '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 757–758. doi:10.1145/2739482.2764940.

[32] The Mental Driller, Metamorphism in practice or How I made MetaPHOR and what I've learnt, Technical Report 29A 6, 29A, 2002.

[33] P. Ször, P. Ferrie, Hunting For Metamorphic, Technical Report, Symantec, 2001.

[34] A. Pellegrini, Hijacker: Efficient static software instrumentation with applications in high performance computing, in: Proceedings of the 2013 International Conference on High Performance Computing & Simulation (HPCS), IEEE, Piscataway, NJ, USA, 2013, pp. 650–655. doi:10.1109/hpcsim.2013.6641486.

[35] V. Bala, E. Duesterwald, S. Banerjia, Dynamo: a transparent dynamic optimization system, SIGPLAN Notices 35 (2000) 1–12. doi:10.1145/358438.349303.

[36] C.-K. Luk, B. C. Ed, F. C. G. Hi, E. D. Q. Rs, A. Tu, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation - PLDI '05 40 (2005) 190. doi:10.1145/1065010.1065034.

[37] J.-Y. Lee, J. H. Suk, D. H. Lee, VODKA: Virtualization obfuscation using dynamic key approach, in: Kang B.B., Jang J. (Eds.), Information Security Applications, volume 11402 LNCS of *Lecture notes in computer science*, Springer International Publishing, Cham, 2019, pp. 131–145. doi:10.1007/978-3-030-17982-3_11.

[38] D. Chisnall, How to design an isa, Commun. ACM 67 (2024) 60–66. URL: https://doi.org/10.1145/3640538. doi:10.1145/3640538.

[39] System V Application Binary Interface AMD64 Architecture Processor Supplement, 2006.

[40] Keystone, Keystone, 2016.

[41] P. Caporaso, G. Bianchi, F. Quaglia, JITScanner: Just-in-Time executable page check in the linux operating system, in: Proceedings of the 18th International Conference on Availability, Reliability and Security, number Article 78 in ARES '23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 1–8.

[42] B. B. Rad, M. Masrom, S. Ibrahim, Evolution of computer virus concealment and Anti-Virus techniques: A short survey, arXiv [cs.CR] 8 (2011) 8.

[43] Q. Zhang, D. S. Reeves, MetaAware: Identifying metamorphic malware, in: Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), IEEE, San Diego, California, 2007, pp. 411–420.

[44] A. Kalysch, J. Götzfried, T. Müller, VMAttack: Deobfuscating Virtualization-Based packed binaries,

in: Proceedings of the 12th International Conference on Availability, Reliability and Security, number Article 2 in ARES '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 1–10.

[45] A. H. Toderici, M. Stamp, Chi-squared distance and metamorphic virus detection, Journal of Computer Virology and Hacking Techniques 9 (2013) 1–14.

[46] R. K. Jidigam, T. H. Austin, M. Stamp, Singular value decomposition and metamorphic detection, Journal of Computer Virology and Hacking Techniques 11 (2015) 203–216.

[47] P. S. Paulrho, mthtest2: maths benchmark test (numerical position of moon), 2017.

[48] A. Carrara, pagerank: C implementation of google's PageRank algorithm, 2022.

[49] M. Malware, Linux Malware Samples - a collection of linux malware binaries, https://github.com/MalwareSamples/Linux-Malware-Samples, 2021. Accessed: 2023-NA-NA.

[50] UPX Team, UPX, https://upx.github.io/, 1996.

[51] VMProtect Software, VMProtect software, https://vmpsoft.com/, 2005. Accessed: 2024-NA-NA.

[52] M. Salvatori, G. Bernardinetti, F. Quaglia, G. Bianchi, ShiftyLoader: Syscall-free reflective code injection in the linux operating system, in: Proceedings of the 2024 Italian Conference on Cybersecurity, ITASEC '24, CEUR-WS.org, Aachen, Germany, 2024, pp. 1–14.

[53] R. Lyda, J. Hamrock, Using entropy analysis to find encrypted and packed malware, IEEE security & privacy 5 (2007) 40–45. doi:10.1109/msp.2007.48.

[54] J. Walker, Ent program, https://www.unix.com/man-page/linux/1/ent, 1998.

[55] Z. Mengdi, Z. Xiaojuan, Z. Yayun, M. Siwei, Overview of randomness test on cryptographic algorithms, J. Phys. Conf. Ser. 1861 (2021) 012009.

[56] VirusTotal, VirusTotal, 2004.