
THE UNIVERSITY OF ARIZONA
University Libraries
UA CAMPUS
REPOSITORY

ENVIRONMENTAL MONITORING DETECTOR

Item type	text; Electronic Thesis
Authors	STEPHENS, JON BARTON
Publisher	The University of Arizona.
Rights	Copyright © is held by the author. Digital access to this material is made possible by the University Libraries, University of Arizona. Further transmission, reproduction or presentation (such as public display or performance) of protected items is prohibited except with permission of the author.
Downloaded	29-Sep-2017 22:18:58
Link to item	http://hdl.handle.net/10150/614237

ENVIRONMENTAL MONITORING DETECTOR

By

JON BARTON STEPHENS

A Thesis Submitted to The Honors College

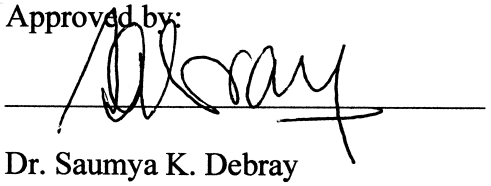
In Partial Fulfillment of the Bachelors degree
With Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

MAY 2016

Approved by:



Dr. Saumya K. Debray
Department of Computer Science

Environmental Monitoring Detector

Jon Stephens
Advisor: Dr. Saumya Debray

Abstract—Malware authors have developed many techniques that allow a malicious program to change its behavior, many of which require information from the computing environment. To fully understand how malware will affect a system, all behaviors it can exhibit need to be examined, so tools are needed that can expose when malware uses information from its environment to change its behavior. This project created such a tool called the environmental monitoring detector that will run a malicious program and search for cases of environmental monitoring while the malware is running. The tool is able to detect when a program uses environmental information to conditionally change its execution path; however, it has been found to be ineffective against obfuscated programs due to the lack of instruction specific taint propagation policies.

1 INTRODUCTION

As computers have become a larger part of everyday life, people have placed greater trust in them. This trust comes with the expectation that any personal information entered in a computer will remain confidential, unmodified and available for recall; however, malware (short for malicious software) threatens all of these expectations. Malware analysis is therefore an important area of research so that defenses against new attacks can be quickly developed.

Malware is a very broad term that is used to identify software designed with malicious intent that has a negative impact on some user. As one can imagine, the software that falls into this category is extremely diverse. All of the software has a malicious payload to execute, but the method in which they deliver the payload to a user, hide from detection, and attack a user can vary. Additionally, malware is constantly evolving to take advantage of new vulnerabilities, evade new defenses, and circumvent analysis attempts. The techniques used to analyze malware therefore have to be extremely general.

Many techniques have been employed to analyze malware, however this paper will focus on a technique called dynamic taint analysis, which tracks the flow of data through a program. This technique has been employed for many purposes including deobfuscating code [2], [3], malware analysis [4], [5], and software testing [6], [7]. It starts by marking the data from some source as tainted, such as data coming from the network. Taint markings are then propagated through the program's execution by marking written data and the instruction that writes the data as tainted if it makes use of a piece of tainted data in some way.

Taint analysis gives an analyst insight into how a program uses information from the taint sources, which the analyst can use to make inferences about the program's behavior. For example, a common technique employed by malware authors is to obfuscate their programs, which basically complicates the logic so that analysis is more difficult. Taint analysis has been successfully employed to deobfuscate programs, which makes further analysis easier [3]. Additionally, taint analysis can reveal if data from an untrusted source, such as the network, is used in an unsafe way, giving insight into how an attack is performed [4]. This project leverages taint analysis to detect if a program is monitoring its environment.

Environmental monitoring occurs when malware uses information extracted from the execution environment to change its behavior. This information could be from many sources, such as the date or time, and could be used for a variety of reasons. Identifying instances of environmental monitoring by finding environmental triggers, or decisions based on environmental values, is important because all execution paths must be explored to fully understand how malware will affect a system. Finding these environmental triggers manually, however, is extremely tedious, as one must either try to trigger these behaviors by modifying their environment, or read a runtime trace in an attempt to understand the malware's logic. These manual methods can easily miss triggers, so tools that can accurately and automatically detect environmental monitoring are desired to cut down on analysis time and improve the accuracy of the results.

One such tool called MineSweeper uses a mix of concrete and symbolic execution in an attempt to automatically detect and report environmental monitoring [10]. While it can detect environmental triggers in malware, it has some deficiencies that malware authors can take advantage of. The tool will assign one symbolic variable per byte of data drawn from possible sources of environmental information and will then propagate these values at the byte-level until a conditional branch is found. Propagating symbolic variables at the byte level can be taken advantage of to cause large amounts of data to falsely be marked as symbolic, which degrades the tool's performance and the accuracy of the results. Additionally, MineSweeper cannot handle obfuscated conditional jumps, allowing possible environmental triggers to escape its notice.

This project builds a tool that is capable of detecting environmental monitoring. Unlike other tools, it employs bit-level taint analysis and identifies conditional control transfers that have been influenced by the environment, including those that do not use conditional branches.

```
1  DATE date
2  date = getDate()
3  if(date.month == APRIL && date.day == 1)
4    //execute payload
5  else
6    //hide
```

Fig. 1: An example of environmental monitoring that will only execute its payload on April 1st.

```
1  long t1, t2, diff
2  t1 = getCycleCount()
3  t2 = getCycleCount()
4  diff = t2-t1
5  if(diff < 256)
6    //execute payload
7  else
8    //hide
```

Fig. 2: An example of using timing to detect if a program is being emulated, inspired by [15]. The payload will only execute if the two consecutive calls to getCycleCount() can be performed in less than 256 clock ticks, otherwise some non-malicious behavior will be performed.

2 BACKGROUND

2.1 Environmental Monitoring

Malware authors are acutely aware that once their malware has been discovered, it will undergo scrutiny. Thus, many malware authors include code intended to evade detection and hinder analysis. A common technique to do so is to monitor the environment by including environmental triggers that will execute the malware's payload only if a set of conditions is met [10]. This allows malware to hide in a system, taking no malicious actions, until some set of environmental conditions are met and only then will the intended payload execute.

One technique used by malware authors is to only run a payload on a certain date or at a certain time. For example, two mass-mailing worms from 2004 called NetSky and MyDoom executed their payloads based on time-sensitive triggers, as demonstrated in figure 1. NetSky would cause a computer to beep if the system time met some criteria that differed for different versions of the worm [11]. NetSky therefore announced its presence to the computer's user once the criteria was met, so by waiting for a particular date, it could hide and allow more computers to become infected before anyone knew about it. Similarly, MyDoom would run a denial of service attack on certain websites based on the date [12]. Like NetSky, waiting for a particular date before executing the attack gave the malware time to spread which would increase the effectiveness of its distributed denial-of-service (DDoS) attack. Additionally, the system time helped coordinate the actions of the infected machines so that they could be directed to attack multiple targets.

Another technique used by malware authors uses timing information to detect if the software is run under some sort of emulation. This relies on the fact that it will take a set of instructions longer to run if the malware is being emulated [13]. Thus, the malware can time how long it takes

```
1  int a, b, c
2  a = read()
3  b = a << 1
4  c = b & 0x1
5  if(c == 0)
6    //Always Executed
7  else
8    //Never Executed
```

Fig. 3: A simple example of a program that will not perform well using byte or word level taint analysis.

a set of instructions to execute and then run the payload only if the time was below some threshold as shown in figure 2. The cutoff threshold must be chosen wisely since many factors affect runtime, including the load factor of the CPU and type of CPU. Thus the chosen threshold must be permissive enough to work on multiple architectures with various loads, but strict enough to catch most emulators.

These are only a few of the possible sources and uses of environmental monitoring. Environmental data can be extracted from many sources making it difficult to pinpoint a small, constant set of inputs to watch; however, most comes from system and library calls. While it would be excessive to monitor every single system or library call, since many innocuous calls are made as a program runs, a subset of the executed calls could be selected and monitored.

2.2 Taint Analysis

To effectively implement taint analysis, one has to ensure that an appropriate amount of taint is being propagated [9]. Under-tainting occurs if data is not marked as tainted, but should be. By taking advantage of under-tainting, malware authors can perform actions that escape the notice of the analysis. Over-tainting, on the other hand, occurs when data is marked as tainted, but shouldn't be. By taking advantage of over-tainting, malware authors reduce the effectiveness of taint analysis since more data and instructions will be identified as tainted. An extreme case of over-tainting is referred to as a taint-explosion, wherein almost every instruction is marked as tainted. Such an event renders the results of taint analysis useless.

Precautions must be taken to ensure precise taint propagation, especially when dealing with malicious programs. Taint markings are maintained by keeping a shadow copy of the architecture, so that when data is determined to be tainted, it is marked as such in the shadow copy. A single taint mark, however, can be used to represent that multiple bits are actually tainted to cut down on the memory cost of taint analysis. This has led to three common implementations of the analysis technique: word-level, byte-level and bit-level. In these taint-analysis implementations a single mark is used to indicate if a word (4 bytes), byte or bit respectively has been tainted. Word-level taint analysis has the advantage of requiring the least amount of additional space for the analysis at the cost of precision, while bit-level taint analysis is the most precise, but requires the most additional space for the analysis. In the case of malware analysis, precision is important since malware authors will employ whatever tricks they can to hinder analyses on their

programs. As such, both word-level and byte-level taint analysis have been found to be too imprecise to analyze malware since malware authors can manipulate individual bits of data to induce under and over-tainting in the analysis [2]. For instance, consider figure 3. In the given code, the variable a would be initially marked as tainted since its value is being set by the function $read$, which is considered an input to the program. Then, the variable b is set to the value of a shifted left by 1. In such a case, the first bit of b should be untainted, since its value no longer depends on the result of the input, while the remaining bits are tainted. Bit-level taint analysis could correctly reflect that b is only partially tainted; however, both word- and byte-level taint analysis will cause over-tainting in this case since they cannot reflect the absence of the first bit's taint. As a result of the over-tainting, both word- and byte-level taint analysis will also taint c, despite the fact that it is merely selecting the first bit of b. As a result, the if-statement on line 5 would mistakenly be interpreted as a case of environmental monitoring.

Tracking taint on the bit level is only a part of what is needed to minimize under- and over-tainting. In the earlier description of taint analysis, it was mentioned that a destination needs to be marked as tainted if it uses any tainted data, but how should the destination be tainted? This question is answered by a taint propagation policy. A common taint propagation policy is to taint an entire destination if any tainted data is used. While such a policy is simple and easy to implement, it ignores the semantics of the instruction that is being executed which can be leveraged to cause under and over-tainting. In the example shown in figure 3, if such a simple taint propagation policy were to be implemented b would be marked entirely as tainted, just like it was with word- and byte-level taint analysis, since a was tainted. As a result, the analysis would report a false positive since it would determine that the if-statement was using tainted data. Thus, in addition to bit-level taint analysis, taint propagation policies that adhere to the semantics of an instruction being executed need to be adopted [2].

2.3 Intel Pin

Pin is a binary instrumentation framework provided by Intel that allows tools to insert code that will run in another binary's execution environment every time a particular event occurs, such as when an instruction executes or a system call is made [14]. Pin achieves this by having tools register callback functions for an event that contains the code the tool wishes to run when the event occurs.

This project is mainly concerned with Pin's instruction instrumentation since taint needs to be propagated whenever an instruction is executed. When a binary's instructions are instrumented, code is injected every time an instruction is run, so better runtimes are achieved by reducing the amount of code that runs per instruction. The authors of Pin were aware of this fact, and observed that many decisions only needed to be made once per instruction, and much of the information about an instruction only needed to be fetched once as well. They therefore separated the instruction instrumentation into two steps: the instrumentation step and the analysis step. The instrumentation step is meant to only be executed once per instruction, and will provide a tool with information about an instruction that does not depend on its runtime environment (instruction address, registers read, registers written, number of memory operands, etc). The analysis step will be run every time an instruction executes, and has access to the runtime information that was missing in the instrumentation step (register values, memory read/write effective addresses, etc) [14]. Tools using Pin's instruction instrumentation therefore need to be split into two steps as well. In their instrumentation step, they need to decide what analysis functions are appropriate to analyze an instruction and register these functions with pin using a callback mechanism. An instruction's analysis step performs whatever actions need to be executed every time a particular instruction occurs.

3 IMPLEMENTATION

The environmental monitoring detector runs on top of Pin so that it can identify environmental monitoring in a binary at runtime. There are 4 main parts to its instruction instrumentation algorithm, which makes up the bulk of the tool: information gathering, taint propagation policy selection, taint fetch and taint propagation. The first two of these steps occur in tool's instrumentation and are shown in algorithm 1. The information gathering step queries Pin to gather static information about an instruction, the most important of which is information about the source and destination operands of the instruction. The next step uses the information to select an analysis function that will appropriately propagate the taint. Taint fetching, shown in algorithm 2, occurs at the beginning of the tool's analysis. It finds the taint markings for the given source and destination operands and provides them to a taint propagation policy, the final step in an instruction's analysis. There are four total taint propagation policies. One, algorithm 3, must run every single time an instruction executes to check if the instruction bytes themselves are tainted. The remaining policies, algorithms 4, 5 and 6, are selected by the instrumentation step, and use the taint markings provided by the taint fetch to update the destination operands' taint.

3.1 Taint Sources

Taint should originate from any location that can provide a program with environmental data. A vast majority of environmental data is extracted using library and system calls so the environmental monitoring detector allows the user to specify which calls they would like to treat as taint sources since it is unlikely that all of them provide environmental data. All instances of the specified calls will be tainted, but different approaches must be taken for the different call types.

When a taint source is a system call, only the outputs will be marked as tainted. The purpose of a system call is to send information to and receive information from programs that run in kernel mode, which is a more privileged execution environment that isn't accessible to normal programs where the operating system and drivers run. Thus, the only information a program can gather from a system call

```
1       push ecx
2       call lib
3  lib: pop ecx
4       sub esp, 8
5       mul ecx
6       mov [esp], eax
7       mov [esp+4], edx
8       ret
```

(a) An example of a library call that will multiply eax and ecx, then store its result on the stack since it won't fit in a register

```
1         lea eax, [string]
2         push eax
3         call strlen ; eax=strlen
4         mov ecx, esp
5         call lib
6    lib: add eax, 1
7         sub esp, eax
8         push ebp
9         lea ebp, [esp+4]
10        push ecx
11        mov ecx, [ebp+eax]
12  loop: cmp eax, 0
13        jz done
14        sub eax, 1
15        mov dl, [ecx+eax]
16        mov [ebp+eax], dl
17        jmp loop
18  done: pop ecx
19        pop ebp
20        ret
21        mov eax, [length]
22        sub ecx, esp
23        cmp eax, ecx
24        jz payload
```

(b) An example of a piece of code that will use ESP to figure if the size of the string copied onto the stack has a specified length.

Fig. 4: Examples of when tainting ESP is valid and invalid.

comes from the call's outputs. Pin allows system calls to be instrumented, and has mechanisms to access the inputs and outputs of a system call, making this a relatively easy task. The difficulty, however, comes from the Microsoft Windows operating system.

Many system calls use the return value as an error flag, and some of the inputs to the system call actually are pointers to locations where outputs can be stored. Thus, the environmental monitoring detector must have some knowledge of each system call to know which of the system call's arguments are outputs, and what the size of each output is. Therefore, each system call must have a handler that knows all of this information. Currently the environmental monitoring detector has handlers for NtCreateFile, NtOpenFile and NtReadFile; however, more can easily be added as they are needed.

Pin identifies a system call by its ordinal, which is basically just a number used to identify what function the system call is referring to. The system call ordinals aren't static however, many of them change across major versions of the windows operating system, and some even change across minor versions of the operating system. As a result, the environmental monitoring detector has a table that maps a system call's ordinal to its handler. Thus, the tool can run on multiple versions of the windows operating system by updating the table to reflect the ordinal numbers on that version.

When a taint source is a library call, all data operated on by the library will be marked as tainted except if the ESP register is one of the destination operands. A library call is simply a call to a piece of code stored in a library on the computer that does some useful computation. Unlike system calls though, a library call will run in the same execution environment as the program that called it. As a result, any data originating from a library call has the potential of being used as a source of environmental monitoring and should be marked as tainted.

All data originating from a library call does not equate to all data written by a library call, so not all destination operands can blindly be marked as tainted. One common operation performed by a library is to store the values of registers it wants to use on the stack and then restore them at the end of the library call. These values obviously do not originate from the library call and so they should not be marked as tainted. To generate taint, the library should operate on a value in some way, not just move it from one location to another. As such instructions that move data should also move the taint markings to the new location so that tainted data will remain tainted and untainted data will remain untainted. Lines 9 through 17 of algorithm 1 demonstrates the associated logic. In order to taint all destination operands using algorithm 5, an instruction cannot be a mov, push or pop, all of which simply copy a value from one location to another. The mov, pop and push instructions that are in a taint source will fall through to the switch statement on line 15, since they are in a library call, where the appropriate taint propagation policy for the instruction type will be called.

The ESP register keeps track of the current position of the stack and is used by two very common operations: push

```
1  mov ecx, eax
2  xor eax, ecx
3  jz alwaysExecute
```

(a) An example of how xor can be used to disguise an unconditional jump as a conditional jump.

```
1  call read
2  mov ecx, eax
3  call read
4  xor eax, ecx
5  jz maybeExecute
```

(b) An example of how xor can be used to compare the values of two registers .

Fig. 5: Examples of how xor can be used to construct an unconditional or conditional jump.

and pop. The frequent use of ESP makes it extremely dangerous since tainting it can quickly lead to a taint explosion as all push and pop instructions would be tainted. ESP therefore needs to be handled with care. In x86, constant stack adjustments are common since the stack is often used to store local variables and hold large return values. Figure 4a, for example, does both of these. Two 32 bit numbers are multiplied together and since the result is too large to fit in a register, it is returned on the stack using the space that was allocated on line 4. One cannot extract environmental information from the stack pointer when constant adjustments such as these are made, so ESP should not be tainted. If, however, ESP is being adjusted by a value that contains environmental data, ESP should be tainted. Consider figure 4b. ESP is being adjusted by the length of some string on line 7 so that the string can be stored on the stack. Then on line 24, the program that called the library takes advantage of this to check the length of the string. It is therefore possible for ESP to yield environmental information, but only if environmental data was used to adjust it. This leads to the final condition on line 9 of algorithm 1. If an instruction in a taint source is making an adjustment to ESP, that instruction's taint propagation policy should be used so that ESP only becomes tainted if it is being adjusted by a tainted value.

3.2 Taint Markings

The environmental monitoring detector must be able to store taint markings for any location that can be influenced by environmental data. As far as the x86 architecture is concerned, data can be stored in two places: in the registers or in memory. As a result, there must be a mechanism to store taint markings for these locations. In addition, the purpose of this tool is to detect environmental monitoring, which occurs when a program will conditionally execute a piece of code. In x86, the conditional execution is typically performed using conditional jumps, which decide whether or not to jump to a particular location based on the status of a subset of the flags stored in a special register called the EFLAGS register. To properly detect environmental monitoring from conditional jumps, it is therefore necessary to track taint through the EFLAGS register as well.

Typical implementations of taint analysis use a single bit to mark if information is tainted; however, useful information such as what taint source the data was influenced by will be lost [2]. Tracking the taint sources that influence a piece of data is useful for a few reasons. First, if there are multiple taint sources, it is useful from the analyst's point of view to know which of those sources will effect the outcome of an instruction. Additionally, some instructions' taint propagation policies, such as xor, relies on this information. Consider figure 5 for example. In figure 5a, bits from the same source are being xor'd on line 2, meaning the result is always zero. This effectively turns the jump if zero on line 3 into an unconditional jump since the result of the xor will always be zero. For the taint analysis engine to determine this though, it must know that the two pieces of data being xor'd are the same. Determining if data is the same requires information about its origin, however, since the data must come from the same place. Knowing that two pieces of data have the same value is not sufficient as shown in figure 5b where the data being xor'd comes from two different taint generating calls. If the values being xor'd on line 4 are the same, the program will jump to the label $maybeExecute$, and if they are not, execution will continue on the current path. Thus, even if the values of the two operands are the same, the result of the xor must be tainted because the output is dependent on the results of the two read calls on lines 1 and 3. Therefore, information about a bit's source is necessary to properly reflect the semantics of instructions such as xor to ensure no over- or under-tainting occurs.

Multiple taint sources can influence data. For example, in figure 5b, the zero flag in the EFLAGS register on line 5 is affected by the result of the two reads on lines 1 and 3, so both of them are taint sources for the zero flag. Accordingly, the environmental monitoring detector must be able to track all of these taint sources. To do so, each taint source is provided an id which the environmental monitoring detector associates with the name of the library or system call. Each bit then stores a list of the taint source ids that impact that bit.

Currently, each instance of a taint source is given a single id to mark all the tainted data from that source. This scheme will not provide enough information to taint propagation policies such as xor since each bit is not uniquely identified; however, currently no instruction specific taint propagation policies have been implemented that can take advantage of bit specific ids. It was therefore decided to assign each instance of a taint source a unique id since the number of taint source instances will be vastly less than the number of number of bits of data that will be tainted by a taint source, meaning a smaller integer could be used to store the id. Thus, less memory is required by the environmental monitoring detector, there is still enough information so that the user can identify where environmental data originated from and a simple modification can be made to implement bit-specific taint source ids when they are required.

The x86 architecture can manipulate individual bits of data, but the smallest amount of data that it can operate on is a byte. It therefore makes sense to group the corresponding

```
struct TaintByte {
        UINT8 mask;
        list<unsigned short> srcs[8];
};
```

Fig. 6: The struct used by the environmental monitoring detector for a taint byte

```
struct ShadowArch {
        TaintByte regs[8][4];
        map<Addr, TaintByte> mem;
        TaintByte flags[4];
};
```

Fig. 7: The struct used by the environmental monitoring detector to store the taint markings

taint markings for a byte of data together since it reduces the amount of taint fetching that must be performed, and decreases the complexity of the taint propagation policies. As a result, each byte of data is associated with the taint markings of the bits in that byte, which will be referred to as a taint byte. Each bit of data requires a single bit to indicate if the data is tainted, so that the tool can quickly check its status, and a list of taint sources that affect that bit. The taint byte therefore has an 8 bit mask containing the taint status for each bit of data, and an array of size 8 that tracks the taint sources for the corresponding bit as shown in figure 6.

A shadow copy of the architecture will be maintained to store taint markings for the registers, memory and EFLAGS. In x86, both the registers and EFLAGS have a small, static size, so they can be defined statically. The 8 32-bit general purpose registers can be represented as 8 arrays that contain 4 taint bytes each, and the 32-bit EFLAGS register can be represented as 4 taint bytes. Memory, on the other hand, has a fixed size on a single machine, but it is very large, making it impractical to statically define taint markings for memory. To make it appear as if memory had been statically defined though, a hash map is used to map the address of a byte of data in memory to its corresponding taint byte. This allows quick access to each taint byte, and reduces the memory requirement of the environmental monitoring detector since the size of the hash map should be on the same order of magnitude as the amount of tainted data. Figure 7 shows the representation of the shadow architecture used by the environmental monitoring detector.

3.3 Taint Propagation

In the x86 architecture there are hundreds of instructions that all need their own semantic-specific taint propagation policies. This, however, is a momentous undertaking so currently the environmental monitoring detector has a single taint propagation policy and the infrastructure has been established so more can easily be added. The algorithm for the current taint propagation policy, referred to as the default taint propagation policy, is shown in algorithm 6. Since the policy does not know anything about the semantics of the instruction, it assumes that any tainted source bits could taint any of the destination's bits. Thus, if any of the sources are tainted, all of the destinations must be tainted as well. The policy ensures that no under-tainting will occur, which will guarantee that all instances of environmental monitoring will be found.

While all environmental triggers will be found, it is not guaranteed that they are the only ones that will be found. Ignoring the semantics of the executed instruction leads to an extreme amount of over-tainting. Consider figure 5a for example. If eax was initially tainted, after line 1 ecx would be tainted as well. At line 2 since both eax and ecx are tainted, the zero flag, which is set by xor, will be tainted. This means the conditional jump on line 3 will be marked as an instance of environmental monitoring, even though it is not. Thus, more taint propagation policies need to be added in the future since the default policy is too general.

To make the creation of new taint propagation policies easier, the environmental monitoring detector will relieve the policies of the burden of fetching the taint. Many x86 instructions allow data to to come from either memory or from the registers, so taint propagation policies must be capable of doing the same. Doing so can lead to a lot of duplicate code since each combination of operand source would have to be considered. Since many instructions would fetch taint in the same way, it made sense to separate the taint fetch from the taint propagation. The taint fetch retrieves the taint for each source and destination operand of the instruction and provides it to the taint propagation policy as shown in algorithm 2. Since the shadow architecture uses a taint byte regardless of whether taint originates from registers or memory, the taint propagation policy can remain ignorant to the storage medium the data came from. Thus, a new taint propagation policy only needs to update the taint bytes it is given, and the shadow architecture will be updated accordingly.

Once a new taint propagation policy for an instruction is created, it must be added to the selection process. Doing so only requires that a new case be added to the switch statement on line 15 of algorithm 1 that will register a call back to the appropriate taint propagation function. Thus, while there is currently only a single taint propagation function, the environmental monitoring detector has been built with the expectation that many taint propagation policies will be used and reduces the amount of work required to add new policies.

3.4 Detecting Environmental Monitoring

Environmental monitoring occurs when a program conditionally executes some program behavior. Typically, conditional execution is performed using conditional jumps, however this is not the only method that can be used. Currently, the environmental monitoring detector supports 3 different types of conditional execution.

Environmental conditional control flows that use conditional jumps or indirect jumps can be detected using similar methods. A conditional jump will use information stored in the EFLAGS register to determine if a particular condition is met, and if it is the execution will continue at the jump location. If a conditional jump is used to monitor its environment, then one of the EFLAGS used by the jump must be influenced by environmental data, and therefore must be

```
1       call read
2       cmp eax, [expected]
3       mov ecx, 0
4       sete cl
5       lea eax, [payload]
6       sub eax, [notPayload]
7       mul ecx
8       lea ecx, [notPayload]
9       add ecx, eax
10      jmp ecx
```

Fig. 8: An example of conditional execution implemented using indirect jumps

```
   jmp: jmp payload
        ...
1       call read
2       cmp eax, [expected]
3       mov ecx, 0
4       sete cl
5       mov eax, [jmp]
6       sub eax, [nop]
7       mul ecx
8       add [nop], eax
9  nop: nop
```

Fig. 9: An example of conditional execution implemented by rewriting the binary

tainted. Indirect jumps, on the other hand, will jump to a location given by either a register or memory location. This type of jump can be used to conditionally transfer control by modifying the jump location based on the output of some computation. Consider figure 8 for example. On line 2, the output of the read call will be compared to some expected value. Then on line 4, the result of this comparison will be used to set the ecx register to either 1 or 0 based on whether or not the two are equal. If the two are equal, then the offset between the two labels $payload$ and $notPayload$ will be multiplied by 1 and added to the location of $notPayload$ stored in ecx. This will effectively change ecx so that it contains $payload$'s location, causing the jump on line 10 to go to the $payload$ label. If the two are not equal, then the offset between $payload$ and $notPayload$ will be multiplied by 0 and added to the $notPayload$'s location in ecx. This means that ecx will still contain $notPayload$'s location and so the jump on line 10 will go to $notPayload$. A program can use environmental data to modify the destination of the indirect jump, therefore allowing the program to monitor its environment. If this were to occur, however, the address of the jump would be tainted, since it is influenced by environmental data. Thus, to detect environmental monitoring through the use of either conditional jumps or indirect jumps, all one must do is check if a jump is using any environmental data as shown in algorithm 4.

The binary itself can be rewritten at runtime to perform conditional execution. To do so a program will write to a location in memory that will be executed in the future. Figure 9 shows an example that will conditionally write a jump into the binary based on the outcome of a call to read. Similar to the example in figure 8, the output of read will be compared to some expected value, and if the values match either a 1 or a 0 will be written into ecx. Lines 5 reads the binary of the jump instruction into the eax register and then Line 6 calculates the offset required to change the nop instruction on line 9 into the given jump instruction. The offset is then multiplied by the outcome of the comparison stored in ecx and the result is added to the nop instruction. If the outcome of the read was equal to to the expected value, then when execution reaches line 9, the jump instruction will actually be executed since the offset added to the binary at the nop label will change the nop instruction into a jump. If the outcome of the read was not equal to the expected value, then when execution reaches line 9, a nop will be executed since the offset was 0. Binary can therefore be rewritten to perform conditional execution. If the binary were to be rewritten based on information derived from the environment, however, the bytes that make up the instruction would be marked as tainted in the environmental monitoring detector. Therefore, conditional execution as a result of rewriting the binary can be detected by checking if an instruction's bytes are marked as tainted, as shown by algorithm 3.

Once environmental monitoring is detected, it must be reported to the user. Currently a message is printed to the user identifying the instruction that monitors its environment, and the taint source(s) that generated the environmental data. This allows an analyst to examine how the environment can be modified to cause the program to travel down an alternate execution path. To aid in this process, the environmental monitoring detector currently prints out a full runtime trace of the program being analyzed as well. While this is excessive since the analyst does not need a full trace, they just need the tainted instructions that operate the environmental data before it is used to monitor the environment, having a full trace makes debugging easier.

4 EVALUATION

4.1 Performance

Any tool that analyzes a binary at runtime will have some sort of slowdown when the execution time is compared with that of the analyzed program alone. It is important to know how much slowdown a tool incurs though since it gives the user an idea of how long they need to wait for the results. Additionally, at the University of Arizona there are tools available that can analyze a runtime trace of a program, so any runtime system should be faster to execute than collecting a trace and running one of these tools on it. Runtime statistics were collected for 4 different programs. Two of the programs, factorial and fibonacci, are meant to simulate computation intensive software while the other two, MD5 and SHA1, simulate I/O intensive software. The table in figure 10 summarizes the runtime statistics of the environmental monitoring detector (EMD), and another tool called Instruction Trace, which uses Pin to generate a runtime trace of a program.

In its current state, the environmental monitoring detector comes with an extremely high performance cost. To put it

Program	Original		EMD with Trace		Instruction Trace		EMD without Trace	
	Runtime	Slowdown	Runtime	Slowdown	Runtime	Slowdown	Runtime	Slowdown
Fibonacci	0.845s	1x	2h 31m 52.989s	10782.323x	2h 13m 10.468s	9454.1774x	8m 21.492s	593.36446x
Factorial	0.670s	1x	1h 29m 33.158s	8020.1007x	1h 44m 18.024s	9340.8724x	5m 56.358s	531.90819x
MD5	0.690s	1x	1h 57m 8.711s	10184.627x	1h 46m 2.007s	9218.5707x	10m 18.541s	896.26858x
SHA1	0.708s	1x	5h 31m 34.591s	28100.931x	4h 23m 54.732s	22366.417x	18m 58.462s	1608.0673x
Average	1x		14271.995x		12595.009x		907.40213x	

Fig. 10: Performance statistics relating to the Environmental Monitoring Detector collected on a virtual machine with a 1 core CPU and 128MB of memory.

into perspective, an average slowdown of 14,272 means that it will take about 4 hours for the environmental monitoring detector to perform 1 second worth of computation from the original program. The environmental monitoring detector is also about 1.15 times slower than Instruction Trace. While this may seem disappointing, these results make sense. Currently the environmental monitoring detector will generate a full trace of the program as it performs its analysis. It will therefore have to perform more computations per instruction than Instruction Trace, which will naturally increase the slowdown of the tool.

Unlike Instruction Trace, the environmental monitoring detector does not need to generate a full trace, it is just useful for debugging purposes. Once complete with instruction-specific taint propagation policies, if only the tainted instructions that affected an instance of environmental monitoring were to be printed it is likely that the number of printed instructions would be much less than the number of instructions in a trace. Thus, runtime statistics were also taken with the tracing behavior removed. This shows the slowdown incurred by the taint analysis engine alone and will hopefully be close to the runtime of the final product since few instructions would be printed. Without printing a trace, the runtime of the environmental monitoring detector improved by a factor of 15.72, meaning that only about 15 minutes is required to perform 1 second worth of computation.

4.2 Precision

To be useful, the tool must be precise. The goal is to minimize the number of conditional branches that must be investigated by an analyst to find environmental monitoring. It is therefore not useful to use a tool that will identify all conditional branches as environmental triggers since it is already known that any branch could be an environmental trigger. Thus precision is important in a tool such as this. In the current state, however, the precision is lacking. It was mentioned earlier that there is currently only one taint propagation policy that is very general. Such a policy leads to frequent over-tainting, especially in obfuscated binaries.

The environmental monitoring detector has been tested on several binaries, both obfuscated and not. The tool performs well on simple programs such as fibonacci or factorial where there aren't very many complex operations. On programs such as these, the environmental monitoring detector can successfully identify all areas of the simulated environmental monitoring with no false positives. As the complexity of the program increases, the precision of the analysis decreases which is particularly evident on obfuscated binaries. Nearly every instruction becomes tainted, leading to every conditional branch being marked as an environmental trigger. The same can be seen in binaries with no obfuscations as well because of common tricks that the environmental monitoring tool's taint propagation policy cannot handle. For example, there is a common way to set a register to be zero by performing an exclusive-or (xor) on the register with itself. With a proper taint propagation policy for xor, this could easily be handled and the register would be marked as untainted. The environmental monitoring detector, however, will taint an entire destination if the source is tainted and so if such an operation were to be performed, the register would remain tainted.

Even though the current version of the environmental monitoring detector lacks precision, it should not be ruled a failure. First of all, no false negatives were found, which is encouraging. The taint propagation policy should result in over-tainting, but not under-tainting. Thus if there were a false negative, it would be due to a problem in the program logic. Secondly, whenever a false positive was found while testing the tool, the taint was able to be tracked back to an instruction that would not be handled properly by the current taint propagation policy. This indicates that there is not a bug in the taint analysis engine, but rather simply too few taint propagation policies, which can be added later.

5 FUTURE WORK

1) Instruction specific taint propagation policies need to be added to the program. As discussed previously in this paper, there currently is a large amount of over-tainting which leads to a large number of false positives. The over-tainting is caused by the lack of taint propagation policies that adhere to the semantics of the executed instruction, so more taint propagation policies need to be added.
2) Currently, the environmental monitoring detector cannot analyze multi-treaded programs. To analyze these types of programs, mechanisms need to be added to propagate thread-specific taint and there are times that data is stored inbetween callback functions that will need to be stored in thread-specific storage.
3) Taint is only propagated through direct data dependencies, however it should be propagated through control dependencies as well. A direct data dependency occurs when the outcome of an instruction is tainted because the instruction uses tainted data

from a previously executed instruction. A control dependency occurs when tainted data affects control flow, and then the change in execution path affects other data. By not considering control dependencies, under-tainting occurs.

4) Currently only 3 system call handlers have been added into the environmental monitoring detector to taint the outputs of a system call. More need be added to the environmental monitoring detector before it can be used to analyze malicious programs.
5) Outputting a full trace is only useful for debugging purposes. To improve runtime, the amount of I/O performed by the environmental monitoring detector should be reduced. To do so, only tainted instructions should be printed so that the analyst can see the execution path taken between a taint source, and an instance of environmental monitoring.

6 CONCLUSION

This project was ultimately able to create a tool that uses dynamic taint analysis to detect environmental monitoring within binaries. While it is not yet prepared to take on the task of analyzing obfuscated binaries, the environmental monitoring detector was able to show that it is capable of discovering environmental triggers hidden in software. The groundwork has been laid, and now the long, tedious task of creating x86 taint propagation policies must begin.

REFERENCES

[1] Schwartz, Edward J., Thanassis Avgerinos, and David Brumley. "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)." In *Security and privacy (SP), 2010 IEEE symposium on*, pp. 317-331. IEEE, 2010.

[2] Yadegari, Babak, and Saumya Debray. "Bit-level taint analysis." In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 255-264. IEEE, 2014.

[3] Yadegari, Babak, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. "A generic approach to automatic deobfuscation of executable code." In *Security and Privacy (SP), 2015 IEEE Symposium* on, pp. 674-691. IEEE, 2015.

[4] Newsome, James, and Dawn Song. "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software." (2005).

[5] Song, Dawn, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. "BitBlaze: A new approach to computer security via binary analysis." In *Information systems security*, pp. 1-25. Springer Berlin Heidelberg, 2008.

[6] Ganesh, Vijay, Tim Leek, and Martin Rinard. "Taint-based directed whitebox fuzzing." In *Proceedings of the 31st International Conference on Software Engineering*, pp. 474-484. IEEE Computer Society, 2009.

[7] Clause, James, Wanchun Li, and Alessandro Orso. "Dytan: a generic dynamic taint analysis framework." In *Proceedings of the 2007 international symposium on Software testing and analysis*, pp. 196-206. ACM, 2007.

[8] Bosman, Erik, Asia Slowinska, and Herbert Bos. "Minemu: The worlds fastest taint tracker." In *Recent Advances in Intrusion Detection*, pp. 1-20. Springer Berlin Heidelberg, 2011.

[9] Cavallaro, Lorenzo, Prateek Saxena, and R. Sekar. "Anti-taint-analysis: Practical evasion techniques against information flow based malware defense." *Secure Systems Lab at Stony Brook University, Tech. Rep* (2007).

[10] Brumley, David, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. "Automatically identifying trigger-based behavior in malware." In Botnet Detection, pp. 65-88. Springer US, 2008.

[11] Hindocha, Neal. "W32.Netsky.D@mm." *Symantec*. 2004. https://www.symantec.com/security_response/writeup.jsp?docid=2004-030110-0232-99&tabid=2.

[12] Gettis, Scott. "W32.Mydoom.B@mm." *Symantec*. 2004. https://www.symantec.com/security_response/writeup.jsp?docid=2004-012816-3647-99&tabid=2.

[13] Ferrie, Peter. "Attacks on more virtual machine emulators." *Symantec Technology Exchange* (2007): 55.

[14] Berkowits, Sion. "Pin - A Dynamic Binary Instrumentation Tool." *Intel Developer Zone*. 2012. https://software.intel.com/en-us/articles/pintool.

[15] Kang, Min Gyung, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. "Emulating emulation-resistant malware." In Proceedings of the 1st ACM workshop on Virtual machine security, pp. 11-22. ACM, 2009.

APPENDIX

Algorithm 1 The instruction instrumentation function

```
1: procedure INSINSTRUMENT(Instruction ins)
2:     regCallback(chkInsTaint, addr(ins), size(ins))
3:     regSrcs ← insRegSources(ins)
4:     memSrcs ← insMemSources(ins)
5:     flagSrcs ← insFlagSources(ins)
6:     regDsts ← insRegDestinations(ins)
7:     memDsts ← insMemDestinations(ins)
8:     flagDsts ← insFlagDestinations(ins)
9:     if inTaintSrc(ins) AND NOT (isMov(ins) OR isPush(ins) OR isPop(ins) OR includes(regDsts, ESP)) then
10:        regCallback(ins, getTaint, taintAll, regDsts, memDsts, flagDsts)
11:    else
12:        if !inLibrary(ins) AND (isCondBr(ins) OR isJmp(ins) OR isCall(ins)) then
13:            regCallback(ins, getTaint, chkEnvMonitor, regSrcs, memSrcs, flagSrcs, dsts, regDsts, memDsts,
    flagDsts)
14:        else
15:            switch opcode(ins) do
16:                case default:
17:                    regCallback(ins, getTaint, taintDefault, regSrcs, memSrcs, flagSrcs, dsts, regDsts, memDsts,
    flagDsts)
```

Algorithm 2 Fetches taint from the shadow architecture

```
1: procedure GETTAINT(FuncPtr propagate, RegVect srcRegs, UINT32 srcMemAddr, UINT32 srcMemSize, Flags
   srcFlags, RegVect dstRegs, UINT32 dstMemAddrs[], UINT32 dstMemSizes[], Flags dstFlags)
2:     srcRegTaint ← getRegTaint(srcRegs)
3:     srcMemTaint ← getMemTaint(srcMemAddr, srcMemSize)
4:     srcFlagTaint ← getFlagTaint(srcFlags)
5:     dstRegTaint ← getRegTaint(dstRegs)
6:     dstMemTaint ← getMemTaint(dstMemAddr, dstMemSize)
7:     dstFlagTaint ← getFlagTaint(dstFlags)
8:     srcTaint ← combineTaint(srcRegTaint, srcMemTaint, srcFlagTaint)
9:     dstTaint ← combineTaint(dstRegTaint, dstMemTaint, dstFlagTaint)
10:    propagate(srcTaint, dstTaint)
```

Algorithm 3 Check if instruction bytes are tainted

```
1: procedure CHKINSTAINT(UINT32 addr, UINT32 size)
2:     t ← getMemTaint(addr, size)
3:     if isTainted(t) then
4:         reportEnvMonitor()
```

Algorithm 4 Check if conditional branches are tainted

```
1: procedure CHKENVMONITOR(Taint srcs[], Taint dsts[])
2:     sourceId ← emptySourceId()
3:     tainted ← false
4:     for s in srcs do
5:         if isTainted(s) then
6:             tainted ← true
7:             sourceId ← addId(sourceId, s.sourceId)
8:     if tainted then
9:         reportEnvMonitor()
10:        for d in dsts do
11:            setTaint(d, sourceId)
```

Algorithm 5 Taints all destinations

```
1: procedure TAINTALL(Taint srcs[], Taint dsts[])
2:     sourceId ← getTaintSourceId()
3:     for d in dsts do
4:         setTaint(d, sourceId)
```

Algorithm 6 Applies the default taint propagation policy

```
1: procedure TAINTDEFAULT(Taint srcs[], Taint dsts[])
2:     sourceId ← emptySourceId()
3:     tainted ← false
4:     for s in srcs do
5:         if isTainted(s) then
6:             tainted ← true
7:             sourceId ← addId(sourceId, s.sourceId)
8:     if tainted then
9:         for d in dsts do
10:            setTaint(d, sourceId)
```