REDIR: Automated Static Detection of Obfuscated Anti-Debugging Techniques

Adam J. Smith*, Robert F. Mills*, Adam R. Bryant†, Gilbert L. Peterson*, Michael R. Grimaila*
*Center for Cyberspace Research, Air Force Institute of Technology
Wright-Patterson Air Force Base, Ohio 45433
{adam.smith, robert.mills, gilbert.peterson, michael.grimaila}@afit.edu
†Cyber Research Laboratory, Riverside Research
Beavercreek, OH 45431
abryant@RiversideResearch.org

Abstract—Reverse Code Engineering (RCE) to detect anti-debugging techniques in software is a very difficult task. Code obfuscation is an anti-debugging technique makes detection even more challenging. The Rule Engine Detection by Intermediate Representation (REDIR) system for automated static detection of obfuscated anti-debugging techniques is a prototype designed to help the RCE analyst improve performance through this tedious task. Three tenets form the REDIR foundation. First, Intermediate Representation (IR) improves the analyzability of binary programs by reducing a large instruction set down to a handful of semantically equivalent statements. Next, an Expert System (ES) rule-engine searches the IR and initiates a sensemaking process for anti-debugging technique detection. Finally, an IR analysis process confirms the presence of an anti-debug technique. The REDIR system is implemented as a debugger plug-in. Within the debugger, REDIR interacts with a program in the disassembly view. Debugger users can instantly highlight anti-debugging techniques and determine if the presence of a debugger will cause a program to take a conditional jump or fall through to the next instruction.

Index Terms—Reverse code engineering, Expert systems, Sensemaking, Anti-debugging

I. INTRODUCTION

Reverse Code Engineering (RCE) is the process of analyzing binary programs without access to source code. Tasks such as malware analysis and software security auditing depend heavily on RCE [1]. However, RCE a time-consuming and complicated task that involves understanding computer hardware and software operations, low-level languages and logical analysis [2, 3]. RCE tools are available to help the human reverse engineer manage the complexity and facilitate the analysis process. However, anti-RCE practices can disrupt the use of RCE tools and techniques. These anti-RCE techniques complicate analysis efforts and are most prevalent in programs such as malware [4].

Anti-debugging is a form of anti-RCE that attempts to prevent a debugger from properly executing the program without intervention from the engineer. With enough skill and experience, the reverse engineer can continue the RCE task with well-placed anti-debugging mitigation techniques. However, regularly during RCE, obfuscations conceal the anti-debugging techniques and make the difficult task of RCE even more challenging. A great deal of experience is required to circumvent obfuscated anti-debugging techniques efficiently. RCE analysts would benefit from a tool that could quickly identify obfuscated anti-debugging techniques for efficient mitigation.

The Rule Engine Detection by Intermediate Representation (REDIR) system for automated static detection of obfuscated anti-debugging techniques is a prototype designed help the RCE analyst quickly, and correctly, avoid anti-debugging techniques.

REDIR relies on several principles to afford identification of obfuscated, anti-debugging techniques. Based on the Data/Frame sensemaking process, REDIR develops minimal starting information into a confirmed detection [5]. First, the Binary Analysis Platform (BAP) Framework translates the program into an IR that converts the Intel Architecture, 32-bit (IA-32) instruction set down to a much smaller set of semantically equivalent statements [6]. Next, the IR is parsed into an Expert System (ES) rule-engine to search the IR for instances of any anti-debugging technique characteristics. These minimal characteristics are IR statements that represent the beginning and ending of a technique based on minimal and unavoidable heuristics. The IR results from the rule-engine form the bounds for chopping the program down to a small sub-program of IR statements. An instrumentation process then adds code to simulate a debugging condition in the IR chop. Finally, concrete evaluation by taint analysis of the instrumented IR chop reveals the presence of the technique.

Unlike other solutions to detect these techniques, REDIR is static. Similar methods for detection use dynamic instruction traces to determine the presence of the desired code feature [7, 8]. Additionally, these methods report detections but do not offer them back for analysis. Conversely, REDIR will highlight the detected anti-debugging technique in the debugger disassembly view.

This paper proposes a method for static detection of obfuscated anti-debugging techniques based on a rule engine sensemaking process with the aid of IR. Section II briefly reviews similar research related to anti-debugging identification methods. Section III describes the goals and hypotheses presented in this document. Section IV provides background of key concepts including the testing methodology. Section V describes the system design and implementation. Lastly, Sec-
II. RELATED WORK

A. Instruction-Based Matching Algorithm

Instruction Trace Pattern Matching (ITPM) is an automated approach to detecting anti-debugging [7]. It is designed to search dynamic instruction traces for instances of anti-debugging patterns. First, to improve detection, a trace refiner scrubbed traces to remove unnecessary or obfuscating instructions. Once scrubbed, heuristic rules attempt to match the traces.

Xie, et al. conducted tests of the ITPM approach using 25 rules in four categories of anti-debugging: API calls, OS flags, magic strings and others [7]. Experiments processed 768 malware samples with a total detection rate of nearly 39%.

ITPM has two key characteristics that should be investigated: static vs. dynamic analysis and heuristic dependence. First, ITPM is a dynamic tool that relies on complete instruction traces. This in turn increases the probability of infection and requires the protection of a VM or emulator. As a result, ITPM must be conscious of execution safety. Static tools do not have this issue. Second, large instruction sets can subvert instruction-level analysis. Instruction matching rules must capture all possible combinations of instructions for a technique. If ITPM used IR, it could eliminate much of the confusion caused by large instruction sets. This would allow for more concise rules that covered many different implementations.

B. Divergence Detector

Anti-VM techniques share many of the characteristics and motivations as anti-debugging techniques. Divergence Detector is a system for detecting such anti-VM techniques in malicious programs [8]. This method followed the principle that at any time during execution, a program can only execute one anti-VM check.

Divergence Detector is a system built upon three common malware analysis VMs: QEMU, Xen and Bochs. Each VM is loaded with the same guest OS and sample program. When ready, the system executes the malware sample, outputs an instruction trace and rolls back to a pre-test state. Divergence Detector compared traces and noted execution differences as divergences. Wherever the execution paths differ, VM checks are present. To eliminate uncertain false-positives, the process repeats several times to remove non-deterministic divergences from analysis.

When tested, Divergence Detector was capable of detecting instances of anti-VM techniques in malware samples. Hsu, et al. describe several trials where a divergence occurs in one of the VM environment but not the others [8]. Analysis of uncertainty reduction in the system revealed as the number of experiment rounds increased, the number of false detections decreased. In the test program, false divergences followed the probabilistic model very closely and disappeared after seven rounds.

III. GOALS AND HYPOTHESIS

The goal of this research is to demonstrate that 1) demonstrate a rule-based ES can detect anti-debugging techniques; 2) prove that IR can facilitate detection by easing rule creation; 3) show that one rule can detect the same anti-debugging technique in several implementations with different common obfuscation techniques; and 4) develop a tool to aid reverse engineers analyzing programs with anti-debugging techniques. To achieve this goal, the Data/Frame sensemaking theory guides the process of developing minimal starting information into complete anti-debugging detections.

The following hypothesis drives this research. Most anti-debugging techniques begin at some calculated or retrieved value $\alpha$ and end at a control-flow decision $\beta$. In program $P$, a rule $R$ that searches for $\alpha$ and $\beta$ can lead to the creation of a sub-program $C = \{\alpha...\beta\}$ for instance $T(R, \alpha, \beta)$ of anti-debugging technique $R$. If $C$ is valid in $P$, then $C$ instrumented with additional data can replicate non-debugging ($C_{nd}$) and debugging conditions ($C_d$). Evaluation of $C_{nd}$ and $C_d$ creates boolean values $E_{nd}$ and $E_d$ respectively. If comparison of $E_{nd}$ and $E_d$ result in an inequality, then the data that replicated the debugging conditions caused the divergence. The divergence confirms the detection of $T(R, \alpha, \beta)$ in $P$.

IV. BACKGROUND

A. Static vs. Dynamic RCE

Static and dynamic analysis methods are the two main types of analysis in RCE [4]. Each has its own merits. Commonly, the first analysis methods attempted with a new program are static methods. These techniques include running analysis tools such as anti-virus or disassemblers. Because the program is never executed, static analysis methods are also usually safer. Conversely, dynamic analysis is often required to analyze encrypted or obfuscated programs. Some programs will decrypt themselves as they execute. Debugging these programs can allow the engineer to decrypt and dump these programs at the right time. Additionally, runtime data will make static analysis almost impossible. Effective, real-world analysis always requires a static and dynamic analysis approach.

B. Anti-debugging

Unlike disassembly which analyzes static executables, debuggers look at the code as it is executing. Unfortunately for reverse engineers, debuggers can be fooled with simple tricks [9]. Calls to system interrupts can force the debugger to lose context while analyzing. To detect debuggers, the program can generate checksums for portions of code as they exist in the execution stack. The breakpoints inserted by the debugger aid to the checksum calculation and the mismatch becomes easily detectable. Debuggers also often save a trace record to the stack. Analyzing the stack at certain points in execution can reveal part of this trace to the program.

Simple anti-debugging techniques include using specific Application Programming Interfaces (API), checking the debugger’s registry values, searching memory for specific de-
bugger strings (e.g. “Olydia”), or scanning for the particular drivers used by debuggers. The following techniques are debugger-agnostic examples of anti-debugging possible in user-level code.

a) Operating System Flags: The easiest method to detect if a debugger is in use is to look for specific flags set by the Operating System (OS) [4]. These flags are normally made available through invocation of the 32-Bit Windows API calls isDebuggerPresent() or isRemoteDebuggerPresent(). Additionally, these OS flags can be checked manually. The code in Listing 1 checks for a debugger by looking at the Process Execution Block (PEB) for the byte used by isDebuggerPresent().

```
mov eax, fs:[30h]
movzx eax, byte ptr [eax+2]
test eax, eax
jne DebuggerFound
```

Listing 1. IA-32 implementation example of manually testing the PEB isDebuggerPresent byte.

b) Timing: Another method for detecting the presence of a debugger is to use a timing comparison [4]. By checking the time twice, once before and again after code segment, the program can detect if its execution was delayed. The code in Listing 2 demonstrates a timing-based detection technique using the IA-32 rdtsc instruction which returns the number of processor cycles since startup.

```
rdtsc
xor ecx, ecx
add ecx, eax
rdtsc
sub eax, ecx
mov ecx, 0FFh
cmp eax, ecx
ja DebuggerFound
```

Listing 2. IA-32 implementation example of the RDTSC Timing detection technique.

c) Interrupt Handling: Interrupt handling techniques are very effective since they prey upon the debugger’s handling of the interrupt and the user’s incomplete understanding of the underlying operations [4, 10]. These techniques attempt to have the debugger change the data stored in flags and registers or act inappropriately. The “move stack segment” (MOV SS) technique is interesting because when a value is set to the Stack Segment (SS) register, the CPU will covertly set the Trap Flag (TF) in a special, multi-purpose data structure known as the EFLAGS register. Next, while debugging, the CPU will advance the Stack Pointer (ESP) and the debugger will clear the flag [3, 10]. While single stepping over the instruction, the debugger will seem to skip to the next instruction. This is because the TF will disable the debugger’s next single step interrupt. If that next instruction happens to persist the TF by pushing it onto the stack, the value is preserved and used later to direct control flow. Testing the TF will inform the program that the debugger cleared the TF. In Listing 3 the pop SS instruction covertly sets the TF. The pushfd instruction then pushes the EFLAGS image onto the stack. Now, the TF is available at any time for use in a control-flow decision.

```
push ss
pop ss
pushfd
test word ptr [esp+1], 1
jne DebuggerFound
```

Listing 3. IA-32 implementation example of the MOV SS detection technique.

C. Obfuscation

If malware authors all wrote malware the same way, the job of analyzing benign malware would be quite easy. To make analyzing programs more difficult, obfuscation techniques can disguise the true nature of a program.

1) Layout Obfuscation: Layout obfuscation techniques attempt to confuse the analyst by concealing important instructions among other irrelevant instructions [11]. Simple techniques include insertion of dead code (nop and other non-functional instructions) between the functional instructions [12]. Reassigning registers between code segments can further disrupt the analyst. The example shown in Listing 4 is the same program from Listing 1 with the instructions reordered in a process called code transposition.

```
jmp step1
step3:
  test eax, eax
  jmp step4
step2:
  movzx eax, byte ptr [eax+2]
  jmp step3
step4:
  jne DebuggerFound
  invoke StdOut, addr NoDebugger
  jmp done
step1:
  mov eax, fs:[30h]
  jmp step2
DebuggerFound:
  invoke StdOut, addr Debugger
  jmp done
done:
  exit
```

Listing 4. IA-32 implementation example of code transposition.

2) Conditional Code Obfuscation: Conditional code obfuscation techniques hide the intended execution paths of programs [13]. The strength of these techniques is that static analysis becomes very difficult as no one true execution path is detectable; dummy code presents a valid execution path.

One such method of conditional code obfuscation is an opaque predicate [14]. Here, the predicate (cause for some control flow decision) is unknown. The opaque predicate is expressible in terms of predicate P and program p. The predicate can evaluate always true P^T, always false P^F, or neither P^F if it does not always point the same direction.

For example, observe Listing 5. This is an example of a P^T opaque predicate. This program employs the algebraic identity (x + y)^2 = x^2 + 2xy + y^2 to form a number-theoretic opaque predicate which always evaluates true [15]. As a result, 26 lines of code have disguised a single unconditional jump. A human reverse engineer would require additional time to analyze this jump and a static analysis tool would likely be
D. Heuristic Analysis

Relating to static analysis of disassembled programs, heuristic analysis refers to pattern detection in programs [4]. Typically, scanning for select byte sequences will detect heuristic patterns. This method is very capable of efficiently detecting known instruction code fragments [16]. However, the efficacy of heuristic analysis degrades as obfuscations are applied. Unfortunately, all possible combinations of obfuscations and heuristic patterns are unknowable. New analysis techniques are required for detecting pattern detection in advanced malware.

```
xor eax, eax
add ax, x
add ax, y
imul ax, ax
push ax ; push (x*y)'2
xor eax, eax
mov ax, x
imul ax, ax
push ax ; push x'2
xor eax, eax
mov ax, y
imul ax, ax
push ax ; push y'2
xor eax, eax
xor ebx, ebx
mov ax, x
mov bx, y
imul ax, 2 ; ax = 2xy
pop bx ; bx = y'2
add ax, bx ; ax = 2xy + y'2
pop bx ; bx = x'2
add ax, bx ; ax = x'2 + 2xy + y'2
pop bx ; bx = (x*y)'2
cmp ax, bx ; always evaluates true: ax == bx
jne fake ; never jumps
```

Listing 5. IA-32 implementation example of a $P^p_T$ number-theoretic opaque predicate.

E. Rule-based ES

Rule-based systems have several advantages over other ES [17]. Rule-bases have a uniform syntax such as ruleid: If antecedent1 and antecedent2 ... then consequent. This syntax makes the rules self-documenting and easy to understand. Rules are also independent since each rule represents one fact about a particular domain. Some rule-based systems contain a working memory that can be updated with new data as it becomes available [18]. This allows rule engines flexibility when handling changing data sets.

There are several disadvantages as well. All rules exist on the same level; thus a hierarchy of rules is not possible [17]. As a result, each decision-making attempt requires testing every rule in the rule base. Rule-based systems also become tedious when representing human problem solving as a single task breaks down into numerous minute subtasks.

F. Testing Methodology

Unfortunately, it is very difficult to find anti-debugging technique samples that will disassemble correctly, are unencrypted, and guaranteed to exhibit the desired behavior. For these reasons, synthetic program samples modeled real-world malware anti-debugging implementations. These samples were implemented based on documented examples from real-world programs [3, 4, 10, 11, 13]. All samples used the IA-32 Microsoft Macro Assembler (MASM) assembly syntax and compiled with the MASM assembler. Each simple program attempts debugger detection and prints either “Debugger found” or “No debugger found” based on the detection result. For each anti-debug technique, one of each of the obfuscations disguised the technique. Due to the similarities among anti-debugging techniques, not all known anti-debugging techniques were required for testing. Many techniques employ the same overall strategy; as a result, their associated clues and instrumentations would only be slightly different and no less solvable. For testing REDIR, the techniques are broken down into representative categories as follows:

1) OS Flags - Represented by the Windows PEB!IsDebugger byte.
2) Timing - Represented by the RDTSC Timing technique.
3) Interrupt Handling - Represented by the MOV SS technique.

For each anti-debug technique, each of the following obfuscations will be applied to form a matrix of anti-debug technique/obfuscation samples.

1) No Obfuscation
2) Dead Code Insertion
3) Register Reassignment
4) Code Transposition
5) Instruction Substitution
6) Conditional Code Obfuscation

Obfuscation numbers two through five are forms of layout obfuscation. Obfuscation six is a form of conditional code obfuscation.

G. Intermediate Representation of Assembly Language Programs

Excluding floating-point and other special purpose instructions, the Intel 64 and IA-32 instruction set contain 254 unique general-purpose instructions [19]. Based on the complexity of the instruction set and the presence of side-effects, assembly-only analysis becomes very difficult [20]. Use of an IR will expose hidden operations and form an abstraction for a robust instruction set architecture [21]. Several IR implementations exist for disassembling and translating x86 programs.

H. Binary Analysis Platform

The BAP is a framework of tools designed to create and manipulate IR of executable programs [22]. BAP is an ongoing project at Carnegie Mellon University and has an active support community.
1) **Semantics:** The BAP Intermediate Language (BIL) is the IR form used by the BAP framework of utilities. BIL will decompose individual disassembled instructions into one or more statements. There are just seven different types of statement in BIL (var := exp, jmp, cmp, halt, assert, label and special) and all have zero side-effects. These statements reduce massive instruction sets down to simple intuitive operations.

Several BIL expressions go into an individual BIL statement. BIL statements use the following expressions to describe instructions. The load expression describes any activity where reading memory and storing the contents in another location. The store expression is the inverse of load as it describes when writing to memory. Additionally, expressions can take the form of binary and unary operations. The remaining expressions (lab, cast, let, unknown, and name) represent less frequent operations.

2) **Utilities:**

   a) toil: The primary purpose of the toil utility is to convert executable programs into BAP BIL. Programs analyzed with toil are first “lifted” to the BIL. Listing 6 demonstrates IA-32 instruction XOR EAX, EAX lifted to BIL. In this example, R_EAX:u32 represents the destination register. The remaining BIL statements expose the side effects of the XOR instruction. Additionally, the toil utility can lift dynamic traces into BIL.

```
addr 0x40100e @asm "xor %eax,%eax"
label pc_0x40100e
R_EAX:u32 = 0:u32
R_AF:bool = unknown "AF is undefined after xor":
  bool
R_ZF:bool = true
R_PF:bool = true
R_OF:bool = false
R_CF:bool = false
R_SF:bool = false
```

Listing 6. XOR EAX, EAX lifted to BAP BIL.

   b) iltrans: For user-prescribed transformations, the iltrans utility can modify BIL code into several different forms. This utility can create Abstract Syntax Tree (AST), Control-Flow Graph (CFG) and many other outputs. Numerous transformations are possible as a series of layers to refine the BIL for a given analysis. Chopping is a transformation which reduces a program down to only the BIL statements that affect a sink node (destination) for a given source node. Other transforms perform the removal of particular undesirable BIL such as dead (unreachable) or indirect (unsolvable) code.

   c) ileval: The ileval utility enables concrete evaluations to execute BIL code natively instead of requiring recompilation into higher-level languages. Variables added to the BIL program can determine how a program would execute. Flags, registers and memory can also be set at any point in the BIL code to simulate specific conditions. For example, if evaluating a suspected anti-debugging technique in a program, tainting the memory address checked by the windows isDebuggerPresent() function could affect the execution of the program (see Listing 7). ileval can execute the tainted BIL and determine the result.

### Listing 7. BIL code to taint the isDebuggerPresent() byte.

```
// initialize segment register base address
R_FS_BASE:u32 = 0x0:u32
mem?:u32 = mem?:u32 with [R_FS_BASE:u32 + 0x30: u32, e_little]:u32 = 0xdeadbeef:u32

// taint fs:[30h] + 2 = 1
mem?:u32 = mem?:u32 with [0xdeadbeef:u32 + 2:u32, e_little]:u8 = 0x1:u8
```

3) **Limitations:** BAP has several limitations. The main limitations outlined in the documentation are 1) only x86 and x86-64 processors supported; 2) does not support analysis of non-deterministic behaviors; 3) user-mode only; and 4) does not support floating-point instructions. Additionally, BAP does not support instruction sequences that form cycles. BAP will “unroll” a loop n times, but prior knowledge of n is required for correct analysis. Furthermore, while BAP can analyze Windows PE binaries, it does not function in Windows environments. BAP utilities can only be executed on Linux or Mac Os.

### I. RCE as a Sensemaking Task

Psychologists describe sensemaking as a composite process that incorporates creativity, curiosity, comprehension, mental modeling and situational awareness [23]. Klein, et al. describe sensemaking as a “motivated, continuous effort to understand connections...in order to anticipate their trajectories and act effectively”. To be an effective aid to sensemaking, a joint human-Artificial Intelligence (AI) team must be “mutually predictable”, “directable” and share a “common ground” (understanding) of the domain and problem [24].

Klein, et al. introduce the Data/Frame sensemaking theory as a process where understanding develops through a transition between frames [5]. These frames form into a closed loop encompassing the life of the sensemaking task (see Figure 1). Frames are created based on only minimal data. Through a process of questioning, elaborating, and reframing, the frame is refined for the life of the task.

![Fig. 1. Klein, et al. Data/Frame sensemaking theory [5].](image)

177
V. DESIGN AND IMPLEMENTATION

A. Design

The REDIR system is a debugger plug-in written in the Java language. It was designed to process 32-bit Windows executables and identify instances of anti-debugging. The integration with the debugger disassembly view provides users visual identification of the anti-debugging instance.

1) Sensemaking: Figure 2 depicts the REDIR concept through the Data/Frame sensemaking process. Frames are "constructed" with the detection of $\alpha$ and $\beta$. First, "questioning" creates the sub-program $C$ that provides for "elaboration" to create instrumented sub-programs $C_{nd}$ and $C_d$ are created. Then, "questioning" resumes by evaluating $C_{nd}$ and $C_d$ to create $E_{nd}$ and $E_d$. Finally, "questioning" determines an inequality confirms the detection of the anti-debugging technique.

![Fig. 2. REDIR concept through the Data/Frame sensemaking process.](image)

2) Rule Engine: The rule engine selected for REDIR was JBoss Drools [25]. Drools was selected because it is a Java-based rule engine which supports object-oriented rule processing with an intuitive syntax.

Rules written for REDIR are high-level patterns of IR. These patterns provide a minimal representation of the technique to build the detection on. Based on the Data/Frame sensemaking model, each rule activation by the rule-engine serves as the starting point for more in-depth analysis. First, Listing 8 depicts the rule used to detect instances of the PEB!sDebugger technique. This implementation is only concerned with accessing the FS segment register where the byte resides and any conditional jump.

![Algorithm 1 Generic Instrumentation Algorithm](image)

3) Intermediate Representation: The IR technology selected for this project was BAP 0.7 for the following reasons [22]. First, BAP is in active development and has an active user group for support. Next, BAP has an abstracted Intermediate Language (IL) that offers easily analyzable instruction semantics. Finally, BAP offers concrete execution of IL. This capability offered to not only detect instances of anti-debugging, but also determine the jump direction caused by the detection of the debugger.

To facilitate the use of BAP for evaluating programs, each frame from the rule engine contains the source and sink nodes that mark the beginning and end for a sub-program in BIL known as a chop. An instrumentation process adds additional variables and tainted values to the chopped program for evaluation. Each anti-debugging technique requires a different unique instrumentation. Listing 9 demonstrates an example of a minimum instrumentation and Algorithm 1 depicts a generic instrumentation process.

![Listing 9. Example of a minimum instrumentation (chopped code omitted).](image)
Algorithm 2 REDIR Algorithm

1: procedure GO(\text{Program } P)
2: \textit{bil} $\leftarrow$ \textit{BAP}.\textit{tail}(\textit{P})
3: \textit{ast} $\leftarrow$ \textit{BAP}.\textit{iltrans}(\textit{bil})
4: \textit{ruleEngine}.\textit{load}(\textit{ast})
5: \textit{F} $\leftarrow$ \textit{ruleEngine}.\textit{fireAllRules}()
6: \textbf{for} $f \in \textit{F}$ \textbf{do} \\
7: \textit{Chop} \textit{C} $\leftarrow$ \textit{BAP}.\textit{iltrans}(\textit{ast}, \textit{f}, \textit{a}, \textit{f}, \textit{b})
8: \textbf{if} valid($\textit{C}$) \textbf{then} \\
9: \textit{C}_{\text{id}} $\leftarrow$ \textit{instrument}($\textit{C}$, True, \textit{f}.\textit{technique})
10: \textit{C}_{\text{nd}} $\leftarrow$ \textit{instrument}($\textit{C}$, False, \textit{f}.\textit{technique})
11: $\textit{E}_{\text{id}}$ $\leftarrow$ \textit{BAP}.\textit{ileval}($\textit{C}_{\text{id}}$)
12: $\textit{E}_{\text{nd}}$ $\leftarrow$ \textit{BAP}.\textit{ileval}($\textit{C}_{\text{nd}}$)
13: \textbf{if} $\textit{E}_{\text{id}} <> \textit{E}_{\text{nd}}$ \textbf{then} \\
14: \textit{f}.\textit{detected} $\leftarrow$ true \\
15: \textbf{else} \\
16: \textit{f}.\textit{detected} $\leftarrow$ false \\
17: \textbf{end if} \\
18: \textbf{end if} \\
19: \textbf{end for} \\
20: \textbf{return} \textit{F}
21: \textbf{end procedure}

6) Design Considerations: Based on previously described limitations of the BAP framework (Section IV-H3), REDIR has two significant restrictions. Future releases of the BAP framework may mitigate these limitations.

a) Cycles: Anti-debugging techniques that form cycles such as loops are not analyzable in the BAP Framework. Subsequently, REDIR cannot detect these techniques.

b) Operating System Compatibility: BAP is not compatible with the Windows OS. BAP operates only in Linux and Mac environments. DigR is a Windows debugger. To facilitate using BAP with DigR a bridge was required. As REDIR was already Java-based, a simple Java-based proxy interacted with REDIR via Java Remote Method Invocation (RMI). This proxy received input from REDIR, executed the desired BAP program and returned the result to REDIR.

B. Implementation

REDIR and the BAP proxy executed within connected Virtual Machines (VMs). As depicted in Figure 3, DigR executed inside a Windows 7 VM and the BAP Framework and proxy inside an Ubuntu Server 12.04 VM. REDIR itself is a plug-in for the DigR debugger. As a plug-in, REDIR had access to show highlight anti-debug techniques in the DigR Disassembly View.

VI. EVALUATION

A. Test Results

The purpose of testing the REDIR plug-in was to determine if the tool was capable of detecting anti-debugging techniques in obfuscated code. Testing followed by initializing the REDIR plug-in for each test program and manually analyzing each result (see Section IV-F). Success was evaluated by $a$) correct identification of instruction lines used by the technique; and $b$) correct determination of jump direction.

The REDIR plug-in analyzed each of the 18 anti-debugging/obfuscation sample programs. In each case, the tool created multiple frames during the analysis. Many of the generated frames were invalid and correctly discarded.

Most programs correctly yielded only one valid frame. In all test cases, the REDIR correctly identified the technique and highlighted the set of instructions that affected the outcome of the program. REDIR was 100\% effective for those 18 test cases. REDIR did not highlight irrelevant instructions with no bearing on the outcome (see Figure 5).

B. Design and Implementation Analysis

REDIR excelled at many of the stated goals. The method for capturing detections by the Data/Frame sensemaking technique seems to be a valid starting point for future research. REDIR created and evaluated frames for correctness before employing more demanding analysis steps. This approach greatly reduced the problem search space and minimized expensive analysis steps by concrete evaluation.

REDIR offered additional benefits that were not originally intended. The original design for confirming the presence of an anti-debugging technique also offered consistent detection of a technique’s designed jump direction.
Most REDIR analysis tasks completed in less than one second. However, due to the less-than optimal multiple VM architecture, execution time results could not be viewed as meaningful metrics.

C. Test Analysis

This research selected a test methodology to demonstrate the feasibility of static analysis by sensemaking and IR analysis. In that task, the REDIR system was very successful. However, many other anti-debugging and obfuscation techniques exist. An exhaustive test of all known techniques was beyond the scope of this project. Definitive tests for real-world malware samples were impossible with static-only analysis, therefore REDIR did not test real-world samples.

VII. CONCLUSION AND FUTURE WORK

The purpose of this paper was to explore detection of anti-debugging techniques by static analysis in obfuscated programs. Following the stated goals and hypothesis, Section IV introduced related concepts and testing methodology. Next, Section V described the design and implementation. Subsequently, Section VI briefed test results and demonstrated the feasibility of the REDIR system. Finally, Section II offered two related projects were for comparison.

For future research, trace data can extend REDIR for dynamic analysis. The DigR debugger can supply the trace to REDIR for transformation to BIL code. Then, the rule engine can take in the resulting code and reprocess the rules. As the debugger progresses through execution, dynamic traced BIL code would replace static BIL code. As a result, a hybrid static/dynamic model of the program could exist in working memory and previously hidden anti-debug techniques would become visible. This enhancement would permit REDIR to analyze techniques containing loops, encryption and other limitation factors. Furthermore, this enhancement will extend the use of the Data/Frame model by introducing reframing and expanding the elaboration phase.

Further research should expand the test corpus to include more anti-debugging techniques, obfuscations and program complexity. Additional tests programs should incorporate these techniques into real-world programs. This expanded corpus would provide a foundation for continued REDIR testing as well as for other similar research.

VIII. ACKNOWLEDGEMENTS

I would like to thank Mr. Lok Yan (Air Force Research Lab, Rome NY, USA) and Mr. Edward Schwartz (Carnegie Mellon University, Pittsburgh PA, USA) for the assistance they provided using the BAP Framework.

REFERENCES


