MAIL: Malware Analysis Intermediate Language - A Step Towards Automating and Optimizing Malware Detection

Shahid Alam  
Department of Computer Science, University of Victoria  
3800 Finnerty Road  
Victoria, BC, V8P5C2, Canada  
salam@cs.uvic.ca

R. Nigel Horspool  
Department of Computer Science, University of Victoria  
3800 Finnerty Road  
Victoria, BC, V8P5C2, Canada  
nigelh@cs.uvic.ca

Issa Traore  
Department of Electrical and Computer Engineering, University of Victoria  
3800 Finnerty Road  
Victoria, BC, V8P5C2, Canada  
itraore@ece.uvic.ca

ABSTRACT
Dynamic binary obfuscation or metamorphism is a technique where a malware never keeps the same sequence of opcodes in the memory. Such malware are very difficult to analyse and detect manually even with the help of tools. We need to automate the analysis and detection process of such malware. This paper introduces and presents a new language named MAIL (Malware Analysis Intermediate Language) to automate and optimize this process. MAIL also provides portability for building malware analysis and detection tools. Each MAIL statement is assigned a pattern that can be used to annotate a control flow graph for pattern matching to analyse and detect metamorphic malware. Experimental evaluation of the proposed approach using an existing dataset yields malware detection rate of 93.92% and false positive rate of 3.02%.

Categories and Subject Descriptors
D.3 [Programming Languages]: Language Constructs and Features—Patterns; K.6 [Management of Computing and Information Systems]: Security and Protection—Invasive software

General Terms
Security, Languages

Keywords
Intermediate languages, Static binary analysis, Malware analysis, Malware detection, Control flow graph

1. INTRODUCTION
Detecting whether a given program is a malware is an undecidable problem [14, 27]. Antimalware detection techniques are limited by this theoretical result. Malware writers exploit this limitation to avoid detection.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

$\$IN'13$, November 26-28, 2013, Aksaray, Turkey
Copyright $\copyright$ 2013 ACM 978-1-4503-2498-4/00/10 ...$\$15.00.
morphic malware. Almost all the malware use binaries, instructions that a computer can interpret and execute, to infiltrate a computer system. There are hundreds of different instructions in any assembly language. We need to reduce and simplify these instructions considerably to optimize the static analysis of any assembly program for malware detection. MAIL provides an abstract representation of an assembly program and hence the ability for a tool to automate malware analysis and detection. We want a common intermediate language that can be used with different platforms, so we do not have to perform separate static analysis for each platform. By translating binaries compiled for different platforms to MAIL, a tool can achieve platform independence. Each MAIL statement is annotated with patterns that can be used by a tool to optimize malware analysis and detection.

The rest of the paper is structured as follows. In Section 2, we discuss related research efforts in the development of intermediate languages for malware analysis and detection. In Section 3 we describe in detail the language design and components. In Section 4 we introduce our malware detection approach. In Section 5 we conduct an experiment to assess the properties of MAIL and our proposed malware detection approach. We finally conclude in Section 6.

2. RELATED WORK

Several intermediate languages for malware analysis and detection have been proposed in the literature [6, 39, 38, 24, 4, 12, 19]. This Section discusses the academic and the commercial research efforts in the development of intermediate languages for malware analysis and detection. First we present one of the commercial efforts and then move to the academic efforts. The reasons for selecting these research efforts are: (1) information about them is available publicly; (2) they are well described, i.e. at least part of the syntax and semantics is either described or defined mathematically; (3) they are currently being used in either academic or commercial malware analysis and detection tools.

REIL is an intermediate language that is being used in a commercial reverse engineering tool named BinNavi [18, 35]. Although REIL is not specifically designed for malware analysis, it is used in BinNavi for manual malware analysis and detection. In [32], Sepp et al. proposed an extension of REIL with relational information by translating the flags (an instruction’s side effects) calculations into arithmetic instructions. The extension also helps reduce the size of a REIL program. The core language has a very reduced instruction set and consists of only 17 different instructions, and uses a flat memory model. The native instructions are translated to REIL instructions using a map. Based on the experiments carried out by the authors, on average an original native instruction is translated into approximately 20 REIL instructions. Unknown native instructions are replaced with NOP instructions which may introduce inaccuracies in disassembling. There are no examples in the paper of translating an assembly program into REIL. Furthermore, REIL does not translate FPU, MMX and SSE instructions, privileged instructions like system calls, interrupts and other kernel-level instructions. The reason for not including these instructions is that the authors think that these instructions are not yet used in exploiting security vulnerabilities. REIL cannot translate instructions of the type that address registers by an index, as in PowerPC. REIL cannot handle self-modifying code. The reason for this is that the REIL instructions cannot be overwritten or modified during the interpretation of REIL code.

SAIL is an intermediate language presented in [13] that represents a control flow graph (CFG) [1] of the program under analysis, and is used in a prototype malware detection tool developed by the authors. Each instruction in SAIL is either an assignment statement or a call statement, and becomes a block [1] and a node in the CFG. The operators supported in SAIL are arithmetic, bit-vector, relational and the special memory addressing operator. A node in the CFG contains only a single SAIL instruction, which can make the number of nodes in the CFG extremely large and therefore can make the analysis excessively slow for larger binary programs.

The VINE Intermediate Language (VINE-IL) proposed by Song et al. [33] is the intermediate language of the static analysis framework VINE used in the BitBlaze project. BitBlaze provides an extensible binary analysis platform for security applications. It is not specifically designed for malware detection but for general security applications. BitBlaze is used in the tool Panorama [37] for malware analysis and detection. The authors chose simplicity over efficiency, so VINE first translates a binary to VEX, an intermediate language used in Valgrind [28] (a dynamic binary instrumentation tool) and then to VINE-IL. The reason for not using VEX intermediate language directly, is the presence of implicit side effects in VEX instructions. In VINE-IL, the final translated instructions have all the side effects explicitly exposed as VINE instructions. While exposing all the side effects in VINE-IL may be appropriate for general security applications such as program verification, this may not be efficient for specific security applications such as malware detection. Different platforms have different number and type of flags. Exposing all the side effects makes this approach general but also makes it difficult to maintain platform independence. In contrast to SAIL, side-effects are avoided in MAIL, making the language much simpler and providing ground for efficient malware detection.

In [3], the authors use an intermediate language called CFGO-IL to simplify transformation of a program in the x86 assembly language to a CFG. After translating a binary program to CFGO-IL, the program is optimized to make its structure simpler. The optimizations also remove various malware obfuscations from the program. These optimizations include dead code elimination, removal of unreachable branches, constant folding and removal of fake conditional branches inserted by malware. The side effects of the assembly instructions are exposed explicitly in the instructions of the CFGO-IL. The authors developed a prototype malware detection tool using CFGO-IL that take advantage of the optimizations and the simplicity of the language. However, by exposing all the side effects of an instruction, the language faces the same problem of maintaining the platform independence like VINE-IL. Furthermore, the size of a CFGO-IL program tends to increase compared to the original assembly program.

In [10], Cesare and Xiang introduce a new intermediate language for malware analysis named WIRE. The language
is currently being used in the Malwise tool [11] developed by
the authors. To the best of our knowledge, this is the only re-
search effort that has the same goals as the MAIL language.
The language is formally defined using an incomplete set of
BNF notations. The authors defined operational semantics
of WIRE and provided manual examples to check the seman-
tic equivalence of obfuscated code using these operational
semantics. WIRE does not explicitly specify the indirect
jumps, making malware detection more complicated. There
is only one instruction *ijmp* in WIRE that uses register as
the branch target. The register contents (address) can be
known or unknown and hence can complicate the malware
analysis, and may render an incorrect analysis. To simplify
malware analysis, in MAIL, this information is made explicit
in the instruction.

Furthermore, the authors mention side effects of the as-
sembly instructions as one of the difficulties of using the na-
tive assembly, but do not say anything about the side effects
of the WIRE instructions. It is not clear how the language is
used in the Malwise tool to automate the malware analysis
and detection process. None of the referred papers [9, 8, 7,
11, 10] covers the automation process using WIRE.

3. THE MAIL LANGUAGE

In this Section, we give an outline of the design of MAIL
and introduce underlying elements.

3.1 Language Design

We believe a good language must start small and simple,
and must give opportunities to the language developers to
grow (extend) the language with the users. Therefore MAIL
is designed as a small, simple, and extensible language. In
this and next subsections we describe how MAIL is designed
in detail.

The basic purpose of MAIL is to represent structural and
behavioral information of an assembly program for malware
analysis and detection. MAIL will also make the program
behavioral information non-trivial. Our goal is to create
as few statements as possible in the intermediate language
and map as many instructions as possible to these state-
ments. For example we do not translate (i.e. ignore) the
following x86 instructions:

- CLFLUSH: Flush caches
- CLTS: Clear TLB
- SMSW: Restore machine status word
- VERR: Verify if a segment can be read
- WBINVD: Writing back and flushing of external caches
- XRSTOR: Restore processor extended states from memory
- XSAVE: Save processor extended states from memory

The complete list of x86 instructions that are not trans-
lated into the MAIL statements is given in [2].

3.2 MAIL Statements

The MAIL statements are divided into the following 8
basic statements (the complete MAIL grammar is given in
[2]):

- statements ::= ( statement* ) ;
- statement ::= assignment_s* | control_s*
  | condition_s* | function_s*
  | jump_s* | lib_call_s*
  | 'halt' | 'lock' ;

Every statement in the MAIL language has a type also
called a pattern that can be used for pattern matching dur-
ing malware analysis and detection. These patterns are in-
roduced and explained in Section 3.4. MAIL has its own
registers but also reuses the registers present in the architec-
ture that is being translated to the MAIL language. There
are other special registers such as:

- **Flag registers**: ZF (zero flag), CF (carry flag), PF
  (parity flag), SF (sign flag) and OF (overflow flag).
  These flag registers are of size one byte and are used
  in conditional statements.
  e.g. if (ZF == 1) jmp 0x05632;
- **eflags**: stores the flag registers.
- **sp**: to keep track of the stack pointer.
- **gr and fr**: these are infinite number of general pur-
  pose registers for use in integer and floating point in-
  structions, respectively, and as they are used they are
  appended by a number, such as gr1, gr2, gr3, fr1, fr2,
  and fr3 etc.

The majority of the assembly instructions are data moving
instructions, as shown above. We introduce in the following,
two MAIL assignment statements covering the data transfer,
arithmetic, logical and some of the system instructions. We
use EBNF [17] notation to define these statements:
Control instructions are very important because they can change the behavior of a program, and they can be changed or added by polymorphic and metamorphic malware to avoid detection. The following MAIL control statement represents the control instructions:

\[
\text{control} \ ::= \ ('\text{if} \ \text{condition} \ (\text{jump} \ | \ \text{assignment}) \ ('\text{else} \ (\text{jump} \ | \ \text{assignment}))?)? ;
\]

3.3 MAIL Library

The current MAIL library contains 22 functions. The following are some of the examples of MAIL library functions:

- \text{compare}(op1, op2): compares two values \(op1\) and \(op2\) and then set the flag register.
- \text{max}(op1, op2) and \text{min}(op1, op2): returns the maximum and minimum of the parameters \(op1\) and \(op2\) respectively.
- \text{swap}(op1, op2): swap the bits in \(op2\) and write back in \(op1\).

Details about all these library functions are given in [2]. These library functions can help in translating most of the complex assembly instructions present in current processor architectures. The purpose of these functions is not to capture the exact functionality of the assembly instruction(s) but to help in analysing the structure and behavior of the assembly program, and capturing some of the patterns in the program that can help detect malware.

3.4 MAIL Patterns for Annotation

MAIL can also be used to annotate a CFG of a program using different patterns available in the language. The purpose of these annotations is to assign patterns to MAIL statements that can be used later for pattern matching during malware detection. There are total 21 patterns in the MAIL language as follows:

\text{ASSIGN}: An assignment statement, e.g. \text{EAX}=\text{EAX}+\text{ECX};

\text{ASSIGN}\_\text{CONSTANT}: An assignment statement including a constant, e.g. \text{EAX}=\text{EAX}+0x10;

\text{CONTROL}: A control statement where the target of the jump is unknown, e.g. if (ZF == 1) JMP [EAX+ECX+0x10];

\text{CONTROL}\_\text{CONSTANT}: A control statement where the target of the jump is known. e.g. if (ZF == 1) JMP 0x400567;

\text{CALL}: A call statement where the target of the call is unknown, e.g. CALL EBX;

\text{CALL}\_\text{CONSTANT}: A call statement where the target of the call is known, e.g. CALL 0x603248;

\text{FLAG}: A statement including a flag, e.g. CF = 1;

\text{FLAG}\_\text{STACK}: A statement including flag register with stack, e.g. EFLAGS = [SP=SP-0x1];

\text{HALT}: A halt statement, e.g. HALT;

\text{JUMP}: A jump statement where the target of the jump is unknown, e.g. JMP [EAX+ECX+0x10];

\text{JUMP}\_\text{CONSTANT}: A jump statement where the target of the jump is known, e.g. JMP 0x680376

\text{JUMP}\_\text{STACK}: A return jump, e.g. JMP [SP=SP-0x8]

\text{LIBCALL}: A library call, e.g. compare(EAX, ECX);

\text{LIBCALL}\_\text{CONSTANT}: A library call including a constant, e.g. compare(EAX, 0x10);

\text{LOCK}: A lock statement, e.g. lock;

\text{STACK}: A stack statement, e.g. \text{EAX} = [SP=SP-0x1];

\text{STACK}\_\text{CONSTANT}: A stack statement including a constant, e.g. [SP=SP+0x1] = 0x432516;

\text{TEST}: A test statement, e.g. EAX and ECX;

\text{TEST}\_\text{CONSTANT}: A test statement including a constant, e.g. EAX and 0x10;

\text{UNKNOWN}: Any unknown assembly instruction that cannot be translated.

\text{NOTDEFINED}: The default pattern, e.g. all the new statements when created are assigned this default value.

4. MALWARE ANALYSIS USING MAIL

4.1 Approach Overview

Almost all the malware use binaries to infiltrate a computer system, which can be a desktop, a server, a laptop, a kiosk or a mobile device. Binary analysis is the process of automatically analysing the structure and behavior of a binary program. We use binary analysis for malware detection.

A binary program is first disassembled and translated to a MAIL program. In [2], we expalin in detail with examples
of translating a x86 and an ARM assembly program into a MAIL program. The MAIL program is then annotated with patterns. We then build a CFG of the annotated MAIL program. This annotated CFG becomes part of the signature of the program and is matched against a database of known malware samples to see if the program contains a malware or not. This approach is very useful in detecting known malware but may not be able to detect unknown malware.

It is difficult to write a new metamorphic malware [34] and in general malware writers reuse old malware. To hide detection the malware writers change the obfuscations (syntax) more than the behavior (semantic) of such a new metamorphic malware. If an unknown metamorphic malware uses all or some of the same class of behaviors as are used by the training dataset (set of old metamorphic malware) then it is possible to detect these type of malware using machine learning techniques. On this assumption and motivation, we train our detector (classifier) on the training dataset and detect unknown malware as follows:

After a program sample is translated to MAIL, an annotated CFG for each function in the program is built. Instead of using one large CFG as signature, we divide a program into smaller CFGs, with one CFG per function. A program signature is then represented by the set of corresponding (smaller) CFGs. A program that contains part of the control flow of a training malware sample, is classified as a malware, i.e. if a percentage (compared to some predefined threshold) of the number of CFGs involved in a malware signature match with the signature of a program then the program is classified as a malware.

4.2 Subgraph Matching

Before explaining the subgraph matching technique used in this paper for malware detection, we first define graph isomorphism [23] as follows:

Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be any two graphs, where $V_G, V_H$ and $E_G, E_H$ are the sets of vertices and edges of the graphs, respectively.

**DEFINITION 1:** A vertex bijection (one-to-one mapping) denoted as $f_V: V_G \rightarrow V_H$ and an edge bijection denoted as $f_E: E_G \rightarrow E_H$ are consistent if for every edge $e \in E_G$, $f_V$ maps the endpoints of $e$ to the endpoints of edge $f_E(e)$.

**DEFINITION 2:** $G$ and $H$ are isomorphic graphs if there exists a vertex bijection $f_V$ and an edge bijection $f_E$ that are consistent. This relationship is denoted as $G \cong H$.

An example of isomorphism is shown in Figure 1. The edges of graphs $G$ and $H_1$ are not consistent, e.g. edge {00, 10} in graph $G$ is not mapped to any edges in graph $H_1$, therefore graphs $G$ and $H_1$ are not isomorphic. Whereas the edges of graphs $G$ and $H_2$ are consistent, therefore graphs $G$ and $H_2$ are isomorphic.

In our malware detection approach, graph matching is defined in terms of subgraph isomorphism. Given the input of two graphs, subgraph isomorphism determines if one of the graphs contains a subgraph that is isomorphic to the other graph. Generally, subgraph isomorphism is an NP-Complete problem [15]. A CFG of a program is usually a sparse graph, therefore it is possible to compute the isomorphism of two CFGs in a reasonable amount of time.

Based on the definition of graph isomorphism presented above we formulate our CFG matching approach as follows:

Let $P = (V_P, E_P)$ denote a CFG of the program and $M = (V_M, E_M)$ denote a CFG of the malware, where $V_P, V_M$ and $E_P, E_M$ are the sets of vertices and edges of the graphs, respectively. Let $P_{sg} = (V_{sg}, E_{sg})$ where $V_{sg} \subseteq V_P$ and $E_{sg} \subseteq E_P$ (i.e. $P_{sg}$ is a subgraph of $P$). If $P_{sg} \cong M$ then $P$ and $M$ are considered as matching graphs.

After the binary analysis performed we obtain a set of CFGs (each corresponding to separate function) of a program. To detect if a program contains a malware we compare the CFGs of the program with the CFGs of known malware samples from our training database. If a percentage of the CFGs of the program, greater than a predefined threshold, match one or several of the CFGs of a malware sample (from the database) then the program will be classified as a malware.

4.3 Pattern Matching

Very small graphs when matched against a large graph can produce a false positive. Likewise to alleviate the impact of small graphs on detection accuracy, we integrate a Pattern Matching sub-component within the Subgraph Matching component. Every statement in MAIL is assigned a pattern as explained in Section 3.4. If a CFG of a malware sample matches with a CFG of a program (i.e. the two CFGs are isomorphic), then we further use the patterns, assigned to MAIL statements, to match each statement in the matching nodes of the two CFGs. A successful match requires all the statements in the matching nodes to have the same (exact) patterns, although there could be differences in the corresponding statement blocks.

An example of Pattern Matching of two isomorphic CFGs is shown in Figure 2. One of the CFGs of a malware sample, shown in Figure 2 (a), is isomorphic to a subgraph of one of the CFGs of a benign program, shown in Figure 2 (b). Considering these two CFGs as a match for malware detection will produce a wrong result, a false positive. The statements in the benign program do not match with the statements in the malware sample. To reduce this false pos-
itive we have two options: (1) we can match each statement exactly with each other or (2) assign patterns to these statements for matching. Option (1) will not be able to detect unknown malware samples and is time consuming, so we use option (2) in our approach, which in addition to reducing false positives has the potential of detecting unknown malware samples.

We conducted an experiment to evaluate the performance of our malware detection technique. The evaluation was carried out using 10-fold cross validation, and prototype implementation of our detector named MARD (for Metamorphic malware Analysis and Real-time Detection). MARD fully automates the malware analysis and detection process, without any manual intervention during a complete run. We present, in this section, the evaluation metrics, the experimental settings and obtained results.

5.1 Performance Metrics
To measure the performance of the malware detection technique we compute the detection rate (DR) and false positive rate (FPR). The DR metric indicates the number of samples correctly recognized as malware out of the total malware dataset. The FPR metric indicates the number of samples incorrectly recognized as malware out of the total benign dataset. These performance metrics are defined as follows:

\[
DR = \frac{\text{Number of correct malware detected}}{\text{Total number of malware in the dataset}}
\]

\[
FPR = \frac{\text{Number of incorrect malware detected}}{\text{Total number of benign programs in the dataset}}
\]

5.2 Dataset
The dataset used for the experiments consisted of total 1387 sample Windows programs collected from two different resources [30, 31]. Out of the 1387 programs, 250 are metamorphic malware samples, and the other 1137 are benign programs. The dataset distribution based on the size of the CFG after normalization is shown in Table 1. The dataset contains a variety of programs with CFGs, ranging from simple to complex for testing. As shown in Table 1, the size of the CFG of the malware samples range from 3 nodes to 129 nodes, and the size of the CFG of the benign programs range from 17 nodes to 15343 nodes. This variety in the samples provides a good testing platform for the graph and pattern matching techniques used in our tool.

<table>
<thead>
<tr>
<th>Size of CFG</th>
<th>Number of Samples</th>
<th>Size of CFG</th>
<th>Number of Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>200</td>
<td>17</td>
<td>127</td>
</tr>
<tr>
<td>88</td>
<td>1</td>
<td>30</td>
<td>44</td>
</tr>
<tr>
<td>91 - 99</td>
<td>38</td>
<td>44 - 998</td>
<td>412</td>
</tr>
<tr>
<td>100 - 104</td>
<td>10</td>
<td>1000 - 9765</td>
<td>535</td>
</tr>
<tr>
<td>129</td>
<td>1</td>
<td>10118 - 15343</td>
<td>19</td>
</tr>
</tbody>
</table>

5.3 Evaluation Methodology and Results
The experiment was run on the following machine: Intel Core 2 Quad (4 Cores) CPU Q6700 @ 2.67 GHz with 4GB RAM, operating system Windows 7 professional.
We conducted 10-fold cross validation by selecting 25 malware samples out of the 250 malware to train our detector. The remaining 225 malware samples along with the 1137 benign programs in our dataset were then used to test the detector. These two steps were repeated 10 times and each time different set of 25 malware samples were selected for training and the remaining samples for testing. The overall performance results were obtained by averaging the results obtained in the 10 different runs.

As explained above, to classify a program as benign or malware we compare to some predefined threshold (value) the percentage of its CFGs that match malware CFGs from the training set. To determine this threshold value empirically, we ran the above experiments with different values of the threshold ranging from 20% to 90%. Table 2 lists the obtained results. As it can be noted the best results are obtained for threshold values of 20% – 25%.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>DR</th>
<th>FPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>99.2%</td>
<td>3.07%</td>
</tr>
<tr>
<td>25</td>
<td>99.2%</td>
<td>3.07%</td>
</tr>
<tr>
<td>30</td>
<td>93.2%</td>
<td>3.07%</td>
</tr>
<tr>
<td>40</td>
<td>86.4%</td>
<td>3.07%</td>
</tr>
<tr>
<td>50</td>
<td>82.8%</td>
<td>3.07%</td>
</tr>
<tr>
<td>60</td>
<td>76%</td>
<td>3.07%</td>
</tr>
<tr>
<td>70</td>
<td>76%</td>
<td>3.07%</td>
</tr>
<tr>
<td>80</td>
<td>76%</td>
<td>3.07%</td>
</tr>
<tr>
<td>90</td>
<td>76%</td>
<td>3.07%</td>
</tr>
</tbody>
</table>

Using a threshold value of 25%, we conducted further evaluation by increasing the size of the training set from 25 samples to 100 and 200 malware samples, respectively. The obtained results are listed in Table 3. The DR improved from 93.92% when the size of the training set is 25 to 99.6% and 100% when we used a training dataset of 100 and 200 samples, respectively.

<table>
<thead>
<tr>
<th>Training set size</th>
<th>DR</th>
<th>FPR</th>
<th>Real-Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>93.92%</td>
<td>3.02%</td>
<td>✓</td>
</tr>
<tr>
<td>100</td>
<td>99.6%</td>
<td>3.43%</td>
<td>✓</td>
</tr>
<tr>
<td>200</td>
<td>100%</td>
<td>3.43%</td>
<td>✓</td>
</tr>
</tbody>
</table>

Real-time here means the detection is fully automatic and finishes in a reasonable amount of time.

6. CONCLUSION

7. REFERENCES


