J Comput Virol Hack Tech (2014) 10:177–187
DOI 10.1007/s11416-013-0194-3

ORIGINAL PAPER

Metamorphic code generation from LLVM bytecode

Teja Tamboli · Thomas H. Austin · Mark Stamp

Received: 3 September 2013 / Accepted: 18 November 2013 / Published online: 30 November 2013

Abstract Metamorphic software changes its internal structure across generations with its functionality remaining unchanged. Metamorphism has been employed by malware writers as a means of evading signature detection and other advanced detection strategies. However, code morphing also has potential security benefits, since it can serve to increase the "genetic diversity" of software. We have created a metamorphic code generator within the LLVM compiler framework. LLVM is a three-phase compiler that supports multiple source languages and target architectures. It uses a common intermediate representation (IR) bytecode in its optimizer. Consequently, any supported high-level programming language is transformed to this IR bytecode as part of the LLVM compilation process. Our metamorphic generator functions at the IR bytecode level, which provides many advantages over morphing at the assembly or source code level. The morphing techniques that we employ include dead code insertion and transposition, where the dead code is actually executed within the morphed code, making its detection and removal more challenging. We have verified the effectiveness of our code morphing using hidden Markov model analysis.

1 Introduction

Software is said to be metamorphic if multiple copies are structurally different, but functionally equivalent. Examples of metamorphic malware generators can be found in [1–4].

To date, metamorphic code generation has primarily been used by malware writers, since well-designed metamorphic code can evade signature-based detection and other advanced detection strategies [3–5]. However, metamorphism also has the potential to provide security benefits by increasing the "genetic diversity" of software, thereby making several types of attacks more difficult and by limiting the damage of successful attacks [6,7].

Many metamorphic malware generators are readily available at [8]. Some notable examples include

- G2 (Second Generation virus generator) [8]
- MPCGEN (Mass Code Generator) [9]
- NGVCK (Next Generation Virus Creation Kit) [9]
- VCL32 (Virus Creation Lab for Win32) [10]
- MetaPHOR [1].

In addition, research morphing engines are presented in [3] and [4]. All of these metamorphic generators work at the assembly language level. Code morphing of high-level source code is far simpler, but much less effective, since such morphing does not provide sufficient control over the resulting executable file.

In this research, we have implemented and analyzed a metamorphic code generator built on the LLVM compiler framework [11,12]. LLVM is a three-phase compiler that supports multiple source languages and multiple target architectures. In the optimization process, code is converted to intermediate representation (IR) bytecode. Our code morphing tool functions at this IR bytecode level, which simplifies many types of morphing (analogous to working at the source code level), but also provides the necessary fine-grained control over the resulting executable (analogous to working at the assembly code level).

Related research involving LLVM IR bytecode manipulation includes a malware encryption technique implemented as optimizer passes [13]. In [14], a "shadow attack" is devel-

T. Tamboli · T. H. Austin · M. Stamp (✉)
Department of Computer Science, San Jose State University, San Jose, CA, USA
e-mail: stamp@cs.sjsu.edu

oped using LLVM. This attack hides system call behavior for the purpose of making behavior-based detection of malware more difficult.

We evaluate our morphing technique using a hidden Markov model (HMM) analysis similar to that in [15], which, in turn, is derived from the HMM-based malware detector analyzed in [5]. This HMM technique has been used as a baseline for comparing other proposed metamorphic detection strategies [3,4,16–19]. This body of work provides a firm basis for analyzing the effectiveness of our morphing approach.

The paper is organized as follows. In Sect. 2.1, we provide relevant background information. Section 3 covers the design and implementation of our metamorphic code generator. Experimental results are analyzed in Sect. 4 while Sect. 5 contains our conclusion and suggestions for future work.

2 Background

In this section we briefly discuss the following topics: malware, metamorphic techniques, the LLVM compiler infrastructure, and hidden Markov models. Each of these topics is relevant to the work presented in Sects. 3 and 4.

2.1 Malware

Malware is software that is designed to perform malicious activity [20]. To date, most development and research into metamorphic code has involved malware. Therefore, we present a brief introduction to metamorphic malware before turning our attention to the general case of metamorphic code generation.

2.1.1 Malware evolution

In this section, we briefly consider the evolution that has led to the development of metamorphic malware. Below, we use the term virus generically to refer to malware.

Since signature detection is the most common anti-virus (AV) technique, virus writers have developed a variety of strategies for evading such detection. Perhaps the simplest method to hide a virus body from static signature detection is to encrypt or pack the executable. For encrypted malware, simple schemes are generally used, such as an XOR of each byte with a fixed value [21], which is equivalent to a simple substitution cipher. The malware writer's goal is to obfuscate the code, so simple encryption schemes suffice. However, decryption code must be included, and that code is not encrypted, which opens the door to signature scanning [22].

To make detection more difficult, malware writers have developed so-called polymorphic code, where the virus body is encrypted (or packed) and the decryption code is morphed between generations. Consequently, there is no fixed signature for the decryptor code, making signature detection far more difficult [21]. However, polymorphic code is subject to detection via emulator—the code will eventually decrypt itself at which point it is subject to standard signature detection [22].

To avoid signature detection by emulation, malware writers have developed "body polymorphic" or metamorphic malware. Metamorphic code changes its internal structure at each generation, without altering its function. Well-designed metamorphic malware will exhibit no common signature and hence there is no need for encryption [22].

2.2 Metamorphic techniques

In this section, we discuss several elementary metamorphic techniques. To date, most hacker-produced metamorphic malware has used only relatively simple morphing strategies. We also mention a relatively sophisticated morphing technique based on formal grammars.

2.2.1 Register swap

Register swapping is one of the simplest code morphing techniques. For example, `PUSH ECX` can be replaced by `PUSH EAX`, provided the `EAX` register is not in use. Note that register swapping does not affect opcode sequences. Furthermore, a wildcard string can be used to overcome register swapping [23].

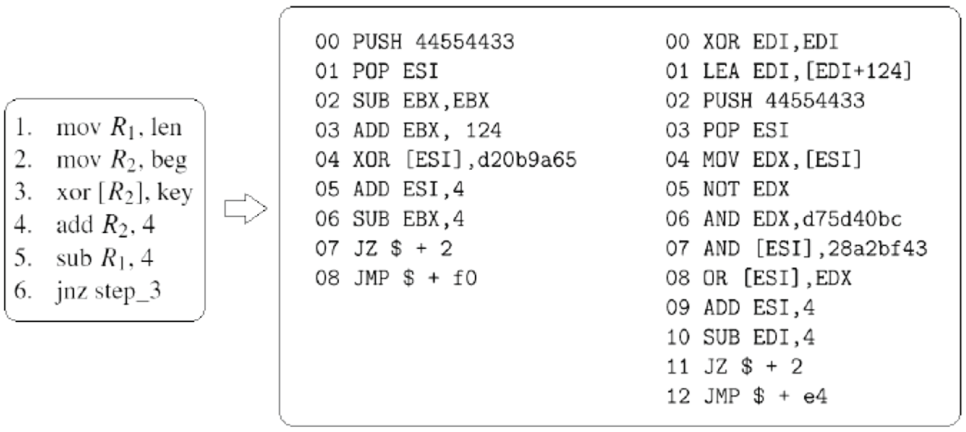
2.2.2 Transposition

Subroutine swapping is another elementary morphing technique. If a program has n subroutines, then $n!$ variants can trivially be generated by simply reordering the layout of the subroutines. As with register swapping, subroutine permutation is a relatively weak malware morphing strategy, particularly with respect to statistical-based detection.

More general transpositions can be used. For example, the instructions

```
1. OPCODE [R1] [R2]
2. OPCODE [R3] [R4]
```

can be swapped, since they are independent of each other. Of course, such transposition can also be applied to group of instructions. Since the order of execution differs, transposition can be an effective means to evade signature detection.



```
1. mov R1, len
2. mov R2, beg
3. xor [R2], key
4. add R2, 4
5. sub R1, 4
6. jnz step_3
```

```
00 PUSH 44554433
01 POP ESI
02 SUB EBX,EBX
03 ADD EBX, 124
04 XOR [ESI],d20b9a65
05 ADD ESI,4
06 SUB EBX,4
07 JZ $ + 2
08 JMP $ + f0
```

```
00 XOR EDI,EDI
01 LEA EDI,[EDI+124]
02 PUSH 44554433
03 POP ESI
04 MOV EDX,[ESI]
05 NOT EDX
06 AND EDX,d75d40bc
07 AND [ESI],28a2bf43
08 OR [ESI],EDX
09 ADD ESI,4
10 SUB EDI,4
11 JZ $ + 2
12 JMP $ + e4
```

Fig. 1 A simple polymorphic decryptor and two variants [26]

2.2.3 Dead code insertion

In its simplest form, dead code is inserted into a program, but not executed. Alternatively, dead code can be executed, provided that it has no effect on the overall program function. Although more difficult, this latter approach can be more effective, since the dead code may be much more difficult to detect.

Dead code can be a highly effective means for evading malware detection, particularly with respect to statistical-based techniques. The dead code can be selected to mask the statistical properties of the underlying code. However, dead code insertion can be challenging at the assembly code level, since care must be taken so that addresses remain valid.

2.2.4 Instruction substitution

An instruction (or group of instructions) can be substituted for another instruction (or group of instructions) with the same functionality. For example, `MOV R1, R2` can be replaced by `PUSH R1` followed by `POP R2`. As another trivial example, `XOR R1, R1` and `SUB R1, R1` both zero the contents of register `R1`. Instruction substitution is a powerful technique for evading signature detection and altering code statistics. However, instruction substitution is relatively difficult to implement at the assembly code level.

2.2.5 Formal grammar mutation

Formal grammar mutation is a formalization of existing morphing techniques [24–26]. Morphing engines can be viewed as non-deterministic automata, since transitions are possible from every symbol (i.e., instruction) to every other symbol [26]. By formalizing mutation techniques, we can apply formal grammar rules to create copies with wide variation. Figure 1 shows a simple polymorphic decryptor template and two possible mutations of the decryptor achieved using the formal grammar in Fig. 2. With this decryptor template and formal grammar combination, it is possible to generate 960 distinct decryptors [26].

$$
\begin{array}{lll}
A & \rightarrow & XB \\
B & \rightarrow & Y_4\varepsilon \\
X & \rightarrow & X_1X_2|X_2X_1 \\
X_1 & \rightarrow & GX_1|mov\ R_1, len|push\ len \oplus pop\ R_1|xor\ R_1, \\
 & & R_1 \oplus lea\ R_1, [R_1 + len]|sub\ R_1, R_1 \oplus add\ R_1, len \\
X_2 & \rightarrow & GX_2|mov\ R_2, beg|push\ beg \oplus pop\ R_2|xor\ R_2, \\
 & & R_2 \oplus lea\ R_2, [R_2 + beg]|sub\ R_2, R_2 \oplus add\ R_2, beg \\
Y_4 & \rightarrow & GY_4|W_1|S_4W_4 \\
W_1 & \rightarrow & GW_1|xor\ [R_2], key\ H_1 \\
W_1 & \rightarrow & not\ [R_2] \oplus xor\ [R_2], key \oplus not[R_2]\ H_1 \\
W_1 & \rightarrow & mov\ R_3, [R_2] \oplus not\ R_3 \oplus and\ R_3, key \oplus and\ [R_2], \\
 & & \neg key \oplus or\ [R_2], R_3\ H_1 \\
H_1 & \rightarrow & GH_1|add\ R_2, 4\ H_2|sub\ R_2, -4\ H_2 \\
S_4 & \rightarrow & GS_1|sub\ R_2, 4|add\ R_2, -4 \\
W_2 & \rightarrow & GW_2|xor\ [R_1][R_2], key\ H_2 \\
W_2 & \rightarrow & not\ [R_1][R_2] \oplus xor\ [R_1][R_2], key \oplus not[R_1][R_2]\ H_2 \\
W_2 & \rightarrow & mov\ R_3, [R_1][R_2] \oplus not\ R_3 \oplus and\ R_3, key \oplus and \\
 & & [R_1][R_2], \neg key \oplus or\ [R_1][R_2], R_3\ H_2 \\
H_2 & \rightarrow & GH_2|sub\ R_1, 4 \oplus jnz\ xxx|sub\ R_1, 4 \oplus jz\ yyy \oplus jmp\ xxx \\
H_2 & \rightarrow & add\ R_1, -4 \oplus jnz\ xxx|add\ R_1, -4 \oplus jz\ yyy \oplus jmp\ xxx \\
H_2 & \rightarrow & sub\ ecx, 3 \oplus loop\ xxx \ \Leftrightarrow\ R_1 \equiv ecx
\end{array}
$$

Fig. 2 Formal grammar for decryptor mutation [26]

2.3 LLVM

LLVM[1] [12] is a compiler infrastructure that has several novel features. LLVM supports a language independent instruction set where each instruction is a static single assign-

[1] "LLVM" was initially derived as an acronym for Low Level Virtual Machine. However, LLVM is now the official name—it is no longer an acronym.

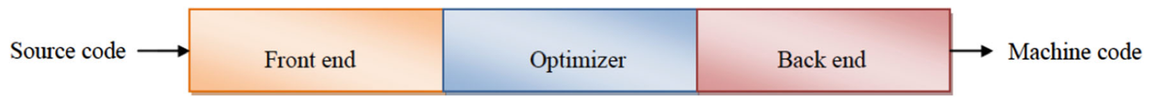


Fig. 3 Three-phase compiler

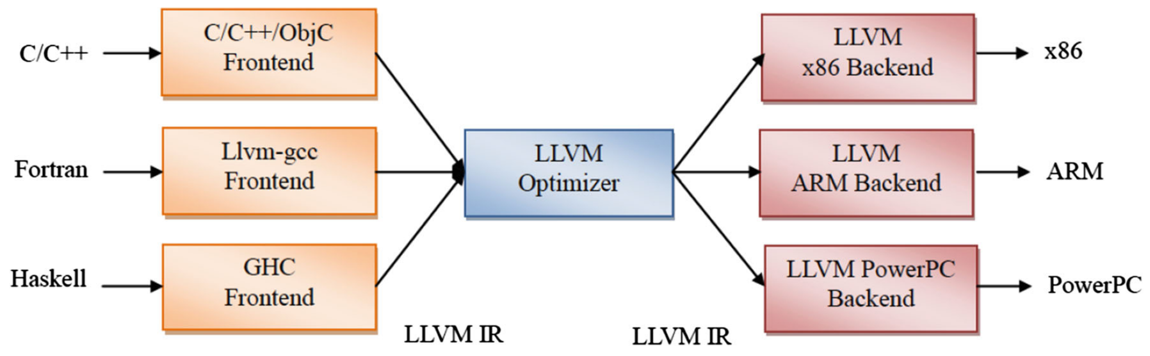


Fig. 4 LLVM design [29]

ment (SSA), which means that each variable is assigned once and then cannot be reassigned [11,27]. Static compilation is supported via late compilation of intermediate representation (IR) bytecode, analogous to the just-in-time (JIT) compiler in Java. The LLVM infrastructure is part of "The Lifelong Code Optimization Project" (LCO-Project) [28].

Most traditional static compilers (e.g., GCC) use three phases, and LLVM follows this approach. These three phases are a frontend, an optimizer, and a backend. Figure 3 illustrates the typical design of a three phase compiler.

The key function of the frontend is to parse the source code, check for syntax errors, and build a language-specific abstract syntax tree (AST). Using the AST, the optimizer manipulates instructions so as to optimize the code. For example, an optimizer removes duplicate code and redundant computations.

The compiler backend generates the machine-dependent representation of the code Backend operations include instruction selection, register allocation, and instruction scheduling [29].

The key feature of the LLVM three-phase compiler design is that it supports multiple frontends and multiple backends, which is greatly simplified by its use of a common intermediate code representation. A frontend can be written for any language. The frontend converts the source code to LLVM IR bytecode which is machine and language independent. A backend can be written for any target platform by generating native code from this common intermediate representation [29,30]. Figure 4 illustrates the LLVM compiler design.

The use of IR bytecode in LLVM effectively separates the frontend and backend components from each other. In addition, the use of IR bytecode supports lightweight runtime optimizations, cross-function or inter-procedural optimizations, program analysis, and aggressive restructuring transformations.

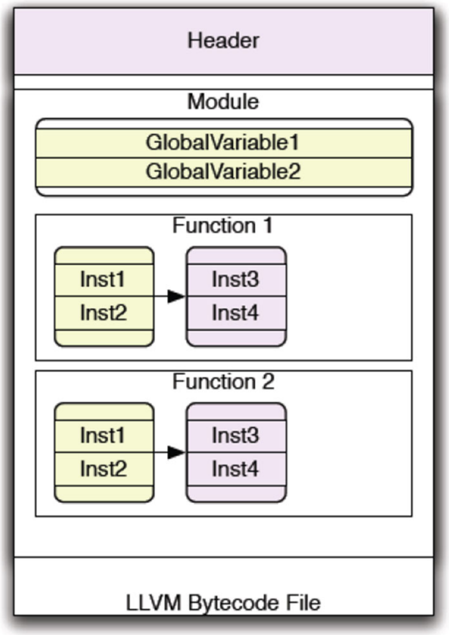
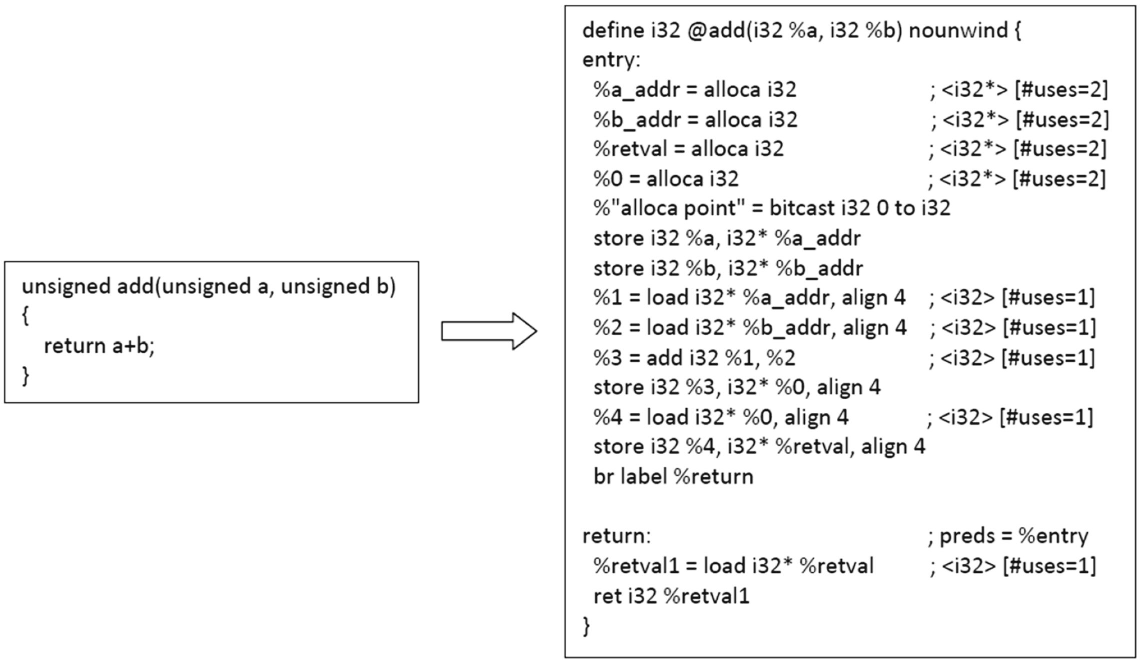


Fig. 5 LLVM bytecode file format [31]

Figure 5 illustrates the structure of LLVM IR bytecode. The following sections are supported [31]:

1. Module—a container that holds functions and global variables
2. Functions—named, callable units of instructions
3. Global variables—variables that can be accessed by any function

Figure 6 shows a simple C function and its corresponding IR representation [32,33].



```
unsigned add(unsigned a, unsigned b)
{
   return a+b;
}
```

```
define i32 @add(i32 %a, i32 %b) nounwind {
entry:
  %a_addr = alloca i32                    ; <i32*> [#uses=2]
  %b_addr = alloca i32                    ; <i32*> [#uses=2]
  %retval = alloca i32                    ; <i32*> [#uses=2]
  %0 = alloca i32                         ; <i32*> [#uses=2]
  %"alloca point" = bitcast i32 0 to i32
  store i32 %a, i32* %a_addr
  store i32 %b, i32* %b_addr
  %1 = load i32* %a_addr, align 4         ; <i32> [#uses=1]
  %2 = load i32* %b_addr, align 4         ; <i32> [#uses=1]
  %3 = add i32 %1, %2                     ; <i32> [#uses=1]
  store i32 %3, i32* %0, align 4
  %4 = load i32* %0, align 4              ; <i32> [#uses=1]
  store i32 %4, i32* %retval, align 4
  br label %return

return:                                   ; preds = %entry
  %retval1 = load i32* %retval            ; <i32> [#uses=1]
  ret i32 %retval1
}
```

Fig. 6 C code and corresponding IR bytecode

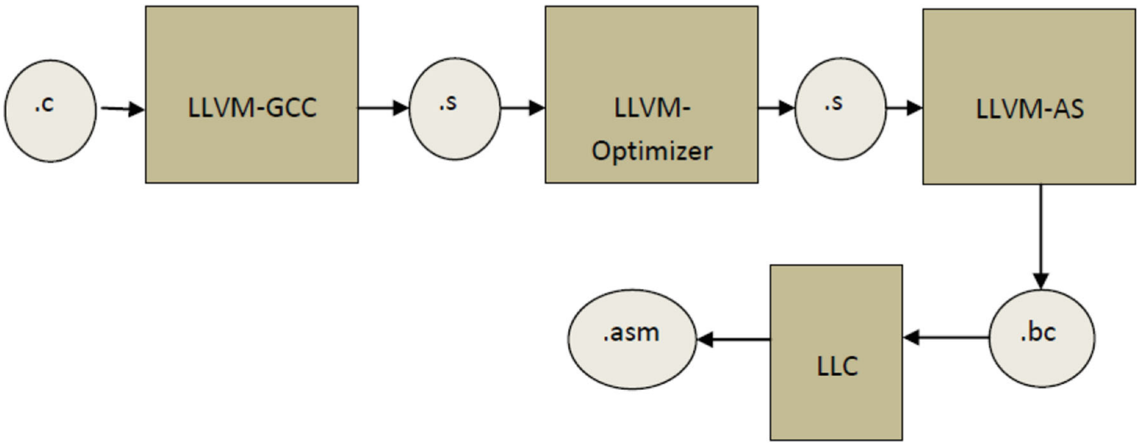


Fig. 7 Program life cycle in LLVM compiler

The program life cycle from source program to executable in LLVM compiler is illustrated in Fig. 7.

In LLVM IR bytecode, the logic is represented in the form of functions, and each function consists of a set of basic blocks. Each basic block consists of a set of instructions and all instructions in a basic block are executed sequentially. A variety of tools are available in the LLVM infrastructure to manipulate IR bytecode.

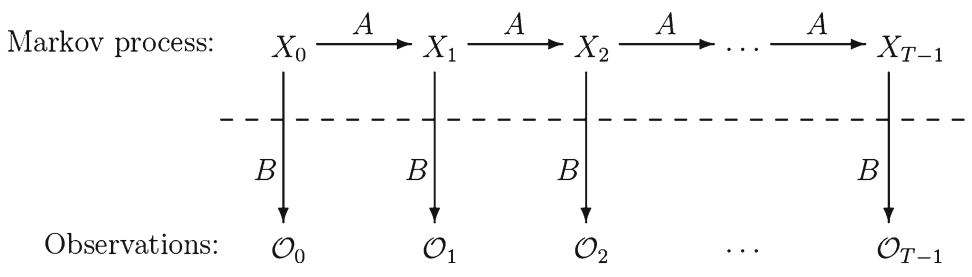
2.4 Hidden Markov models

In this paper, we use hidden Markov models (HMM) as a tool to measure the effectiveness of our morphing strategy. In this section, we provide a very brief introduction to HMMs; see [34] for additional details.

Hidden Markov models can be viewed as a machine learning technique. We can train an HMM to fit a given observation sequence. The resulting model can then be used to score an unknown sequence to measure its similarity to the training data.

As the name suggests, a hidden Markov model includes a “hidden” Markov chain. Although this Markov chain is not directly observable, it is probabilistically related to a sequence of observed symbols. Figure 8 provides a generic illustration of an HMM, where the $\mathcal{O}_i$ are the observations, the matrix A drives the hidden Markov process, and the matrix B contains probability distributions that relate the hidden states to the observations.

Let π be the initial state probability distribution of the underlying Markov process. Then we denote an HMM as $\lambda =$



$$\begin{tikzcd}
X_0 \arrow[r, "A"] \arrow[d, "B"'] & X_1 \arrow[r, "A"] \arrow[d, "B"'] & X_2 \arrow[r, "A"] \arrow[d, "B"'] & \cdots \arrow[r, "A"] & X_{T-1} \arrow[d, "B"'] \\
\mathcal{O}_0 & \mathcal{O}_1 & \mathcal{O}_2 & \cdots & \mathcal{O}_{T-1}
\end{tikzcd}$$

Fig. 8 Generic HMM [34]

(A, B, π). The utility of HMMs derives largely from the fact that there are efficient algorithms to solve each of the following three problems [34].

- Problem 1: Given a model $\lambda = (A, B, \pi)$ and an observation sequence $\mathcal{O}$, we can compute $P(\mathcal{O} \mid \lambda)$. That is, we can score a sequence against a model.
- Problem 2: Given a model $\lambda = (A, B, \pi)$, we can determine an optimal state sequence for the Markov process. That is, we can "uncover" the hidden state sequence.
- Problem 3: Given an observation sequence $\mathcal{O}$, we can determine the model $\lambda = (A, B, \pi)$ that maximizes $P(\mathcal{O})$. That is, we can train a model to fit a given sequence of observations.

In this paper, we first train a model (Problem 3) on opcode sequences derived from a base piece of software. Then we use the trained model to score (Problem 1) morphed versions of this base software. Previous research has shown that HMMs are effective at detecting most metamorphic malware, and that HMMs can also be used to detect certain types of software piracy [15]. That is, HMMs have proven useful at detecting morphed or disguised versions of code. Consequently, HMM analysis provides a challenging test for any code morphing technique.

3 Design and implementation

We have implemented two elementary metamorphic techniques at the LLVM IR bytecode level. Specifically, we use dead code insertion and function permutation. This morphing is available as an LLVM compile-time option.

Code morphing at the IR level offers the following advantages.

- A wide variety of front ends are available in LLVM. The supported languages include Objective-C, FORTRAN, Ada, Haskell, Java bytecode, Python, Ruby, Action Script, GLSL, D, and Rust. Using our tool, code written in any of these language can be morphed.
- The IR form is platform independent.
- At the IR level, virtual addresses are not assigned—addresses are first assigned at the bitcode (i.e., binary executable) level. Therefore, by morphing at the IR level, we avoid one of the major difficulties associated with morphing at the assembly level, namely, dealing with addresses.

Morphed copies of a program must have the same functionality as the base code. In addition, the higher the percentage of inserted or modified code, the more the morphed files should differ (on average) from the base file. In this research, we employ HMM analysis to measure the differences between files. As previously mentioned, HMMs have a proven record of being able to effectively "see through" metamorphic code. Consequently, if we can morph code sufficiently to defeat HMM-based analysis this will provide a strong indication of the success of our morphing strategy.

3.1 Morphing technique

As we are morphing at IR bytecode level, it is difficult to adopt some of the techniques described in Sect. 2.2. For example, register swapping is relatively difficult to implement at the IR level. Therefore, to provide a proof of concept, we have restricted our code morphing to a combination of dead code insertion and subroutine permutation. We accomplish both of these morphing strategies by inserting randomly selected complete subroutines of dead code selected from other program files. In addition, the order of these dead subroutines is randomized. In this way, we create a significant amount of transposition and code variation between different morphed copies. In addition, we insert call statements to all dead code subroutines so that they are not trivially identifiable as dead code.

We have used core-util [35] Linux command files as the source of our dead code subroutines. These files include system level code to do operations that we would expect to be somewhat similar to our selected base code. By selecting morphing code that is similar to our base file, we are creating a more challenging task for our morphing engine, since the goal is to make the morphed code as different as possible from the base code.

The high-level architecture of our morphing engine appears in Fig. 9. Next, we provide a detailed description of each of the three main phases of our morphing engine.

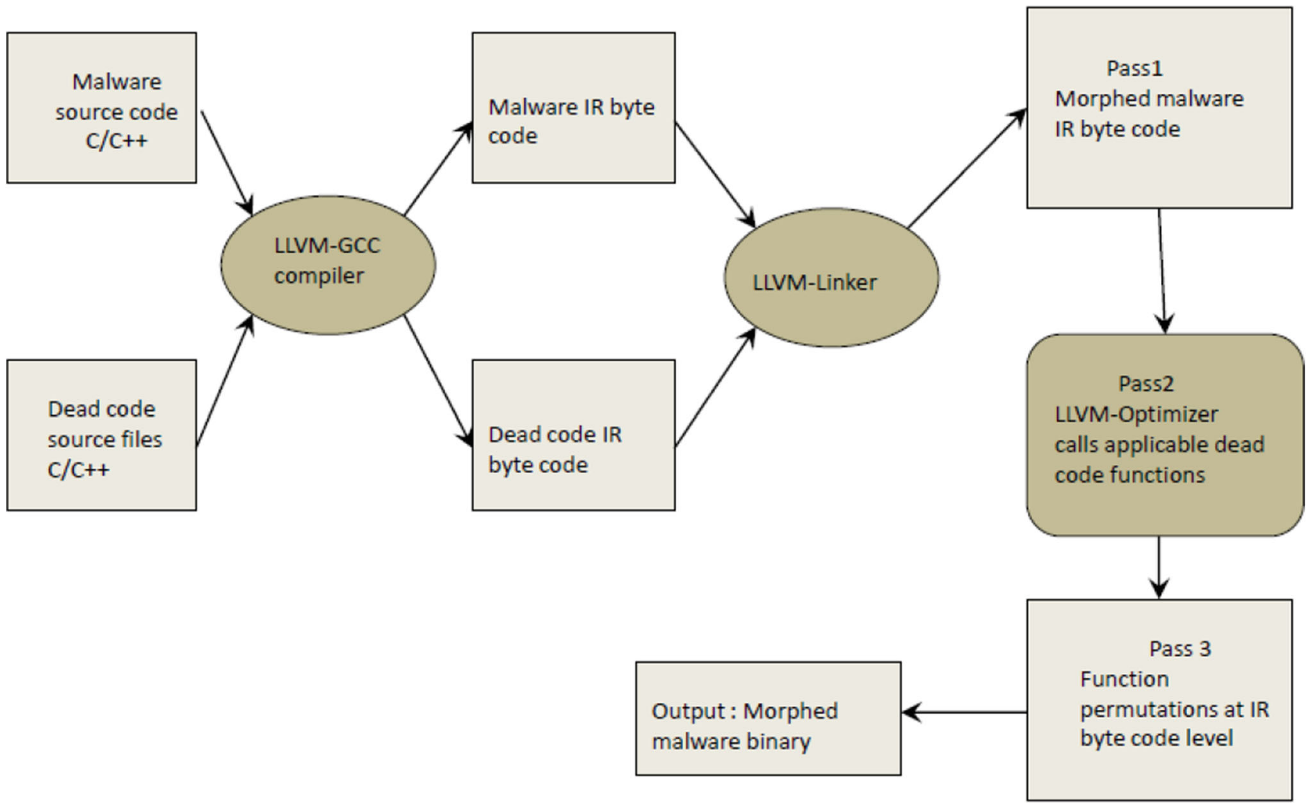

Malware source code C/C++
Dead code source files C/C++
LLVM-GCC compiler
Malware IR byte code
Dead code IR byte code
LLVM-Linker
Pass1 Morphed malware IR byte code
Pass2 LLVM-Optimizer calls applicable dead code functions
Pass 3 Function permutations at IR byte code level
Output : Morphed malware binary

Fig. 9 Metamorphic code generator architecture diagram

3.1.1 Dead code insertion

A base file, a morphing file (i.e., a source of dead code), and a dead code percentage are specified. Based on the dead code percentage, we determine the total number of lines we want to insert into the base file. We then select complete functions from the morphing file so that the total size approximates the number of lines we want to insert into the base file. These subroutines are integrated into the base file at the linking stage. The details of this first phase of our code morphing technique are given below.

1. Compile selected morphing file using the `llvm-gcc` command to generate its IR bytecode.
2. From this IR bytecode, determine function dependencies.
3. For each function, calculate its number of lines.
4. Based on the total number of dead code lines, use a greedy strategy to determine a subset of functions which best approximates the number of lines to be inserted.
5. Copy selected functions to a temporary IR bytecode file.
6. Create bitcode files for the base code and temporary IR bytecode file.
7. Merge these two files (using `llvm-link`).
8. If there are any subroutine naming conflicts, replace each offending name in the temporary IR bytecode file with a random string.
9. Delete the temporary IR bytecode file.

3.1.2 Call dead functions

In this pass, we use the LLVM optimizer to insert a call instruction for each dead code subroutine. The optimizer takes a function name as input. It then finds the `main` function definition in the IR bytecode and inserts a call type of instruction after every load type of instruction. The current implementation does not support structure type of parameters.

For each dead code subroutine, we perform the following steps.

1. Find the "function" object of the `main`.
2. Iterate over instructions in the function object.
3. If an instruction is of type load then insert a call instruction. To insert call instruction for dead function, iterate over its parameters and for each parameter, allocate memory and initialize with a random value.
4. Finally, insert a call instruction.

3.1.3 Function permutation

The third pass performs function permutation by simply reordering functions in the IR bytecode file. This pass is straightforward and we omit the details. Additional details on the entire process can be found in [36].

4 Experimental results

In this section, we use the HMM technique developed in [5] to test the effectiveness of our LLVM-based metamorphic code generator. We add increasing percentages of dead code to find the threshold at which HMM detector starts to fail. We show that after adding about 20 % (or more) dead code, our metamorphic code is not reliably distinguished using this HMM technique. These results indicate that our LLVM-based morphing strategy is more effective than the hacker-produced metamorphic malware generators considered in previous research [5], and is at least as effective as an experimental metamorphic malware generator that was designed specifically to evade HMM-based detection [4].

For the experiments given here, we use spike fuzzer [37] as our base software. Fuzzing is a process of sending malformed data to an application to generate failures or errors in the application [38]. This base code was morphed using our LLVM metamorphic generator and the morphed versions were then analyzed using HMM-based analysis. Spike fuzzer consists of about 6000 lines of assembly code.

For each experiment, we generate 50 morphed copies by inserting dead code from different morphing files. As previously mentioned, the morphing files are randomly selected from coreutil Linux commands files [23].

Once the morphed files are generated, we use an HMM scoring technique similar to that in [15]. Previous research has consistently shown that the number of hidden states in the HMM does not significantly impact the quality of the file classification. Consequently, we only consider HMMs with $N = 2$ hidden states.

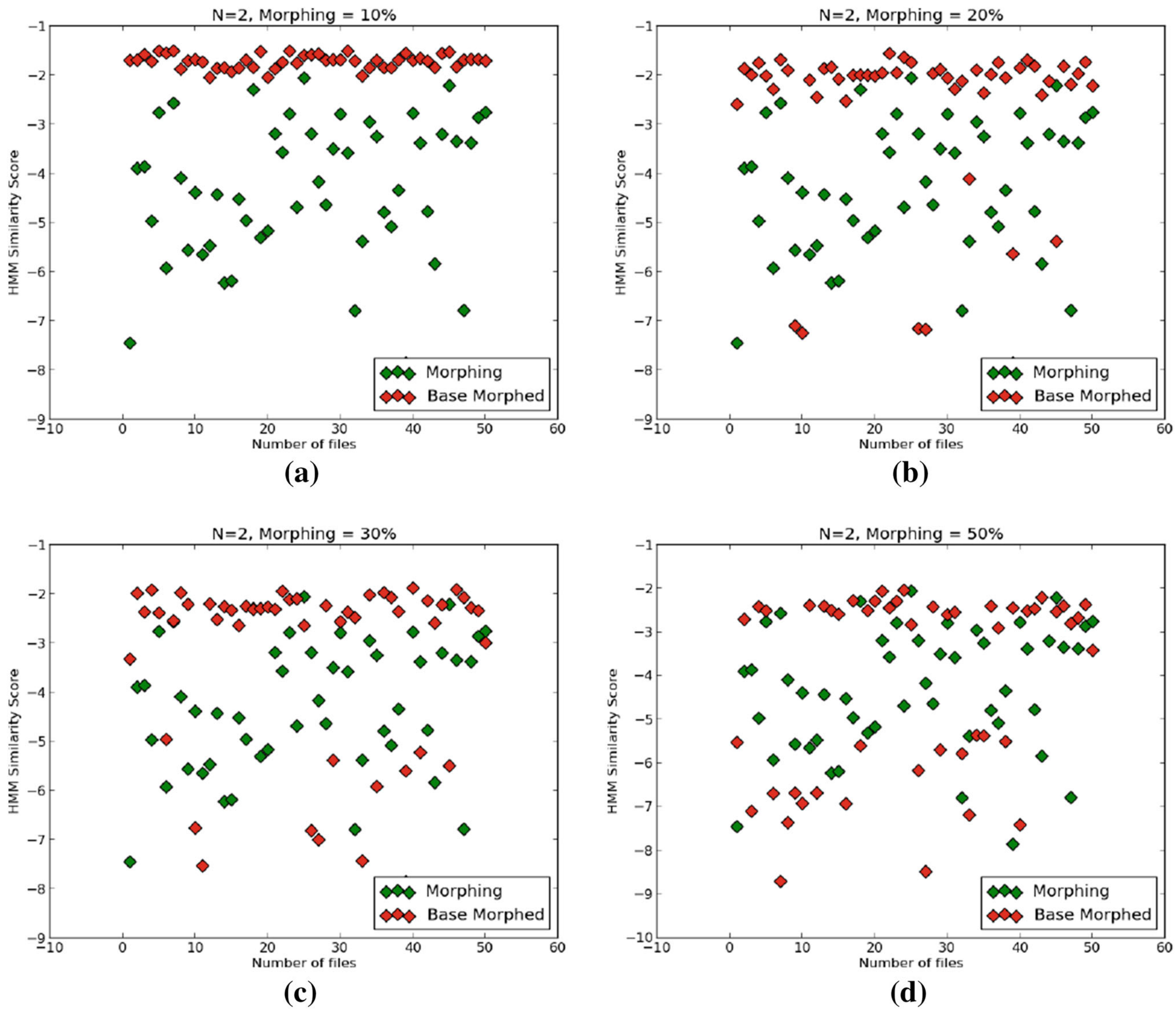


Fig. 10 HMM scores for various morphing percentages. (**a**) 10 % morphing, (**b**) 20 % morphing, (**c**) 30 % morphing, (**d**) 50 % morphing

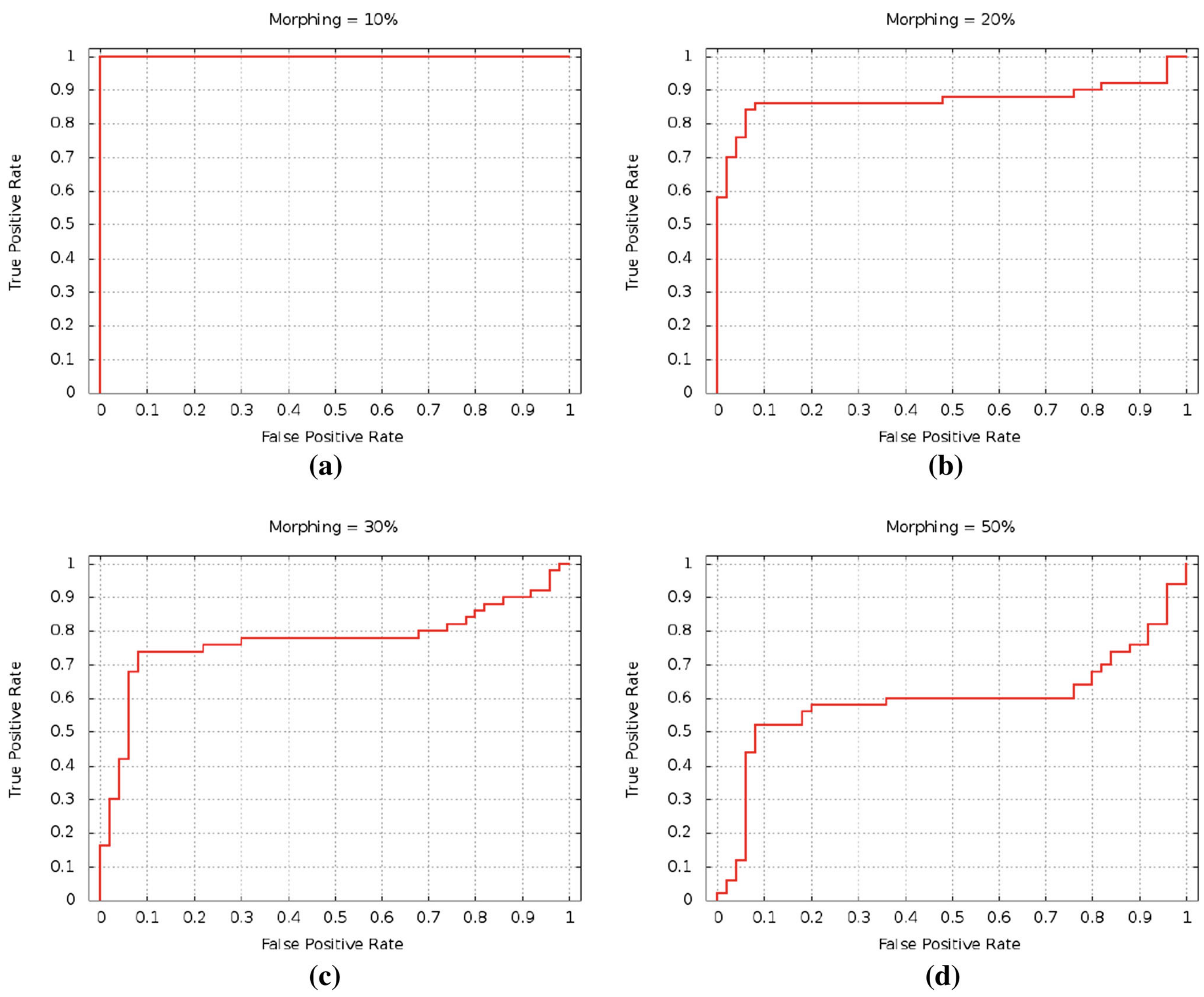


Fig. 11 ROC curves for various morphing percentages. (**a**) 10 % morphing, (**b**) 20 % morphing, (**c**) 30 % morphing, (**d**) 50 % morphing

First, we train an HMM to model the base file. To obtain sufficient observations for training, we generated 50 copies of the base file, each having a 5 % rate of morphing. We then trained an HMM on these 50 morphed files. We refer to this model as the "base HMM." As discussed in [15], the purpose of the slight morphing at this stage is simply to prevent the base HMM from overfitting the available data in the base file. Consequently, we use a minimal amount of morphing at this step.

Next, we use this trained HMM to score 50 morphing files. Specifically, we score the coreutil Linux commands files that we use as our source of morphing code in the experiments described below.

We then conducted experiments where we morph the base file at each of the following rates: 10 %, 20 %, 30 %, and, finally, 50 %. In each case, we generated 50 morphed versions of the base file, with each file morphed at the given rate. These morphed copies were then scored using the base HMM and these scores were compared to the scores obtained for the morphing files as mentioned in the previous paragraph. As the morphing percentage increases, we expect the scores of the morphed files to converge towards the scores of the morphing files. Note that all scores are normalized to a per opcode basis so that file size does not affect the results.

Figure 10 (a) through (d) contain our score results for 10 % 20 %, 30 %, and 50 % morphing, respectively. From these results, we see that after inserting 20 % dead code, the scores are starting to merge, which indicates that the morphed base files are difficult for the HMM to distinguish from the morphing files. This is precisely the effect that we hope to achieve through code morphing.

The results in Fig. 10 are summarized in the form of ROC curves in Fig. 11. These ROC curves plot the false positive rate versus the true positive rate as the threshold is varied throughout the score range.

Table 1 ROC AUC statistic

Dead code insertion %	AUC
10	1.0000
20	0.8708
30	0.7724
50	0.5924

The area under the ROC curve (AUC) is equal to the probability that a classifier ranks a randomly chosen positive instance higher than a randomly chosen negative one [39]. The AUC values for the ROC curves in Fig. 11 are given in the Table 1. Note that an AUC of 1.0 indicates ideal separation (i.e., no false positives or false negatives), while an AUC of 0.5 indicates that the classifier yields results that are no better than flipping a coin. After inserting 20 % dead code, our HMM classifier does poorly, and at higher morphing rates, the rate of classification failure increases dramatically. Again, these results show that our code morphing technique is highly effective, at least with respect to this HMM classifier.

5 Conclusion and future work

In this paper, we presented and analyzed a novel code morphing technique based on LLVM IR bytecode. Our approach makes strong code morphing available as a compile-time option, and requires no special effort on the part of the software developer. As far as the authors are aware, this is the first general purpose code morphing tool of its kind.

Our metamorphic generator uses dead code insertion and function permutation. The dead code is in the form of functions copied from other programs. These dead functions are called within the program, which makes their detection and removal more challenging.

We tested the effectiveness of our code morphing using an HMM technique that has proven successful in metamorphic malware detection and for detection of certain types of software piracy. We verified that our morphing technique is highly effective, in the sense that an HMM cannot effectively distinguish our morphed code from other code, even at relatively low morphing rates.

There are many possible improvements to the metamorphic generator presented here. The dead code insertion could be improved by removing the dependence on complete subroutines—it would be possible to do such insertion at the level of basic blocks. Other powerful morphing techniques, such as instruction substitution, could be included. It would also be interesting to employ formal grammar mutation as a framework for implementing the morphing. Additional user control of morphing (via compile-time flags) would be valuable. Finally, improvements in the LLVM infrastructure itself would serve to make our code morphing techniques more robust. For example, in our current implementation, tools available within the LLVM framework could be used to analyze the morphed bitcode. However, if the LLVM bitcode is converted to, say, a Windows PE file, then the tools within LLVM are no longer directly applicable to such analysis.

References

1. The Mental Driller, Metamorphism in practice or “How I made MetaPHOR and what I’ve learnt” (2002). http://download.adamas.ai/dlbase/Stuff/VX%20Heavens%20Library/vmd01.html
2. An example of metamorphic virus. http://spth.virii.lu/main.html
3. Lin, D., Stamp, M.: Hunting for undetectable metamorphic viruses. J. Comput. Virol. **7**(3), 201–214 (2011)
4. Sridhara, S., Stamp, M.: Metamorphic worm that carries its own morphing engine. J. Comput. Virol. Hacking Tech. **9**(2), 49–58 (2013)
5. Wong, W., Stamp, M.: Hunting for metamorphic engines. J. Comput. Virol. **2**(3), 211–229 (2006)
6. Gao, X., Stamp, M.: Metamorphic software for buffer overflow mitigation. In: Dey, P.P., Amin, M.N. (eds.) Proceedings of 3rd Conference on Computer Science and its Applications. San Diego, California (2005)
7. Stamp, M.: Risks of monoculture, Inside Risks 165. Commun. ACM **47**(3):120 (2004). http://www.csl.sri.com/users/neumann/insiderisks04.html#165
8. Open Malware. http://www.offensivecomputing.net/
9. Virus Construction Kits. http://computervirus.uw.hu/ch07lev1sec7.html
10. Attaluri, S., McGhee, S., Stamp, M.: Profile hidden markov models and metamorphic virus detection. J. Comput. Virol. **5**(2), 151–169 (2009)
11. Lattner, C., Adve, V.: Architecture for a next generation GCC. In: First GCC Annual Developer’s Summit (2003). http://llvm.org/pubs/2003-05-01-GCCSummit2003pres.pdf
12. The LLVM Compiler Infrastructure Project. http://llvm.org/
13. Sharif, M. et al.: Impending Malware Analysis Using Conditional Code Obfuscation. College of Computing, Georgia Institute of Technology. http://cyber4.us/sites/default/files/Impeding%20Malware%20Analysis%20Using%20Conditional%20Code%20Obfuscation-NDSS2008.pdf
14. Ma, W., et al.: Shadow attacks: automatically evading system-call behavior. J. Comput. Virol. **8**(1–2), 1–13 (2012)
15. Kazi, S., Stamp, M.: Hidden Markov models for software piracy detection. Inf. Secur. J. A Glob. Perspect. **22**(3), 140–149 (2013)
16. Baysa, D., Low, R.M., Stamp, M.: Structural entropy and metamorphic malware. J. Comput. Virol. Hacking Tech. **9**(4), 179–192 (2013) (to appear)
17. Runwal, N., Low, R.M., Stamp, M.: Opcode graph similarity and metamorphic detection. J. Comput. Virol. **8**(1–2), 37–52 (2012)
18. Shanmugam, G., Low, R.M., Stamp, M.: Simple substitution distance and metamorphic detection. J. Comput. Virol. Hacking Tech. **9**(3), 159–170 (2013)
19. Toderici, A.H., Stamp, M.: Chi-squared distance and metamorphic virus detection. J. Comput. Virol. Hacking Tech. **9**(1), 1–14 (2013)
20. Panda Security, Virus, worms, trojans and backdoors: other harmful relatives of viruses (2011). http://www.pandasecurity.com/homeusers-cms3/security-info/about-malware/generalconcepts/concept-2.html
21. Aycock, J.: Computer Viruses and Malware. Springer, New York (2006)

22. Filiol, E.: Computer Viruses: From Theory to Applications, vol. 1, pp. 19–38. Birkhäuser (2005)
23. Computer virus creation kit. http://www.informit.com/articles/article.aspx?p=366890&seqNum=6
24. Beaucamps, P.: Advanced metamorphic techniques in computer viruses. In: International Conference on Computer, Electrical, and Systems Science, and Engineering, CESSE'07. Venice, Italy (2007)
25. Filiol, E.: Metamorphism, formal grammars and undecidable code mutation. Int. J. Comput. Sci. **2**, 70–75 (2007)
26. Zbitskiy, P.: Code mutation techniques by means of formal grammars and automatons. J. Comput. Virol. **5**(3), 199–207 (2009)
27. LLVM Programming Manual. http://llvm.org/docs/ProgrammersManual.html
28. The Lifelong Code Optimization Project. http://www-faculty.cs.uiuc.edu/vadve/lcoproject.html
29. LLVM Architecture. http://www.aosabook.org/en/llvm.html
30. Lattner, C., Adve, V.: A compilation framework for lifelong program analysis and transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (2004). http://www.cgo.org/cgo2004/papers/06_76_lattner_c.pdf
31. Praher, J.: A Change Framework Based on the Low Level Virtual Machine Compiler Infrastructure. Thesis Report, Johannes Kepler University (2007). http://llvm.cs.uiuc.edu/pubs/2007-04-PraherMSThesis.pdf
32. LLVM, IR Bytecode Format. http://llvm.org/releases/1.3/docs/BytecodeFormat.html
33. LLVM Helloworld in C. http://projects.prabir.me/compiler/wiki/LLVMHelloworldInC.ashx
34. Stamp, M.: A revealing introduction to hidden Markov models (2012). http://www.cs.sjsu.edu/stamp/RUA/HMM.pdf
35. Linux coreutils source code. http://ftp.gnu.org/gnu/coreutil
36. Tamboli, T.: Metamorphic code generation from LLVM IR bytecode, Master's Project 301 (2013). http://scholarworks.sjsu.edu/etd_projects/301/
37. Spike Fuzzer Source Code. http://www.immunitysec.com/resources-freesoftware.shtml
38. Introduction to fuzzing using spike fuzzer. http://resources.infosecinstitute.com/intro-to-fuzzing/
39. Bradley, A.P.: The use of the area under the ROC curve in the evaluation of machine learning algorithms. Pattern Recognit. **30**, 1145–1159 (1997)