

# HeapBud: A Software Tool for Heap-Based Bug Detection

Ahmad Almheidat and Mazen Kharbutli

Department of Computer Engineering  
Jordan University of Science and Technology  
{anmhiedat04, kharbutli}@just.edu.jo

**Abstract:** With software applications continuously growing in size and complexity, they become increasingly prone to memory bugs. Memory bugs are reported daily in commercial and open source applications alike including common applications such as Microsoft Internet Explorer and Macromedia Flash. Such memory bugs can lead to crashes, runtime slowdowns, incorrect output, and most seriously security threats. To help developers detect memory bugs, several static and dynamic bug detection tools have been introduced. In this paper, we introduce Heap Bug Detector (HeapBud), a software-based static tool that analyzes programs written in C and pinpoints possible memory bugs. The tool relies on analyzing all possible paths in the program's control flow graph in addition to all pointer usages in search of possible scenarios leading to memory bugs such as dangling pointers, double and invalid frees, and memory leaks. HeapBud does not require hardware modification, does not require the availability of the original source code, does not incur runtime overhead, and is input independent, making it an attractive solution.

**Keywords:** Heap Bug Detection; Heap Memory; Program Bugs; Memory Leaks; Double Free; Dangling Pointers.

## I. INTRODUCTION

As software applications become more complex in terms of code size and structure, they become increasingly prone to memory bugs which in many cases manifest in the heap memory. Moreover, they become more and more difficult to thoroughly debug and verify. Memory bugs are more likely to appear in codes written collaboratively due to poor communication and documentation between collaborators or due to changing assumptions and requirements over the project development cycle.

Memory bugs often manifest in one of several forms: a crash that always happens, a crash that sometimes happens (instability), poor performance (runtime

slowdowns), or incorrect output (Gottbrath, 2008). More alarmingly, recent studies have pointed out that memory bugs can be easily exploited to launch security attacks and lead to service interruptions, information leakage, and unauthorized access (Lvin et al., 2008). Unfortunately, in many cases, such bugs are not discovered until an attack exploits them. Examples of programs with heap related security vulnerabilities include Microsoft Internet Explorer, Macromedia Flash and Macromedia Cold Fusion (CERT, 2002a) to name a few. A successful attack can cost billions of dollars as was the case with the Code Red worm (CERT, 2002b).

Heap memory bugs come in various forms: double free, dangling pointers, reads to uninitialized memory, writes to unallocated memory, and memory leaks. A double free bug occurs when a previously freed memory block is freed incorrectly again. Dangling pointers occur when a freed memory block is referenced. Memory leaks occur when objects that are no longer used are not freed. For long-running applications such as server applications, memory leaks can consume the physical memory slowing down the application significantly and eventually crashing it when it runs out of memory. This emphasizes the need for memory debugging tools to help developers efficiently and thoroughly debug their codes before they are released.

In this paper, we propose Heap Bug Detector (HeapBud), a static tool that analyzes programs written in C and pinpoints potential bugs. HeapBud takes as input the assembly file corresponding to the application. It then generates and analyzes its control flow graph and pointers usage. If potential bugs are found, they are printed on the screen with corresponding line numbers in the original C code. While analyzing the program, HeapBud starts by dividing the program into basic blocks. A basic block is a group of instructions with one entry point (the first instruction) and one exit point (a branch or jump at the end). It then creates the program's control flow graph. The control flow graph is a cyclic directed graph with basic blocks representing its nodes and branch/jump targets representing its arrows. It shows all possible paths between basic blocks based on branch/jump

targets. After that, it identifies pointers in the code and keeps track of the usage of each pointer in the basic blocks (allocation, free, read, write). Finally, it analyzes all possible paths in the control flow graph in search of scenarios that may lead to bugs such as reading the memory pointed to by a previously freed pointer, freeing a pointer twice ...etc.

HeapBud has the following advantages: First, it does not require hardware modification which is usually impractical. Second, it does not require the availability of the original source code since the assembly program can be obtained directly from the binary executable using common tools such as *objdump* in Unix/Linux. This is an important advantage as it allows for checking of bugs in legacy code for which the original code may not be available. Third, it is a static tool and thus does not incur run-time overhead in production runs as do dynamic tools. Finally, it is input-independent as it analyzes all possible paths in the control flow graph regardless of what the input could be. Other previously proposed static tools suffer from this shortcoming as they analyze the program based on specific input scenarios and, therefore, may leave some bugs undetected.

The rest of this paper is organized as follows: Section II discusses the related work, Section III discusses heap bugs, and Section IV introduces the proposed heap bug detector. In Section V we evaluate our approach and in Section VI we summarize our findings.

## II. RELATED WORK

Many software tools have been developed to detect bugs. Those tools fall into two categories: static and dynamic. Static tools analyze the code during development and before release (Musuvathi et al., 2002; Choi et al., 2002; Engler et al., 2003; Hallem et al., 2002). They rely on the static analysis of the code and are usually built around the compiler. HeapBud falls into this category.

Dynamic tools detect bugs at run time. They rely on instrumenting the code with checks as in IBM Purify (IBM), Valgrind (Valgrind, 2000), Intel Thread Checker (Intel), DIDUCE (Hangal et al., 2002), Eraser (Savage et al., 1997), CCured (Necula et al., 2005), and others. The main disadvantage of dynamic tools is that they significantly slow down the application's execution time making them unattractive. This is because instrumented memory references (loads and stores) execute often, and execution of the instrumentation code is interleaved with normal program execution (Shetty et al., 2006). In addition, dynamic tools are input-dependent and thus can only detect bugs if they manifest for the specific input tested. With modern

complex applications, there can be numerous possible inputs and thus the bug may only appear after deployment or if exploited by an attacker.

Hardware support for detecting memory bugs or for supporting debugging has also been proposed such as in HeapMon (Shetty et al., 2006), iWatcher (Zhou et al., 2004a), AccMon (Zhou et al., 2004b), SafeMem (Qin et al., 2005), and DISE (Corliss et al., 2005). Such solutions are unattractive because they require costly hardware modifications to the processor.

## III. HEAP MEMORY BUGS

### A. Memory Leak

A memory leak occurs when a program continuously allocates memory on its heap but fails to deallocate it when the memory will be no longer used. Heap management libraries are designed to recycle deallocated memory for use by newly allocated objects with the goal of keeping the program's memory footprint bound. A common example is when an object is allocated inside the scope of a function and is not deallocated before the function returns. Once the function returns, the object is outside the scope of other program parts and yet the memory remains reserved. A growing memory leak can reduce the performance of the computer to unacceptable levels and can cause the program, in the worst case, to crash. This is especially evident in programs that run for long times such as in server applications (Wikipedia, 2009a).

### B. Dangling Pointer

Dangling pointers refer to pointers that do not point to valid objects in memory. In many programming languages such as C, deallocating an object from memory explicitly or by destroying the stack frame on a function return does not alter associated pointers. Dangling pointers arise when an object is deallocated but the pointer which pointed to the deallocated object is subsequently used by the program before re-allocating new memory to it. Because deallocated memory is recycled and used for allocation of new objects, the dangling pointer now points to an object it was not supposed to point to. If referenced, a silent corruption of data may occur leading to subtle bugs that can be extremely difficult to find, cause segmentation faults (UNIX) or general protection faults (Windows). Furthermore, if the overwritten data is bookkeeping data used by the heap management library, the corruption can cause system instabilities (Wikipedia, 2009b).

### C. Double Free

A double free occurs when the same memory

location is deallocated (freed) twice before it is reallocated. This corrupts the heap management library's bookkeeping information and may corrupt the program's data. This corruption can cause the program to crash or make it vulnerable to a buffer overflow attack. Moreover, double free vulnerabilities can be exploited in denial-of-service attacks (Open Web).

#### D. Invalid Free

Invalid free occurs when the address passed to the function `free()` is invalid. For example, the address may not correspond to the starting location of the heap block or may not point to the heap memory region at all. An invalid free can cause corruption of the program's data or bookkeeping information. This can lead to crashes or incorrect output.

## IV. HEAPBUD

### Running HeapBud

To run HeapBud, the assembly program must be first generated. The assembly program is generated by either compiling the source code itself (if available) or by extracting it from the binary executable using common tools such as `objdump` in Linux. While the design ideas and concepts of HeapBud can be applied to check programs in any instruction set (ISA), the MIPS ISA was chosen in implementing the HeapBud prototype for two reasons: First, MIPS or MIPS-like instruction sets are very common in embedded processors which in turn are mostly programmed in C. Second, the MIPS instruction set is RISC-based and thus has fewer instructions and addressing modes compared to a CISC-based instruction set such as x86. This simplifies the programming of the HeapBud prototype.

We emphasize here that the same design ideas and concepts can be used to extend HeapBud to check programs using other instruction sets such as x86. Moreover, if the source code is available, it can be compiled for a MIPS target, checked using HeapBud, debugged and fixed, and then compiled to the desired instruction set target. HeapBud points out the bugs in the original C program which would manifest whatever the target instruction set was.

Next, the assembly program is used as an input to HeapBud which in turn analyzes the program and prints out a detailed report of possible bugs in it. If the program is compiled with the debugging flag set (e.g. with the `-g` flag in GNU GCC), the exact line numbers in the program where the bugs occur are printed out. Otherwise, only the function names and pointer locations where the bugs occur are printed out.

### HeapBud Implementation

The execution of HeapBud goes through the following internal steps:

#### Step 1: Basic Block Partitioning and Control Flow Graph Generation.

In the first step, HeapBud goes through the program partitioning instructions into basic blocks and generating the control flow graph. A basic block is defined as a maximum sequence of consecutive instructions with the following two properties: First, the control flow can only enter the basic block through the first instruction in it. In other words, none of the instructions inside the basic block (except for the first instruction) can be a target of a branch or jump. Second, the last instruction in the basic block is either a branch or jump to another basic block, or precedes an instruction which is a branch/jump target. In other words, once the first instruction in the basic block is executed, all other instructions will be executed sequentially.

Starting from the first instruction in the program's main function, begin a new basic block and keep adding instructions until a label, branch or jump is encountered indicating the start of a new basic block. In the absence of jumps and labels, control proceeds sequentially from the instruction to the next. The first instruction in a basic block is referred to as a leader instruction. A leader instruction can be:

- The first instruction in a function.
- Any instruction that is the target of a jump or branch.
- An instruction that immediately follows a branch or jump.

Using leader instructions, each basic block consists of a leader instruction and all instructions following it up to, but not including, the next leader instruction.

In constructing the control flow graph, the basic blocks become the nodes and edges represent control transitions between blocks. That is, edges show which blocks can follow which other blocks. There is an edge from block A to block B if and only if it is possible for the first instruction in block B to immediately follow the last instruction in block A. There are two cases in which such an edge could be specified:

- There is a branch or jump from the end of block A to the beginning of block B.

- Block B immediately follows block A in the original order of instructions and block A does not end in an unconditional jump.

Control flow graphs are commonly used in program optimization during compilation.

In addition to finding basic blocks and constructing the control flow graph, this step includes building a functions list.

The following is an example code segment illustrating the partitioning algorithm and control flow graph construction:

```
void print_end()
{
    printf("Program End\n");
}

void print_even_number(int num)
{
    if(num%2 == 0)
        printf("%d is an even number\n",num);
}

main()
{
    int num;
    for(num=1; num<=10; num++)
        print_even_number(num);
    print_end();
}
```

The following code represents the intermediate MIPS assembly code and the highlighted instructions represent the leader instructions in the code. Each line represents a single instruction with the following three fields: the instruction's virtual address, the 32-bit instruction code, and the corresponding assembly representation.

```
004002d0 <print_end>:
4002d0: 27bdffe8 addiu sp,sp,-24
4002d4: afbf0010 sw ra,16(sp)
4002d8: 3c040046 lui a0,0x46
4002dc: 248490d0 addiu a0,a0,-28464
4002e0: 0c100224 jal 400890 <_IO_printf>
4002e4: 8fbf0010 lw ra,16(sp)
4002e8: 27bd0018 addiu sp,sp,24
4002ec: 03e00008 jr ra

004002f0 <print_even_number>:
4002f0: 27bdffe8 addiu sp,sp,-24
4002f4: afbf0010 sw ra,16(sp)
4002f8: afa40018 sw a0,24(sp)
4002fc: 8fa20018 lw v0,24(sp)
400300: 30420001 andi v0,v0,0x1
400304: 00000000 nop
400308: 14400005 bnez v0,40031c
<print_even_number+0x2c>
40030c: 3c040046 lui a0,0x46
400310: 248490e0 addiu a0,a0,-28448
400314: 8fa50018 lw a1,24(sp)
400318: 0c100224 jal 400890 <_IO_printf>
40031c: 8fbf0010 lw ra,16(sp)
400320: 27bd0018 addiu sp,sp,24
400324: 03e00008 jr ra
```

```
00400328 <main>:
400328: 27bdffe0 addiu sp,sp,-32
40032c: afbf0018 sw ra,24(sp)
400330: 24020001 li v0,1
400334: afa20010 sw v0,16(sp)
400338: 8fa20010 lw v0,16(sp)
40033c: 2842000b slti v0,v0,11
400340: 00000000 nop
400344: 14400003 bnez v0,400350 <main+0x28>
400348: 00000000 nop
40034c: 081000da j 400368 <main+0x40>
400350: 8fa40010 lw a0,16(sp)
400354: 0c1000bc jal 4002f0
<print_even_number>
400358: 8fa20010 lw v0,16(sp)
40035c: 24420001 addiu v0,v0,1
400360: afa20010 sw v0,16(sp)
400364: 081000ce j 400338 <main+0x10>
400368: 00000000 nop
40036c: 0c1000b4 jal 4002d0 <print_end>
400370: 8fbf0018 lw ra,24(sp)
400374: 27bd0020 addiu sp,sp,32
400378: 03e00008 jr ra
```

The following three tables represent the functions list, the leader instructions list, and the basic blocks list.

### Functions list

Function Name	Start Address	End Address
<main>	0x400328	0x400378
<print_even_number>	0x4002f0	0x400324
<print_end>	0x4002d0	0x4002ec

### Leader instructions list

Leader Instruction Address	Function
0x400328	<main>
0x400338	<main>
0x400348	<main>
0x400350	<main>
0x400368	<main>
0x4002f0	<print_even_number>
0x40030c	<print_even_number>
0x40031c	<print_even_number>
0x4002d0	<print_end>

### Basic blocks list

ID	Next 1	Next 2	Basic Block Contents
1	2	-	400328: 27bdffe0 addiu sp,sp,-32 40032c: afbf0018 sw ra,24(sp) 400330: 24020001 li v0,1 400334: afa20010 sw v0,16(sp)
2	3	4	400338: 8fa20010 lw v0,16(sp) 40033c: 2842000b slti v0,v0,11 400340: 00000000 nop 400344: 14400003 bnez v0,400350 <main+0x28>

3	5	-	400348: 00000000 nop 40034c: 081000da j 400368 <main+0x40>
4	2	-	400350: 8fa40010 lw a0,16(sp) 400354: 0c1000bc jal 4002f0 <print_even_numbers> 400358: 8fa20010 lw v0,16(sp) 40035c: 24420001 addiu v0,v0,1 400360: afa20010 sw v0,16(sp) 400364: 081000ce j 400338 <main+0x10>
5	-	-	400368: 00000000 nop 40036c: 0c1000b4 jal 4002d0 <print_end> 400370: 8fbf0018 lw ra,24(sp) 400374: 27bd0020 addiu sp,sp,32 400378: 03e00008 jr ra
6	7	8	4002f0: 27bdffe8 addiu sp,sp,-24 4002f4: afbf0010 sw ra,16(sp) 4002f8: afa40018 sw a0,24(sp) 4002fc: 8fa20018 lw v0,24(sp) 400300: 30420001 andi v0,v0,0x1 400304: 00000000 nop 400308: 14400005 bnez v0,40031c<print_even_number+0x2c
7	8	-	40030c: 3c040046 lui a0,0x46 400310: 248490e0 addiu a0,a0,-28448 400314: 8fa50018 lw a1,24(sp) 400318: 0c100224 jal 400890 <_IO_printf>
8	-	-	40031c: 8fbf0010 lw ra,16(sp) 400320: 27bd0018 addiu sp,sp,24 400324: 03e00008 jr ra
9	-	-	4002d0: 27bdffe8 addiu sp,sp,-24 4002d4: afbf0010 sw ra,16(sp) 4002d8: 3c040046 lui a0,0x46 4002dc: 248490d0 addiu a0,a0,-28464 4002e0: 0c100224 jal 400890 <_IO_printf> 4002e4: 8fbf0010 lw ra,16(sp) 4002e8: 27bd0018 addiu sp,sp,24 4002ec: 03e00008 jr ra

## Step 2: Dynamic Pointers and Dynamic Memory Operations Identification.

In C/C++, memory is allocated on the heap through calls to one of the functions *malloc()*, *calloc()*, *realloc()*, or to the *new* operator. The argument passed to the function is the required allocation size in bytes and the returned value is a pointer to the allocated heap block if the operation is successful. The required size in bytes is passed through one of the argument registers and the pointer address is returned through one of the value registers. To identify dynamic pointers in the program, HeapBud traverses basic blocks in search for calls to one of the above functions. Once a pointer is identified, it is added to the dynamic memory pointers list.

Next, for each pointer in the dynamic memory pointers list, HeapBud traverses through all basic blocks of the function where it is declared and identifies all operations on the pointer including memory loads, memory stores, dynamic memory allocation, and dynamic memory deallocation. In addition, any pointer aliases are also identified. These operations are added to the dynamic memory pointers list entry corresponding to that pointer. The above operations can be easily identified in the assembly program. For example, if the pointer address is passed as an argument to the *free* function then there is a deallocation operation. Moreover, if the pointer is used as an effective address in a memory load or store instruction, then a dynamic memory read or write operation is identified. Finally, if the pointer address is stored in another pointer, then this pointer becomes an alias to the original pointer.

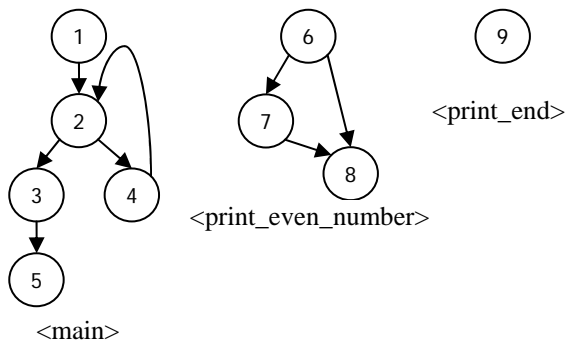
The above step is illustrated in the example below.

```
main()
{
    int *p1,*p2;
    p1 = (int *)malloc(sizeof(int));
    *p1 = 5;
    p2 = p1;
    printf("%d\n", *p2);
    free(p1);
    p1 = p2 = NULL;
}
```

Below is an assembly translation of the code above where the pointer operations are highlighted. This is followed by the dynamic memory pointers list:

```
004002d0 <main>:
4002d0: 27bdffe0 addiu sp,sp,-32
4002d4: afbf0018 sw ra,24(sp)
4002d8: 24040004 li a0,4
4002dc: 0c100ca2 jal 403288
<__libc_malloc>
```

Finally, the figure below shows the program's control flow graph. The number inside a node represents the basic block's number while edges show possible control flow transitions. Note that there are three separate graphs each representing a function from the three program functions (*main*, *print\_end*, and *print\_even\_number*). This is because transitions on function calls and function returns are not shown here.



Control Flow Graph

```

4002e0: afa20010 sw    v0,16(sp)
4002e4: 8fa30010 lw    v1,16(sp)
4002e8: 24020005 li    v0,5
4002ec: ac620000 sw    v0,0(v1)
4002f0: 8fa20010 lw    v0,16(sp)
4002f4: afa20014 sw    v0,20(sp)
4002f8: 8fa20014 lw    v0,20(sp)
4002fc: 3c040046 lui   a0,0x46
400300: 24849080 addiu a0,a0,-28544
400304: 8c450000 lw    a1,0(v0)
400308: 0c100210 jal   400840
40030c: 8fa40010 lw    a0,16(sp)
400310: 0c100d1c jal   403470
400314: afa00014 sw    zero,20(sp)
400318: afa00010 sw    zero,16(sp)
40031c: 8fbf0018 lw    ra,24(sp)
400320: 27bd0020 addiu sp,sp,32
400324: 03e00008 jr    ra

```

Dynamic Memory Pointers List:

Pointer Address	Function	Pointer Aliases	Operation Address	Operation Type
16(sp)	<main>	20(sp) <del>20(sp)</del>	0x4002e0	Allocation
			0x4002ec	Memory write
			0x4002f4	Pointer alias
			0x400304	Memory read
			0x40030c	Free
			0x400314	Alias remove
			0x400318	Pointer reassignment

**Step 3: Control Flow Graph Traversal and Bug Detection.**

For each pointer in the dynamic memory pointers list, HeapBud starts the control flow graph traversal from the basic block where the pointer is first assigned to an allocated dynamic memory location. Then, all possible paths are considered and traversed from that point until the last basic block in the function is reached. This highlights HeapBud's main advantage over existing heap bug detection tools in that it is input-independent and considers all possible paths in the program. Input-dependent tools may ignore paths not taken for the input examined although those ignored paths may lead to a bug. Examining all possible inputs is quite impossible for modern complex applications because they usually have numerous input combination possibilities.

The traversal is repeated for each memory pointer in each function. To avoid infinite loops in the graph traversal, each sequence of visited basic blocks is recorded to avoid traversing it more than once.

During the control flow graph traversal, the state of

the memory pointer is maintained: *NULL*, *allocated* or *deallocated*. Bugs are detected by examining any operation on the pointer and the pointer's state at that point.

- A possible *double free* bug is detected if the memory operation is a *deallocation* but the pointer's state is already *deallocated*.
- A possible *memory leak* bug is detected if the memory operation is an *allocation* but the pointer's state is already *allocated*.
- A possible *invalid free* bug is detected if the memory operation is a *deallocation* but the pointer's state is *NULL*.
- A possible *dangling pointer* bug is detected if the operation is a *read* or *write* but the pointer's state is *NULL* or *deallocated*.

Finally, HeapBud maps each assembly instruction that may cause a bug to its corresponding source code line number and prints the source code's line number, bug type, pointer state, and line number where the pointer's state was last modified on the screen.

Limitations of HeapBud

The above description of HeapBud's implementation points to its main limitation. HeapBud can successfully identify bugs within the scope of a function. However, with its current implementation, HeapBud cannot identify bugs when operations are performed across multiple functions. This happens when pointers are passed between functions as arguments or are global pointers. These two cases are not very common and adding this functionality to HeapBud is not very difficult but is set aside for future work.

V. EVALUATION AND TESTING

To evaluate HeapBud's capabilities, we use three test programs: *test1.c*, *test2.c*, and *test3.c*. The three programs contain the most common heap bug scenarios. We show HeapBud's output which is printed on the screen along with the output of another commonly used bug detection tool: Valgrind [Valgrind, 2000].

test1.c

```

1. #include<stdio.h>
2. #include<stdlib.h>
3. #include<string.h>
4. void dangling pointer()
5. {
6.     int x = 0,*p = NULL;
7.     free(p);
8.     p = (int *)malloc(4);
9.     printf("input the value of x?");
10.    scanf("%d",&x);

```

```

11.     if(x) free(p);
12.     *p = 5;
13.     printf("%d", *p);
14.     }
15.     main()
16.     {
17.         dangling_pointer();
18.     }

```

### HeapBud Output:

```

# HeapBud test1
test1.c:7: invalid free to unallocated
pointer.
*****
test1.c:12: invalid memory write.
test1.c:11: pointer is freed at this line,
dangling pointer.
*****
test1.c:13: invalid memory read.
test1.c:11: pointer is freed at this line,
dangling pointer.
*****
test1.c:14: function end without free the
pointer may cause memory leak.
test1.c:8: pointer allocated at this line.
*****

```

### Valgrind Output: input x = 0:

```

# valgrind test1
input the value of x?0
==2985==
==2985== ERROR SUMMARY: 0 errors from 0 contexts
(suppressed: 12 from 1)
==2985== malloc/free: in use at exit: 4 bytes in 1
blocks.
==2985== malloc/free: 1 allocs, 0 frees, 4 bytes
allocated.
==2985== For counts of detected errors, rerun
with: -v
==2985== searching for pointers to 1 not-freed
blocks.
==2985== checked 52,400 bytes.
==2985==
==2985== LEAK SUMMARY:
==2985== definitely lost: 4 bytes in 1 blocks.
==2985== possibly lost: 0 bytes in 0 blocks.
==2985== still reachable: 0 bytes in 0 blocks.
==2985== suppressed: 0 bytes in 0 blocks.
==2985== Rerun with --leak-check=full to see
details of leaked memory.

```

### Valgrind Output: input x = 1:

```

# valgrind test1
input the value of x?1
==3000==
==3000== Invalid write of size 4
==3000== at 0x80484B6: dangling pointer
(test1.c:12)
==3000== by 0x80484E8: main (test1.c:17)
==3000== Address 0x4028028 is 0 bytes inside a
block of size 4 free'd
==3000== at 0x400590A: free
(vg_replace_malloc.c:323)
==3000== by 0x80484B2: dangling pointer
(test1.c:11)
==3000== by 0x80484E8: main (test1.c:17)
==3000==
==3000== Invalid read of size 4
==3000== at 0x80484BF: dangling pointer
(test1.c:13)
==3000== by 0x80484E8: main (test1.c:17)
==3000== Address 0x4028028 is 0 bytes inside a
block of size 4 free'd
==3000== at 0x400590A: free
(vg_replace_malloc.c:323)

```

```

==3000== by 0x80484B2: dangling pointer
(test1.c:11)
==3000== by 0x80484E8: main (test1.c:17)
==3000==
==3000== ERROR SUMMARY: 2 errors from 2 contexts
(suppressed: 12 from 1)
==3000== malloc/free: in use at exit: 0 bytes in 0
blocks.
==3000== malloc/free: 1 allocs, 1 frees, 4 bytes
allocated.
==3000== For counts of detected errors, rerun
with: -v
==3000== All heap blocks were freed -- no leaks
are possible.

```

In the first test, there are 4 possible bugs: First, there is an *invalid free* bug on line 7 when the pointer *p* is deallocated while its value is NULL. Second, there is a possible *dangling pointer* bug on line 12 where there is a write to the memory location pointed to by *p* when the pointer *p* may be deallocated on line 11. Third, there is a possible *dangling pointer* bug on line 13 where there is a read to the memory location pointed to by *p* when the pointer *p* may be deallocated on line 11. Fourth, pointer *p* may not be deallocated when the function returns in case the condition on line 11 is false. This points to a possible *memory leak*. Examining the output of HeapBud, we find that HeapBud successfully finds all four possible bugs and highlights the exact line numbers. On the other hand, Valgrind cannot identify some possible bugs when the input *x* is assumed to be 0 although it can identify them when the value of *x* is assumed to be 1. This again points to the main drawback of input-dependent tools in that they may miss bugs for certain input combinations if not explicitly tested. In both cases, Valgrind does not detect the possible *invalid free* bug on line 7.

### test2.c

```

1.     #include<stdio.h>
2.     #include<stdlib.h>
3.     #include<string.h>
4.     void memory_leak()
5.     {
6.         int x = 0;
7.         char *str = NULL;
8.         printf("input the value of x?");
9.         scanf("%d", &x);
10.        while(x) str = (char *)malloc(20);
11.        str = (char *)malloc(20);
12.        strcpy(str, "Heap Bug Detector");
13.    }
14.    main()
15.    {
16.        memory_leak();
17.    }

```

### HeapBud Output:

```

# HeapBud test2
test2.c:10: this operation may cause memory
leak.
test2.c:10: pointer allocated at this line.
*****

```

```

test2.c:11: this operation may cause memory
leak.
test2.c:10: pointer allocated at this line.
*****
test2.c:13: function end without
deallocation for the pointer may cause
memory leak.
test2.c:11: pointer allocated at this line.
*****

```

### Valgrind Output: input x = 0:

```

# valgrind test2
input the value of x?0
==3023==
==3023== ERROR SUMMARY: 0 errors from 0 contexts
(suppressed: 12 from 1)
==3023== malloc/free: in use at exit: 20 bytes in
1 blocks.
==3023== malloc/free: 1 allocs, 0 frees, 20 bytes
allocated.
==3023== For counts of detected errors, rerun
with: -v
==3023== searching for pointers to 1 not-freed
blocks.
==3023== checked 52,416 bytes.
==3023==
==3023== LEAK SUMMARY:
==3023== definitely lost: 0 bytes in 0 blocks.
==3023== possibly lost: 0 bytes in 0 blocks.
==3023== still reachable: 20 bytes in 1 blocks.
==3023== suppressed: 0 bytes in 0 blocks.
==3023== Rerun with --leak-check=full to see
details of leaked memory.

```

### Valgrind Output: input x = 1:

```

# valgrind test2
input the value of x?1
==3023==
Out of memory

```

The second test highlights a major drawback in Valgrind in which it is stuck in an infinite loop when the value of x is 1 because it is a dynamic tool. The program eventually runs out of memory and crashes. HeapBud does not suffer from this drawback and successfully identifies the possible *memory leak* bug.

### test3.c

```

1.  #include<stdio.h>
2.  #include<stdlib.h>
3.  #include<string.h>
4.  void double_free()
5.  {
6.      int x = 0;
7.      char *str = (char *)malloc(20);
8.      printf("input the value of x?");
9.      scanf("%d",&x);
10.     if(x) free(str);
11.     free(str);
12. }
13. main()
14. {
15.     double_free();
16. }

```

### HeapBud Output:

```

# HeapBud test3
test3.c:11: this operation may cause double
free.
test3.c:10: pointer is freed at this line.
*****

```

### Valgrind Output: input x = 0:

```

# valgrind test3
input the value of x?0
==3098==
==3098== ERROR SUMMARY: 0 errors from 0 contexts
(suppressed: 12 from 1)
==3098== malloc/free: in use at exit: 0 bytes in 0
blocks.
==3098== malloc/free: 1 allocs, 1 frees, 20 bytes
allocated.
==3098== For counts of detected errors, rerun
with: -v
==3098== All heap blocks were freed -- no leaks
are possible.

```

### Valgrind Output: input x = 1:

```

# valgrind test3
input the value of x?1
==3111==
==3111== Invalid free() / delete / delete[]
==3111== at 0x400590A: free
(vg replace malloc.c:323)
==3111== by 0x80484AB: double free (test3.c:11)
==3111== by 0x80484C3: main (test3.c:15)
==3111== Address 0x4028028 is 0 bytes inside a
block of size 20 free'd
==3111== at 0x400590A: free
(vg replace malloc.c:323)
==3111== by 0x80484A0: double free (test3.c:10)
==3111== by 0x80484C3: main (test3.c:15)
==3111==
==3111== ERROR SUMMARY: 1 errors from 1 contexts
(suppressed: 12 from 1)
==3111== malloc/free: in use at exit: 0 bytes in 0
blocks.
==3111== malloc/free: 1 allocs, 2 frees, 20 bytes
allocated.
==3111== For counts of detected errors, rerun
with: -v
==3111== All heap blocks were freed -- no leaks
are possible.

```

In the third, a possible *double free* bug is successfully detected by HeapBud but is only detected by Valgrind when the input x is equal to 1.

## VI. CONCLUSION

We introduced HeapBud, an efficient heap memory bug detector that can comprehensively detect heap related bugs in C programs such as double free, dangling pointer, memory leak, and invalid free bugs. Those bugs, if left undetected, may lead to serious program problems such as security attacks, unacceptable slowdowns, crashes, and incorrect output.

HeapBud is a static tool and detects bugs by analyzing all possible paths in the program's control flow graph in addition to all pointer usages in search of possible scenarios leading to memory bugs. HeapBud does not require hardware modification, does not require the availability of the original source code, does not incur runtime overhead, and is input independent, making it an attractive solution.

## REFERENCES

- [1] Aho, A., Lam, M., Sethi, R., & Ullman, J. (2006). *Compilers: Principles, Techniques, and Tools*. 2<sup>nd</sup> Edition. Addison-Wesley.
- [2] Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., & Sridharan, M. (2002). Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- [3] Corliss, M., Lewis, E., & Roth, A. (2005). Low-Overhead Interactive Debugging via Dynamic Instrumentation with DISE. *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*.
- [4] Engler, D., & Ashcraft, K. (2003). RacerX: Effective, Static Detection of Race Conditions and Deadlocks. *Proceedings of the 19th ACM Symposium on Operating Systems Principles*.
- [5] Gottbrath, C. (2008). *Catch that Memory Bug Before it Catches You*. From <http://www.linuxinsider.com/story/62579.html?wlc=1243446219>.
- [6] Hallem, S., Chelf, B., Xie, Y., & Engler, D. (2002). A System and Language for Building System-Specific, Static Analyses. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- [7] Hangal, S., & Lam, M. (2002). Tracking Down Software Bugs Using Automatic Anomaly Detection. *Proceedings of the 24<sup>th</sup> International Conference on Software Engineering*.
- [8] IBM Corporation. *Rational Software*. From <http://www-306.ibm.com/software/rational/>.
- [9] Intel Corporation. *Intel Thread Checker*. From <http://software.intel.com/en-us/intel-thread-checker/>.
- [10] Lvin, V., Novark, G., Berger, E., & Zorn, B. (2008). Archipelago: Trading Address Space for Reliability and Security. *Proceedings of the ACM International Conferences on Architectural Support for Programming Languages and Operating Systems*.
- [11] Musuvathi, M., Park, D., Chou, A., Engler, D., & Dill, D. (2002). CMC: A Pragmatic Approach to Model Checking Real Code. *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*.
- [12] Necula, G., Condit, J., Harren, M., McPeak, S., & Weimer, W. (2005). CCured: Type-Safe Retrofitting of Legacy Code. *ACM Transactions on Programming Languages and Systems*.
- [13] Open Web Application Security Project. *Double Free*. From [http://www.owasp.org/index.php/Double\\_Free](http://www.owasp.org/index.php/Double_Free).
- [14] Qin, F., Lu, S., & Zhou, Y. (2005). SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*.
- [15] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., & Anderson, T. (1997). Eraser: A Dynamic Data Race Detector for Multithreaded Program. *ACM Transactions on Computer Systems, No. 4*.
- [16] Shetty, R., Kharbutli, M., Solihin, Y., Prvulovic, M. (2006). HeapMon: A Helper-thread Approach to Programmable, Automatic, and Low-overhead Memory Bug Detection. *IBM Journal of Research and Development, 50(2-3): 261-276*.
- [17] CERT: United States Computer Emergency Readiness Team (2002a). *Buffer Overflow in Microsoft Internet Explorer*. From <http://www.us-cert.gov/cas/techalerts/TA04-315A.html>.
- [18] CERT: United States Computer Emergency Readiness Team (2002b). *Code Red Worm Exploiting Buffer Overflow in IIS Indexing Service DLL*. From <http://www.cert.org/advisories/CA-2001-19.html>.
- [19] Valgrind Developers (2000). *Valgrind*. From <http://www.valgrind.org/>.
- [20] Wikipedia (2009a). *Memory Leaks*. From [http://en.wikipedia.org/wiki/Memory\\_leak](http://en.wikipedia.org/wiki/Memory_leak).
- [21] Wikipedia (2009b). *Dangling Pointer*. From [http://en.wikipedia.org/wiki/Dangling\\_pointer](http://en.wikipedia.org/wiki/Dangling_pointer).
- [22] Zhou, P., Qin, F., Liu, W., Zhou, Y., & Torellas, J. (2004a). iWatcher: Efficient Architectural Support for Software Debugging. *Proceedings of the 31st Annual International Symposium on Computer Architecture*.
- [23] Zhou, P., Liu, W., Fei, L., Lu, S., Qin, F., Zhou, Y., Midkiff, S., & Torellas, J. (2004b). AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-Based Invariants. *Proceedings of the 37th Annual International Symposium on Microarchitecture*.