I, Joshua W Burbrink, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Science.

It is entitled:
Patterns in Dynamic Slices to Assist in Automated Debugging

Student's name: Joshua W Burbrink

This work and its defense approved by:

Committee chair: John Franco, Ph.D.
Committee member: Raj Bhatnagar, Ph.D.
Committee member: Paul Talaga,
Patterns in Dynamic Slices to Assist in
Automated Debugging

A thesis submitted to the
Graduate School
of the University of Cincinnati
in partial fulfillment of the requirements for the degree of

Master of Science
in the Department of Computer Science
of the College of Engineering and Applied Science

by

Joshua W. Burbrink

B.S. (Computer Science)

University of Cincinnati, Cincinnati, Ohio, USA

Thesis Advisor and Committee Chair: Dr. John Franco
Abstract

The process of manually debugging programs can be difficult and time consuming. The goal of this thesis is to aid developers in debugging their crashed programs by identifying common programming mistakes using information collected during the execution of the crashed program. Specifically, the solution proposed by this thesis calculates a dynamic slice with respect to conditions at the time of the crash and inspects the dynamic slice for characteristics that are only present when certain types of programming errors occur. Common programming mistakes explored by this thesis include: uninitialized heap variables, uninitialized stack variables, dangling pointers, and stack overflow. A prototype, based on the GNU Debugger, was developed to implement the concepts of this thesis and was used to identify common programming mistakes made in a set of example C programs. Experimental results showed that the solutions presented in this thesis performed very well. The prototype was able to identify all occurrences of uninitialized stack variables, uninitialized heap variables, and stack overflow, and only a few instances of dangling pointers were missed. Upon identifying a common programming mistake in the dynamic slice, the prototype was able to produce a high level debug report indicating source code lines relevant to the coding mistake.
## Contents

1. Introduction........................................................................................................................................... 1

2. Background ........................................................................................................................................... 3

3. Initialization ........................................................................................................................................ 13

4. Common Programming Mistakes ....................................................................................................... 16

5. Evaluation ............................................................................................................................................. 23

   5.1 Uninitialized Stack Variables ........................................................................................................... 23

   5.2 Uninitialized Heap Variables .......................................................................................................... 25

   5.3 Invalid PC Due to Corrupt Return Address .................................................................................. 27

   5.4 Dangling Pointers ............................................................................................................................ 30

   5.5 Limitations ...................................................................................................................................... 33

6. Related Work ....................................................................................................................................... 34

7. Conclusions and Future Work ............................................................................................................ 37
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>x86 Stack Layout</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Data Dependency Example</td>
<td>6</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Data Slice Example</td>
<td>7</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Dataflow Overview</td>
<td>11</td>
</tr>
<tr>
<td>Figure 5</td>
<td>SIGFPE Example</td>
<td>13</td>
</tr>
<tr>
<td>Figure 6</td>
<td>C Uninitialized Variable Example</td>
<td>16</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Uninitialized Stack Variable Pseudo Code</td>
<td>17</td>
</tr>
<tr>
<td>Figure 8</td>
<td>C Uninitialized Heap Variable Example</td>
<td>18</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Uninitialized Heap Variable Pseudo Code</td>
<td>19</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Corrupt Return Address Detection Pseudo Code</td>
<td>21</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Uninitialized Stack Variable Pseudo Code</td>
<td>22</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Uninitialized Stack Variable Dynamic Slice</td>
<td>23</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Uninitialized Stack Variables</td>
<td>24</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Uninitialized Stack Variable Example Source Code</td>
<td>25</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Uninitialized Heap Dynamic Slice</td>
<td>25</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Uninitialized Heap Variables</td>
<td>26</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Uninitialized Heap Example Source Code</td>
<td>27</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Corrupt Return Address Dynamic Slice</td>
<td>28</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Return Address Corruption Example</td>
<td>29</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Corrupt Return Address Example Source Code</td>
<td>30</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Dangling Pointer Dynamic Slice</td>
<td>31</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Dangling Pointer Example</td>
<td>32</td>
</tr>
<tr>
<td>Figure 23</td>
<td>Dangling Pointer Example Source Code</td>
<td>32</td>
</tr>
</tbody>
</table>
1 Introduction

To be successful at programming software, developers must learn the skill of debugging errors in their applications. Software bugs come in many shapes and sizes and require a wide variety of techniques to determine their root cause. Several tools, debuggers, and software monitors have been developed over the years to assist software developers in debugging their programs. These tools provide features like breakpoints, single-stepping, and symbol resolvers to aid the developer in visualizing the inner workings of software.

Further, techniques have been developed to automate the process of determining the root cause of software crashes for both the authors of the software as well as software security analysts. Some techniques rely on static analysis of the source code to produce a symbolic representation that can be used by theorem provers to prove certain properties about the source code. Other techniques augment binaries during compilation to add better variable and control flow tracking. Some techniques even record the entire execution of a program, providing the ability to know the values of memory and registers and the ability to reverse execution of the program one step at a time. Dynamic slicing is one such technique which walks an execution trace in reverse to determine all statements which contribute to a variable's value.

The goal of this thesis is to assist developers in debugging program crashes by augmenting dynamic slices to identify patterns of common programming mistakes which lead to the crash. The hypothesis is that there exist classes of common programming mistakes which yield similar behavior that can be observed in a dynamic slice taken with respect to the values that directly caused the program to crash.
To test this hypothesis, the first task is to obtain an execution log of a crashing program. The next step is to identify what variable(s) were directly responsible for the crash and then compute a dynamic slice of the target program with respect to that variable. The last step is to identify the existence of common programming mistakes by inspecting the generated dynamic slice for specific characteristics that are only present when such a mistake is made.

To perform these tasks, the GNU Debugger (GDB) was modified to identify the variable determined to be the direct cause of the crash and to compute the dynamic slice of that variable for Linux Executable and Linkable Formats files (ELFs) on the Intel x86 architecture. This thesis attempts to determine patterns in the dynamic slices of crashes due to common programming mistakes such as: uninitialized stack variables, uninitialized heap variables, stack overflow, and dangling pointers.

Finally, this thesis provides conclusions and lessons learned based on the preliminary results of this work. Future work considerations are also provided, including how this work can be expanded to other languages, support more classes of crashes, and support for conditional statements.
2 Background

This chapter is a basic overview of the specific environment in which this thesis will operate and describes the main challenges involved in computing dynamic slices. It provides the basic definitions which are used in later chapters.

Operating System and Architecture Details

This section describes the operating system and computer architecture details that are relevant to this thesis.

CPU Architecture

Intel's x86 architecture was chosen for the implementation of the prototype of this thesis. x86 is a little endian, Complex Instruction Set Computer (CISC) architecture [8]. In this architecture, a byte is 8 bits, a word is 2 bytes and a double word is 4 bytes. This architecture features memory and a set of registers which can be used for a variety of purposes.

Registers

The 32-bit x86 processors define a number of general and special purpose registers. The purpose of most of these registers is irrelevant to this thesis with the following exceptions:

- Extended Instruction Pointer (EIP) - Holds the memory location of the next instruction to be executed by the CPU.

- Extended Base Pointer (EBP) - Holds the address of the current stack frame.

- Extended Stack Pointer (ESP) - Holds the address of the top of the stack.
Memory Regions

When an ELF program is launched, different sections of virtual memory are assigned to the text and data segments. Additionally, memory segments are also reserved for the heap and stack. Each segment has its own set of permissions such as read-execute or read-write.

The Stack

The stack is a region of memory used for storing local variables and arguments for functions. As mentioned earlier, the top of the stack is pointed to by the ESP register. The stack grows from high to low virtual memory addresses. A function can allocate memory for its local variables by subtracting the number of desired bytes from ESP.

The compiler decides at compile time the size of allocations for each function and embeds the required instructions to modify the stack pointer. The stack is also used during function calls to store metadata necessary to restore state when the function completes. Such metadata includes the value of EIP, the current program counter, so that execution resumes in the calling function once the called function has executed. When a function is called, ESP is decremented by the number of bytes deemed necessary by the compiler.

Another piece of metadata stored per function call is the value of EBP. EBP points to the base of the frame for the function which is used to access its local variables and parameters. The original value of EBP is restored when the called function returns, and ESP is restored back to the current value of EBP.
Figure 1 visually depicts the stack and its intended use.

\[ \text{Figure 1: x86 Stack Layout} \]

Definitions

This section provides formal definitions used in the crash analysis of a program.

Definition: Instruction

An instruction is an atomic operation which can be executed by a CPU. Each instruction contains a set of source operands \( S \), which provide the input for the instruction. The set of source operands for an instruction \( I \) is denoted \( S(I) \). The instruction also contains a set of destination operands \( D \), which are written to as result of the instruction. The set of destination operands for an instruction \( I \) is denoted \( D(I) \). Operands may be hard coded numbers, registers, or memory locations.
**Definition: Program**

Every program may contain a potentially infinite number of paths. Each path is a potentially infinite sequence of pairs of an address and the associated assembly level instruction at that address.

**Definition: Execution Trace**

An execution trace is a path in a program where the value of each operand of an instruction is known, including registry values and memory locations.

**Definition: Instruction Data Dependency**

An instruction A is data dependent on another instruction B if instruction B contains a destination operand that is also a source operand for instruction A.

\[
\text{Data Dependent}(A, B) \text{ if } \exists x \text{ s.t. } x \in S(A) \cap D(B)
\]

For example, in Figure 2, \(S(3) = \{EAX, EBX\}\), and \(D(2) = \{EBX\}\). \(EBX\) is in both \(S(3)\) and \(D(2)\); therefore, instruction 3 is data dependent on instruction 2. Likewise, \(D(1) = \{EAX\}\), and \(EAX\) is in both \(S(3)\) and \(D(1)\). Therefore, instruction 3 is also data dependent on instruction 1.

1: XOR EAX, EAX
2: MOV EBX, DWORD PTR[ECX]
3: MOV DWORD PTR[EBX], EAX

**Figure 2:** Data Dependency Example

**Definition: Dynamic Slice**

A common practice in debugging is to determine how a variable's value was set that caused the crash. At a high level, the act of determining all instructions that lead to a variable's value during execution is a dynamic slice. The variable whose value is being traced is commonly called the slicing criteria.

Formally, a dynamic slice with respect to \(x\) is the subset of instructions from an execution trace from
which \( x \) is data dependent. For the purposes of this thesis, the slicing criteria may be one of the following:

- An x86 CPU register
- A memory address of a specified length

With respect to Figure 3, the data slice with respect to \( EBX \), are instructions 1 and 2. Instruction 2 directly modifies \( EBX \), and instruction 1 modifies \( ECX \), which is a source operand for instruction 2. Note that instruction 3 was not included in the slice as \( EAX \) has no impact on the value of \( EBX \) at instruction 4.

1: MOV ECX, DWORD PTR[EBP+4]
2: MOV EBX, DWORD PTR[ECX]
3: XOR EAX, EAX
4: MOV DWORD PTR[EBX], EAX

**Figure 3:** Data Slice Example

As has been previously mentioned, other additional information and instructions will be collected when computing the dynamic slice. For each instruction, the value of the stack pointer register \( ESP \) will be collected. Per each instruction, the return addresses of all function frames to the point of that instruction will also be recorded. Also, all CALL instructions will be added to the dynamic slice, as well as the first parameter to the function being called. GCC, by default, uses the C declaration (cdecl) function calling convention. Therefore, at the time of a CALL instruction, the first argument of any function is guaranteed to be at the memory location pointed by the stack register \( ESP \). The return values of functions are also collected. Because the cdecl calling convention is assumed, the value of \( EAX \) at a RET instruction is guaranteed to contain the return value of the function.
Definition: Tracked Dependency List

In the calculation of a dynamic slice, the list of data dependencies being tracked must be maintained after evaluating each instruction in reverse. The list of data dependent operands at a given instruction $I$ is denoted $DL(I)$. For example, when calculating the data slice with respect to $EBX$ in Figure 3, the list of dependent operands after evaluating instruction 3 is $DL(3) = \{EBX\}$; Instruction 3 did not modify $EBX$. After evaluating instruction 2, the list is $DL(2) = \{ECX, [0x1000]\}$; assuming the value of $ECX$ is 0x1000, the memory address 0x1000 becomes a data dependency. After evaluating instruction 1, the list is $DL(1) = \{EBP, [0x2004], [0x1000]\}$; assuming the value of $EBP$ is 0x2000.

Definition: Signals

A crash signal halts the execution of the currently running program. Signals may occur for a variety of reasons. The prototype of this thesis focuses on the following common signals in Linux C programs: $SIGSEGV$ and $SIGFPE$.

- $SIGSEGV$: The current instruction attempted to access memory which is unmapped or write to read-only memory.
- $SIGFPE$: The current instruction attempted a mathematical operation such as dividing by zero.
- $SIGILL$: An attempt was made to execute an illegal instruction.

Definition: Malloc

malloc is a common function for allocating memory for use by a program. For this thesis, malloc is formally defined as the following:

- $malloc(X) \rightarrow Z = [A, A+X]$ where $[A, A+X]$ is not partially contained by any previous output of $malloc$. The values at memory addresses $[A, A+X]$ are undefined.
Example:

- \textit{malloc(4)} \rightarrow Z = 0x10 = [0x10, 0x13].

\textit{malloc} will not return any memory allocation containing addresses within the range [0x10, 0x13] until those memory resources have been released.

\textbf{Definition: Free}

\textit{free} is another common function for releasing memory resources which have been previously allocated by \textit{malloc}. For this thesis, \textit{free} is formally defined as the following:

- Given: \(Y\), a previous output value of \textit{malloc} (\(X\)): \textit{free}(\(Y\)) \rightarrow Releases memory allocation [\(Y, Y+X\)]. \([Y, Y+X]\) then becomes available for use by future invocations of \textit{malloc}.

Example:

- \(X = \textit{malloc(4)} = 0x10 [0x10, 0x13]\)

- \textit{free}(\(X\)) \rightarrow Memory resources [0x10, 0x13] have been released and may be used for future calls of \textit{malloc}.

\textbf{C Standard Library}

The C standard library is a collection of functions, macros, and definitions for common programming tasks such as memory allocation, string manipulation, and processing input and output. The GNU C Library (GLIBC) is a specific implementation of the C standard library and is used in the prototype of this thesis. Of particular interest are the \textit{malloc} and \textit{free} functions.

\textbf{Definition: Debugging Information}

Debugging symbols are data structures that can be included with programs that provide high level information about the source code. With debugging symbols a debugger can determine which line in

\textit{...}
the source code a particular instruction is mapped to or what variable is assigned to a specific memory address. A benefit of choosing to modify GDB is that GDB’s functions for handling debug symbols will be reused for the purposes of the prototype of this thesis.

Strategy

With these definitions, the strategy for testing the hypothesis of this thesis can be better defined. Given an execution trace of a program that has crashed due to a supported signal, calculate a dynamic slice of that execution trace with respect to the operand that was determined to be directly responsible for the crash. Also, record the following information:

- Every CALL instruction along with its current value of ESP, and the first stack value, which corresponds to the first function parameter.
- Every RET instruction and the value of EAX.
- Current set of data dependent operands per instruction

This additional information will be vital in identifying patterns for common classes of programming mistakes. Upon successful determination of a common mistake, produce a high level bug report to the developer. Figure 4 shows a high level overview of how data flows from a crash to a debug report.
Assumptions

This thesis will make the following assumptions. First, the programs being debugged are C applications compiled by GCC. Second, it is only focusing on single-threaded applications. Next, some of the patterns for the class of common mistakes will require debug symbols of the target program. For example, to determine the existence of dangling pointers and uninitialized heap variables, the locations of `malloc` and `free` will need to be known. Other patterns require nothing other than the target binary, although the debug report will not be as detailed. Third, the cdecl calling convention is assumed to be used throughout the program.

Another assumption is that the C code being debugged does not include any hand written assembly. This is to ensure that the only instruction to ever intentionally set the return address for a function is the `CALL` instruction. This also means that only the compiler is generating instructions that modify the stack and frame pointer registers. The compiler is trusted to adjust the stack and frame
pointers correctly for each function; therefore, both frame and stack pointer registers can be trusted as being correct in the calculation of the dynamic slice. This assumption is vital in detecting crashes where the return address has been corrupted as well as detecting uninitialized stack variables.
3 Initialization

To calculate a dynamic slice, the slicing criteria must first be determined. Therefore, the first task of this thesis is to determine the slicing criteria based on the signal that was raised to stop the program. The signals supported by the prototype of this thesis are SIGFPE, SIGILL, and SIGSEGV. This is because these are the only signals that are raised exactly when an erroneous instruction is executed. Signals like SIGABRT, are manually raised by the GLIBC library after an error has been manually detected at a much later time. Other signals, such as SIGUSR1 and SIGCHLD may not indicate an error has occurred at all.

SIGFPE

This signal indicates that an arithmetic exception has occurred, such as a division by zero. In this case, the instruction that caused the SIGFPE be raised must be a division instruction such as DIV or IDIV. The source operands for this instruction are taken verbatim to be the slicing criteria.

\[ 1: \text{DIV EBX} \]

Figure 5: SIGFPE Example

In the assembly of Figure 5, DIV divides EAX by the source operand, EBX in this case, and stores the result back into EAX. Therefore, EBX must have been 0 and will be used as the slicing criteria.

SIGSEGV

The SIGSEGV signal indicates that an invalid memory address was accessed. This may mean that one of the source operands of the signal causing instruction was a memory operand with an unmapped memory address. This may also mean that the instruction was attempting to write to read-only memory. The following are examples of how an instruction could have an invalid memory address as an operand:
• The operand could be a hard coded value. In this instance, there is no dynamic slice necessary to be computed and this statement is included in the bug report.

• The operand is an implied operand for an instruction like push or pop. In this case, the stack pointer register, ESP, contains an invalid value and is used as the slicing criteria.

• The operand is specified in the instruction. For example, consider the instruction “MOV EAX, DWORD PTR[EBX]”. The operands inside the square brackets are intended to be used in calculating a valid memory address. In this case, EBX is intended to contain a valid memory address; yet this instruction caused a SIGSEGV. Therefore, EBX would be used as the slicing criteria.

Invalid EIP

Another form of crashing occurs when program execution stops without raising a signal. One common occurrence of this is when the program instruction pointer, EIP, contains an invalid value such as a memory address that is not mapped. Program execution will also stop if EIP points to an illegal x86 instruction, which can raise the SIGILL signal.

An invalid EIP can occur for a variety of reasons. The first is that the operand to a CALL or JMP class statement was invalid. If the operand is hard coded, then no dynamic slice will be calculated and this statement would be included in the bug report. If the operand is a register, then it will be the slicing criteria of the dynamic slice.

If the previous statement is a RET, then the return address in the stack was overwritten by an erroneous value, perhaps due to a buffer overflow. Alternatively, the return address of a function could have been overwritten to point at code other than where the calling function called the current function. In this scenario, when the function that the modified return address points to returns, a crash is very likely to occur. This is because the compiler makes specific assumptions about stack sizes for each
function. Typically, at the beginning of each function the stack is decremented by a certain amount, and at the end of the function the stack pointer is incremented by the same amount. This symmetry means that a calling function can assume that the stack pointer is the same before and after a call to a sub function. If the return address of a sub function is changed to some other code, the stack pointer could be altered and break the symmetry of the stack pointer adjustments made by the calling function.

In both scenarios, the return address that the \textit{RET} instruction would be restoring the program counter to is used as the slicing criteria.
4 Common Programming Mistakes

This chapter describes the types of common errors made by developers when developing applications. It also describes the aspects of the dynamic slices for which this the prototype of this thesis will be using to detect the common programming mistake.

Uninitialized Stack Variables

An uninitialized variable in this context is a stack variable whose value was not set before its use in other computations. For example, observe the following C code in Figure 6:

```c
int Add(int x, int y) {
    int a;
    int b = y;
    return a + b;
}
```

**Figure 6: C Uninitialized Variable Example**

Note that the variable `a` is used in the return statement but is never initialized. When this function returns, the instance of `a` will be lost. At the assembly level, when the function `Add` is called, the stack pointer is decremented and memory is reserved to account for the stack variables `a` and `b`. When the function returns, the stack pointer is restored to its previous value, and the memory locations where `a` and `b` are stored can now be used as stack space for another function.

Conceptually, if an uninitialized variable is part of a dynamic slice, it will be identified for the following reason. The stack address of the variable will be used as a source operand for an instruction in the dynamic slice. However, that stack address will never be set inside that function. Therefore, to catch uninitialized variables, the prototype of this thesis will monitor all memory locations in the stack found in the dynamic slice while performing the backwards calculation. If a stack address has not been set, and the `CALL` instruction which called that function is reached, then an uninitialized variable has been found. The bug report will then include the uninitialized variable and the function in which the
variable was declared. Note that to detect an uninitialized stack variable, no source or debugging symbols are required. The debugging symbols are only used to translate memory addresses to source code for the debug report.

Figure 7 includes the pseudo code used by the prototype to detect uninitialized stack variables.

```
Input: Dynamic Slice D
Output: (true/false), true if an uninitialized variable is found; false otherwise
Procedure: CheckUninitializedStackVars(D)
  for each instruction i in D do
    if opcode of i is CALL then
      for each dependency d in DL(i) do
        if d is a memory address inside the stack memory region and d < ESP at i then
          Generate a debug report to include d, i, and the function containing i
          return true
        endif
      endfor
    endif
  endfor
return false
endprocedure
```

**Figure 7: Uninitialized Stack Variable Pseudo Code**

**Uninitialized Heap Variables**

Uninitialized heap variables are very similar to uninitialized stack variables in that their values are never explicitly set during execution. The difference is that heap variables reside in memory addresses allocated by `malloc`. Therefore, the lifespan of a heap variable may extend past the current function in which the variable is being accessed. In fact, a heap variable is valid until it has been freed by `free`.

Figure 8 is an example of an uninitialized heap variable.
```c
int* Add(int x, int y) {
    int sum;
    int* a = (int*)malloc(sizeof(int));
    sum = x + *a;
    return sum;
}
```

**Figure 8: C Uninitialized Heap Variable Example**

The intent of this function is to allocate memory whose contents contain the sum of parameters \(x\) and \(y\). The stack variable `sum` does contain the calculated sum of \(x\) and the value located at memory address \(a\). However, \(^*a\) was never set, so the value of `sum` is undeterminable.

Failure to initialize heap variables which are expected to be used later can certainly lead to a crash. To detect uninitialized heap variables, the prototype of this thesis will look for when `malloc` is called and determine if that function allocated memory which was a data dependency for the crash. To do so two pieces of information must be learned about the call to `malloc`:

- The memory address returned by `malloc` – assuming it succeeded
- The size of memory requested by the calling function of `malloc`

To determine this information, the `CALL` and `RET` instructions for `malloc` are utilized. The memory address returned by `malloc` is guaranteed to be the value of `EAX` at the `RET` instruction for `malloc`. The size of memory requested by the user is guaranteed to be the top value in the stack at the time of the `CALL` instruction for `malloc`. This is due to the assumption of that the cdecl calling convention is being used.

Knowing the memory address and size allocated by `malloc`, it is then possible to determine if a given heap address data dependency was uninitialized. The heap address space is intended to be maintained by `malloc` and `free`. `malloc` is guaranteed to never hand out memory which was previously allocated. Therefore, the memory returned by `malloc` should only ever be referenced by the owner that
call to malloc after that call to malloc has succeeded. If there is a data dependency in the dependency list of the CALL to malloc in the range of memory allocated by a given call to malloc, then that data dependency is deemed to be uninitialized. Prior to that call to malloc, the memory returned by malloc is unusable as it might be returned to other users of malloc. Also, the exact memory address returned by malloc is nondeterministic, meaning that the user of malloc, in general, cannot know beforehand what memory address to expect to be returned.

Note that to detect an uninitialized heap variable, source code is not required; however, debugging symbols are necessary to identify where malloc is located.

Figure 9 shows the pseudo code used by the prototype of this thesis to detect uninitialized heap variables.

**Input:** Dynamic Slice $D$

**Output:** (true/false), true if an uninitialized variable is found; false otherwise

**Procedure:** `CheckUninitializedHeapVars(D)`

```
for each instruction $i$ in $D$ do
    if opcode of $i$ is RET and the function containing $i$ is malloc then
        let $ptr =$ the value of register EAX at time of $i$
        advance $i$ until the opcode of $i$ is CALL and the function being called is malloc
        let $length =$ the value of DWORD PTR[ESP] at time of $i$
        for each dependency $d$ in $DL(i)$ do
            if $d$ is a memory address and $d \geq ptr$ and $d \leq ptr + length$ then
                Generate a debug report to include $d$ and the source line containing this call to malloc
                return true
            endif
        endfor
    endif
endfor
return false
```

*Figure 9: Uninitialized Heap Variable Pseudo Code*

**Invalid EIP Due to Corrupt Return Address**

In this case, the return address in the stack was an invalid value. This most commonly happens when functions like strcpy or memcpy are used improperly and overwrite the return address of a function,
resulting in an immediate crash. It can also happen if the return address of a function is overwritten to point to other valid code other than the original code. In this case, the code that improperly received execution may execute many sub functions; however, when execution does return, a crash is likely to occur.

The slicing criterion for this error is determined via the technique described in Chapter 2. Based on the assumption that the only instruction to ever set a return address should be the \textit{CALL} instruction, the prototype of this thesis will inspect the dynamic slice, looking for memory data dependencies that match one of the return addresses of any of the function frames leading to that instruction. Should any instruction be found, and it is not a \textit{CALL} instruction, it would be flagged as the instruction that erroneously overwrote the return address and be included in the debug report. Also, the instruction responsible for the incorrect value of the program counter will be included in the debug report. Note that to detect the instruction that corrupted the stack, no debugging symbols or source code is required. The debugging symbols and source code are only used in the generation of the debug report.

Figure 10 shows the pseudo code used by the prototype of this thesis to detect the faulty instruction that overwrote the return address.
**Input:** Dynamic Slice $D$

**Output:** $(true/false)$, $true$ if the instruction that overwrote the return address is found; $false$ otherwise

**Procedure:** CheckCorruptReturnAddress($D$)

for each instruction $i$ in $D$ do
  if opcode of $i$ is CALL then
    continue;
  endif

for each dependency $d$ in $D(i)$ do
  if $d$ is a memory address then
    for each return address $r$ in the call stack leading to $i$ do
      if $d == r$ then
        Generate a debug report to include $i$ and the function containing $i$
        return $true$
      endif
    endfor
  endif
endfor
endfor

return $false$
endprocedure

**Figure 10:** Corrupt Return Address Detection Pseudo Code

### Dangling Pointers

A dangling pointer refers to a pointer whose memory address is being used after it has been freed. In GLIBC's version of free, the free function has been observed to modify values in the range of the memory allocated by `malloc`. This presents the easiest scenario in which dangling pointers may be detected. Memory accesses should not occur in memory that has been freed; therefore, the prototype of this thesis will inspect all memory data dependencies in the data slice that are in the heap. If one of those data dependencies were set by an instruction inside of `free`, then that data dependency will be flagged as being a dangling pointer. The fact that there exists a memory data dependency that is being modified by `free` implies that the program tried to dereference a memory address that was freed, which can certainly lead to a crash. When an instruction inside of `free` is detected modifying a data dependency, the argument to `free`, which is a memory pointer returned by `malloc`, will be included in a debug report as a dangling pointer. Note that to detect a dangling pointer, no source code is necessary; however debug symbols are required to identify the location of `free`.
Figure 11 shows the pseudo code the prototype of this thesis will use to detect dangling pointers.

**Input:** Dynamic Slice $D$

**Output:** (true/false), true if a dangling pointer is found; false otherwise

**Procedure:** CheckUninitializedStackVars($D$)

for each instruction $i$ in $D$ do

if opcode of $i$ is inside the function `free` then

for each dependency $d$ in $D(i)$ do

if $d$ is a memory address inside the heap memory region then

Advance $i$ until the `CALL` instruction for `free` is reached

let $ptr =$ the value of `DWORD PTR [ESP]` at time of $i$

Generate a debug report to include $d$ and $ptr$

return true

endif

endfor
endif
endfor

return false

endprocedure

**Figure 11:** Uninitialized Stack Variable Pseudo Code
5 Evaluation

The prototype code, to evaluate its effectiveness, was tested against five example crashing programs per each of the supported common programming mistakes: uninitialized stack variables, uninitialized heap variables, invalid EIP due to corrupt return address, and dangling pointers.

5.1 Uninitialized Stack Variables

The prototype was able to successfully identify every uninitialized stack variable that led to a crash for each of the five programs. Every report included the uninitialized variable name as well as the function which contained the variable.

1: ADD EAX,0x14
2: MOV EAX,DWORD PTR[EBP+0x8]
3: MOV DWORD PTR [EBP-0x54], 0x7C
4: CALL 0x8048F47
5: MOV DWORD PTR [ESP],EAX
6: SUB EAX,0x1
7: MOV EAX, DWORD PTR [EBP+0x8]
8: CALL 0x8048F47
9: DWORD PTR [ESP],EAX

Figure 12: Uninitialized Stack Variable Dynamic Slice

Figure 12 displays a subset of the dynamic slice that was calculated for one of the sample programs that crashed due to an uninitialized variable. Figure 13 visually depicts the criteria that are considered for detecting uninitialized stack variables.
Figure 13: Uninitialized Stack Variables

The white area of the graph in Figure 13 indicates memory that has already been reserved for stack use by previous calling functions. The darker area indicates stack space which has not yet been used. While computing the dynamic slice from instructions 1 to 3, the prototype computed up to three stack variables that were data dependencies. Instructions 4 and 8 were CALL instructions, meaning that the first statement of a called function had been reached for those respective functions. However, there was still one data dependency remaining in the stack space allocated for the function called by instruction 4. That stack variable was flagged as being uninitialized, which lead to that variable and its function being included in the debug report.
20. void test() {
    ...
27. unsigned int* k;
28. unsigned int j;
29. unsigned int c;
    ...
49. j = k + sizeof(unsigned int);
    ...
52. c = *j;

Figure 14: Uninitialized Stack Variable Example Source Code

Figure 14 shows the relevant source code pertaining to the crash in this example. Because the prototype was able to identify the uninitialized stack variable \( k \), a debug report was generated with the following items included: an error message indicating \( k \) was uninitialized inside function \textit{test}, the line containing the reference to the uninitialized variable (line 49), and the line of the crash (line 52).

5.2 Uninitialized Heap Variables

The prototype was able to successfully identify every uninitialized heap variable that led to a crash for each of the five example programs. Every report included the uninitialized variable name, the line where the variable was allocated by \textit{malloc}, and the line where the uninitialized variable was referenced.

1: MOV EAX,DWORD PTR[EAX]
2: MOV EAX,DWORD PTR[EBP-0x10]
3: MOV DWORD PTR [EBP-0x10],EAX
4: MOV EAX, DWORD PTR[EAX+0xC8]
5: MOV EAX,DWORD PTR [EBP-0x14]
6: MOV DWORD PTR[EBP-0x14],EAX
7: CALL 0x804E400 <malloc>
8: CALL 0x8048EE0

Figure 15: Uninitialized Heap Dynamic Slice
Figure 15 is a subset of the dynamic slice for one of the test programs that crashed due to an uninitialized heap variable. Figure 16 visually depicts the criteria used by the prototype to determine the existence of uninitialized heap variables.

![Uninitialized Heap Variables](image)

**Figure 16: Uninitialized Heap Variables**

The white space in the graph in Figure 16 indicates unused heap memory while the dark space indicates memory that has been allocated by `malloc`. While calculating a dynamic slice, as many as two data dependencies have been identified in this particular memory range. However, instruction 7 is a `CALL` that calls the `malloc` function which allocated the memory containing a data dependency. Note
that the instruction numbers increase by going in reverse in time. Therefore, instruction 8 occurred before the call to malloc, and the data dependency exists in unclaimed heap memory. Therefore, this indicates a breach in the usage of memory handled by malloc and the data dependency was flagged as being uninitialized.

Figure 17 shows the source code corresponding to the dynamic slice for this example. Because this prototype was able to identify the uninitialized heap variable f->y, line 16 was included in the debug report along with line 17 where the crash occurred.

```c
... 
3. typedef struct {
4.   char x[200];
5.   int* y;
6.   char z[50];
7. } foo;
... 
15. foo* f = (foo*)malloc(sizeof(foo));
16. int j = f->y;
17. int k = &j;
... 
```

**Figure 17:** Uninitialized Heap Example Source Code

### 5.3 Invalid PC Due to Corrupt Return Address

The prototype was able to successfully identify every occurrence where the return address of the current function was being overwritten by an instruction other than CALL. Every report included the location where the return address was overwritten as well as the line that was responsible for setting the value that eventually was an invalid value for EIP.
1: RET
2: LEAVE
3: SUB ESP,0x10
4: PUSH EBP
5: RET
6: POP EBP
7: MOV DWORD PTR [EAX], 0x8048414
8: LEA EAX,[EBP+0x4]
9: MOV EBP,ESP
10: PUSH EBP

**Figure 18:** Corrupt Return Address Dynamic Slice

Figure 18 shows a subset of the dynamic slice calculated for one of the example programs that crashed due to a corrupt return address. When this type of crash is detected, the goal is to identify what instruction modified the return address of its function that was not a *CALL* instruction.
Figure 19: Return Address Corruption Example

Figure 19 graphically depicts the criteria the prototype used to identify the instruction that corrupted the return address. To do this, the prototype searched for memory data dependencies that had the same address as the return address of the function. In this run, instructions 2, 3, and 6 have data dependencies that match the return address. Note that the data dependency in instructions 2 and 3 is never resolved in this portion of the slice. However, the data dependency in instruction six was resolved by instruction 7, which was, in fact, a MOV instruction. Based on the assumption that only CALL and RET functions should access the return address, this MOV instruction was identified to be buggy instruction that corrupted the stack.
Figure 20 shows the source code that corresponds to where the faulty instruction overwrote the return address which ultimately caused a crash. Because the prototype was able to identify this instruction as being erroneous, it included line 16 in the debug report.

```c
... 13. int* foo (int* bar) { 14.    *(x-1) = (int*)&func1; 15.    return x; 16. }
```

Figure 20: Corrupt Return Address Example Source Code

### 5.4 Dangling Pointers

The prototype was able to successfully identify three of the five cases where a dangling pointer caused a crash. Successful reports included the location where the return address was overwritten as well as the line that was responsible for setting the value that eventually was an invalid value for EIP.

In the unsuccessful cases, memory was allocated that had the same memory address as the dangling pointer. The dynamic slice showed that instead of free modifying the dangling pointer, the initialization code for the new pointer had modified the values that lead to the crash. Because free did not show up in the dynamic slice as modifying the dangling pointer memory values, the prototype did not detect the dangling pointer.

Figure 21 shows a subset of a dynamic slice calculated for one of the successful detections of a dangling pointer.
1: MOV EAX, DWORD PTR[EAX]
2: MOV EAX, DWORD PTR[EAX+0XC8]
3: MOV EAX, DWORD PTR[EBP-0X14]
<free> 4: RET
<free> 5: MOV DWORD PTR [EBX+ESI*1], ESI
<free> 6: AND ESI, 0xFFFFFFFF8
<free> 7: MOV ESI, EAX
<free> 8: MOV EAX, DWORD PTR[EDX+0X4]

**Figure 21**: Dangling Pointer Dynamic Slice

Figure 22 graphically depicts how the dangling pointer is detected. Note that instructions 4 through 8 are all inside `free`. In the graph, instructions 2 through 4 are tracking a memory heap dependency that is resolved by instruction 5. Since instruction 5 is inside free, the prototype was able to detect the dangling pointer.
Figure 22: Dangling Pointer Example

This example corresponds to the following code in Figure 23. Since this prototype was able to detect the dangling pointer, it generated a debug report which included the dangling pointer `foo` being freed on line 19 and inappropriately being access on line 20.

```c
...  
18. *(foo->bar) = 0x1234  
19. free(foo)  
20. d = *(foo->bar);  
...
```

Figure 23: Dangling Pointer Example Source Code
5.5 Limitations

As observed in the results of the dangling pointer evaluation, the current algorithm is not capable of detecting dangling pointers when their memory has been reallocated to new pointers.

Other limitations stem from the assumptions made by this thesis. This prototype does not support multi-threaded applications. The prototype currently only supports the Intel x86 architecture; although the principals could be applied to other architectures like Intel’s x86-64 architecture. It currently assumes the cdecl calling convention; although similar characteristics of common programming mistakes could be found in other calling conventions. It is not guaranteed to function correctly if the developer has manually added assembly code which would invalidate any of the x86 specific assumptions made by this thesis.

During development and experiment phases, it became clear that the impact of having GDB record program execution of an application would have a serious impact on the performance and timing of the debugged program. For example, one of the test programs normally took 0.182 seconds to execute. With GDB recording turned on, its execution time rose to 3.860 seconds when the program was dynamically linked. With static linking, the execution time of GDB recording that program did drop to 0.035 seconds. An attempt was made to have GDB record larger graphical applications, but the applications either did not load at all or were unresponsive.
6 Related Work

Dynamic slicing was first introduced in [10], and has since served in the debugging of programs. Its primary advantage over conventional static program slicing was that it explicitly stated which instructions actually impacted the slicing criteria’s value, whereas static program slicing included all instructions which might alter the slicing criteria’s value.

The work in [15] combines both dynamic and static slicing of target criteria to generate a concise description of the slice code. Using this description, it becomes easier for the programmer to understand exactly what code had an impact on a given variable. This thesis is different in its goal of attempting to produce a high level diagnosis of an already crashed program.

There are many derivations of dynamic slicing that differ in what data dependencies are included in the slice. Thin slicing [13] includes a subset of data dependencies. Data slicing [19] includes all data dependencies in the slice. Full slicing [16] incorporates both data and control dependencies. Relevant slicing [4] includes all predicates, and potential dependencies rooted at these predicates, whose execution could have affected it with a different evaluation. There have also been many surveys, such as [14] and [12], done on the various types of program slicing available.

Other enhancements to dynamic slicing [18] attempt decrease the nodes in the dependency slice. Dynamic slices can get very large, and the ability to keep the dynamic slice small improves the chances of finding the faulty code. There is the risk of losing the faulty instruction when pruning the dynamic slice. To mitigate that risk, the authors assign a confidence value to the nodes to establish the likelihood of the instruction being relevant to the bug.

In [20], the authors set a goal for being able to compute a dynamic slice of a long running program. To do so, it augments the target program to create an execution trace which is used to form
the data dependencies of the slice. The slice is computed over a target portion of the program chosen by
the developer. It then then uses checkpoints to log execution of a long running program and replays the
log with the dynamic slice instrumentation in play. This work may be beneficial to this thesis as it
proposes a solution for dealing with long running programs.

The work in [17] computes slices based on the symbolic value of data dependencies rather than
their actual value. This thesis augments dynamic slicing with the ability to identify patterns in order to
provide higher level details of the bug to the developer.

The authors of [1], [2], and [5] produced and enhanced a debugging tool called SPYDER which
was capable of calculating dynamic slices for C source level code. The prototype of this thesis
calculates dynamic slices based on x86 assembly.

Relevant work has also been done in using dynamic data dependencies to find the relevant parts
of an input that are responsible for a given failed output. Works such as [15] combine symbolic
execution and dependency analysis for test-suite augmentation. The work in [7] utilizes symbolic
execution of programs, static data and control dependencies to generate criteria for additional test cases
for to improve the effectiveness of the test-suite. The purpose of this thesis is different. Instead of
trying to find test cases that stress a given program change, this thesis attempts to provide higher level
information about a dynamic slice that will aid the developer in debugging the problem.

The work in [6] computes dynamic slices of a known good implementation of software with a
newer buggy version of that software. It then calculates weakest preconditions of those slices and their
differences in an attempt to isolate what the issue is with the new software. This thesis does not assume
that a golden implementation of the target debugged application exists.
Dynamic slicing has also been used in finding vulnerabilities and exploits in target software applications. The work in [9] compares multiple traces of the same program to determine the exact differences between successful and crashing executions of a function.

The work in [11] computes a dynamic slice of a crashing program and watches for potentially vulnerable functions like `strcpy` or `memcpy` for use in a potential attack signature. The goal of this thesis is different in that a high level reason for the crash is provided to the developer as opposed to searching for ways to exploit the crash.
7 Conclusions and Future Work

Automatically determining the cause of a crash is a difficult task. There are many factors to consider due to the many variables, code paths, and types of crashes that can occur. The goal of this thesis was to assist developers in debugging their crashes by identifying common programming mistakes in their code.

This thesis provided a successful solution to identifying uninitialized stack pointers, uninitialized heap pointers, and faulty code which overwrites the stack’s return address. The prototype produced by this work was able to successfully identify each of these mistakes successfully in every test conducted. This thesis solution for identifying dangling pointers was not quite as successful. The prototype missed a couple cases in which the dangling pointer’s memory was reallocated; however, no false-positives were given by the prototype and this solution could still be used to identify a subset of programming mistakes in the class of dangling pointers.

Leveraging an existing code base as feature complete as GDB was beneficial; it provided a lot of pre-existing functionality required to implement this thesis. However, recording program execution to produce a trace proved to have a significant impact on programs execution time. In some cases, the impact was as much as 2120\% and would be severe enough to prevent reproducing the crash if the crash was time sensitive. Recording program execution with GDB, as expected, proved to have a large impact on memory usage. For longer running programs, the system may run out of memory before the program actually crashes. The overhead of calculating dynamic slices has been a known problem, and researchers such as [20] recognized that most of the recording of larger applications is generally not relevant to the problem of interest. An example might be event handling that is waiting on user input. That concept could be applied to GDB to further enhance its recording capability to only record threads or functions of interest.
This thesis covered some of the more basic classes of crashes caused by common programming mistakes. There exist other classes of mistakes, such as overflowing heap variables causing heap corruption, infinite loops, and type misuse which could also be analyzed for leaving patterns in a dynamic slice. To support these types, infinite loops in particular, control flow dependencies might also need to be calculated as well as data dependencies.

This thesis assumes a single threaded environment. However, common mistakes of multi-threaded programming like race conditions or dead lock may also leave patterns in a dynamic slice. These mistakes are often hard to debug and having a tool that could pinpoint the order of events necessary to produce a race condition could save a lot of time spent in debugging. This thesis could be expanded to support such issues.

This thesis currently only supports the Intel x86 architecture. However, its principles could be applied to other assembly languages like x64, MIPS or ARM.
Bibliography


