# Binary Transformation of Applications to run on Bare PCs

Rasha Almajed
Department of Computer & Information Sciences
Towson University
Towson, MD 21252
ralmajed@towson.edu

Ramesh Karne
Department of Computer & Information Sciences
Towson University
Towson, MD 21252x 6221
rkarne@towson.edu

Alexander Wijesinha
Department of Computer & Information Sciences
Towson University
Towson, MD 21252
awijesinha@towson.edu

## ABSTRACT

Operating [1] systems or kernels assist in running conventional applications. Bare machine computing (BMC) applications run on bare machines without using an operating system or a kernel. BMC applications are thus independent of any execution platform promoting longevity and reducing obsolescence. We investigate the binary transformation of conventional applications enabling them to run on bare machines. Three sample applications are used to demonstrate the binary transformation process and show the feasibility of our approach. Two Visual Studio applications are transformed to run on a bare PC and one large bare PC application binary is linked dynamically and transformed to run with a bare PC Web server. The transformation process and the dynamic linking of binary modules help to identify design and research issues. Our work lays a foundation to achieve the ultimate goal of making applications independent of computing platforms and environments.

## CCS CONCEPTS

• **Software and its engineering~Software reverse engineering** • *Social and professional topics~Software reverse engineering*

## KEYWORDS

binary transformation, reverse engineering, bare PC, bare machine computing (BMC), computer applications

**ACM Reference format:**

R. Almajed, R. Karne, and A. Wijesinha. 2019. In *Proceedings of ACM SAC Conference, Pau, France, April 8-12, 2019 (SAC'19)*, 8 pages. DOI: xx.xxxx/xxx_x

## 1 INTRODUCTION

Many complex bare machine computing (BMC) applications have been developed previously [1-5]. Instead of developing applications that can run on bare PCs or bare machines, which requires considerable effort and time, we consider the binary transformation of existing conventional applications to run on bare machines. Without having ownership or affinity, users will be able to use computers anywhere and own their applications with total control. When the OS or kernel is eliminated, the layered approach to computing is replaced with an application-centric approach wherein applications directly interface with hardware. Potential advantages of BMC applications include automatic ubiquity and the homogenization of hardware and software. Duplication of hardware and software will also be avoided when the BMC paradigm is used in pervasive devices.

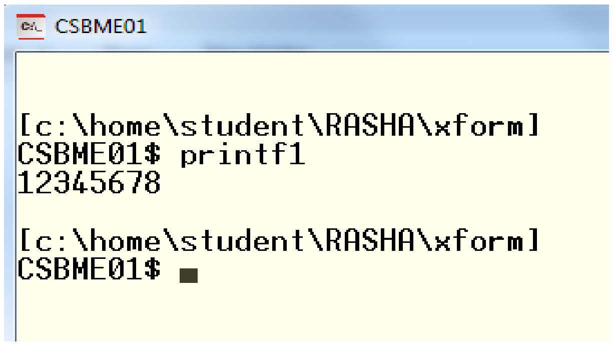
There are two possible approaches to transforming computer applications to run on bare PCs. The first is a source code approach such as that used to transform SQLite to run on a bare PC [6]. This approach requires the source code to be compiled without using any system libraries. In the BMC application, header files have to be replaced with direct hardware and software interfaces, which are included with the program. As the SQLite package is amalgamated and contains only a few source and header files, this technique was successful. Unfortunately, the compilation process becomes tedious and difficult when there are more source and header files, and also if the header files have a long chain of include files. A second approach is to perform a binary transformation using the existing binary executable. In this case, the transformed binary code for the target application has to run in the bare PC with the support of another application running in the machine.

The binary transformation approach is challenging since it requires running foreign binary code with an existing bare BMC application, which is a single monolithic executable that is statically bound to memory. The binary code is a separate executable designed to run with a given OS or kernel and is compiled with its separate address space. We assume that the binary executable is architecturally compatible with the BMC application running in the machine (both x86 and Microsoft executables). We identify the relevant design issues and describe a methodology for binary transformation. We select three applications to illustrate our approach: printing data (bPrintf), bubble sort (bSort) and a binary network interface driver (bNIC). Although bPrintf and bSort are trivial programs in a conventional sense, they serve to identify critical issues that need to be resolved during the binary transformation process enabling their executables to run on a bare PC. The bNIC application is a large binary executable that represents Intel gigabit NIC device driver code, which works with an existing bare PC networking application [7]. We show how to transform this binary driver so that it can run with a BMC application. The rest of this paper is organized as follows. Section 2 describes the binary transformation process for the three applications: print, sort, and a network interface device driver. Section 3 discusses issues requiring further research and the significance of the present work. Section 4 presents related work, and section 5 contains the conclusion.

## 2 BINARY TRANSFORMATIONS

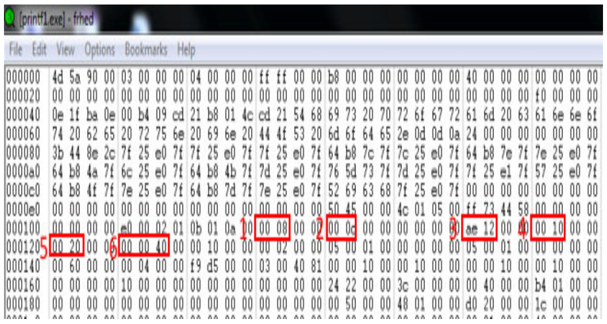
### 2.1 Print Application (bPrintf)

We first consider the binary transformation process for a simple "printf" program written in Visual Studio (VS). The executable has 6144 bytes or 12 sectors. The output of this executable running in DOS window is shown in Fig. 1. Notice that even this single C statement resulted in a large binary file. The header section in the file contains the attributes related to the binary executable as shown in Fig. 2. The PEView tool [8] tool generates file parameters from the header section as shown in Fig. 3, which include the labeled data items 1-6. The Objdump tool [9] generates an assembly listing of this binary file, which is over ten pages. The PEExplorer tool [10] is used to view calls in the assembly code. A partial list of these calls is shown in Fig. 4. Using the above tools, we obtain the structure of the binary executable, which is useful for understanding the binary transformation technique. The following subsections describe the steps to transform this application.



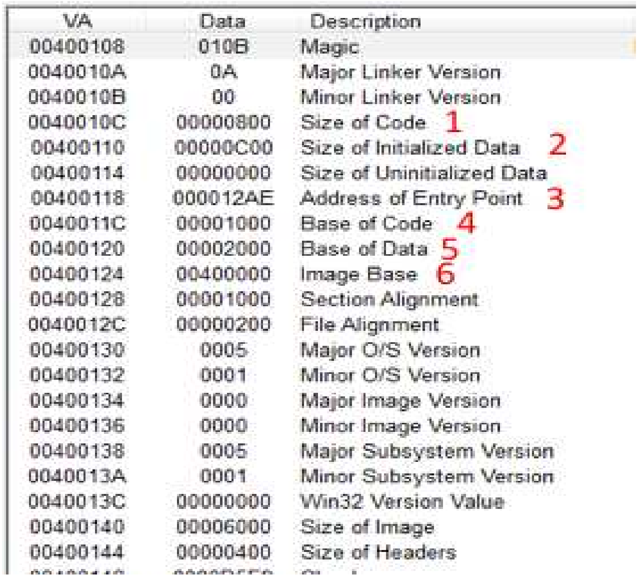
```
[c:\home\student\RASHA\xform]
CSBME01$ printf1
12345678

[c:\home\student\RASHA\xform]
CSBME01$ 
```

Figure 1: Printf.exe Output on DOS Windows.



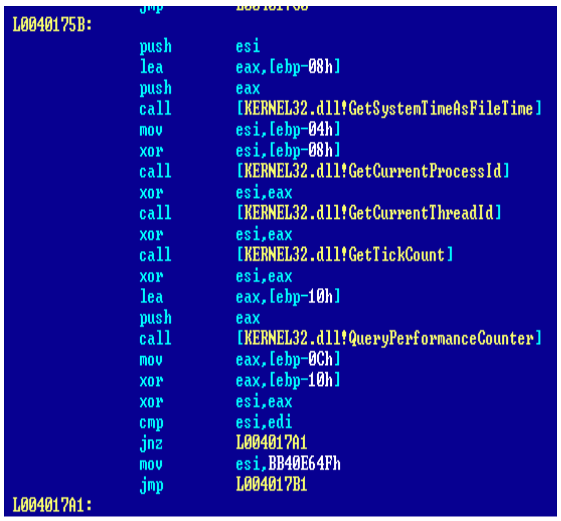
```
000000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 f0 00 00 00
000040  0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f
000060  74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00
000080  3b 44 8e 2c 7f 25 e0 7f 7f 25 e0 7f 7f 25 e0 7f 64 b8 7c 7f 7c 25 e0 7f 64 b8 7e 7f 7e 25 e0 7f
0000a0  64 b8 4a 7f 6c 25 e0 7f 64 b8 4b 7f 7d 25 e0 7f 76 5d 73 7f 7d 25 e0 7f 7f 25 e1 7f 57 25 e0 7f
0000c0  64 b8 4f 7f 7e 25 e0 7f 64 b8 7d 7f 7e 25 e0 7f 52 69 63 68 7f 25 e0 7f 00 00 00 00 00 00 00 00
0000e0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 50 45 00 00 4c 01 05 00 ff 73 44 58 00 00 00 00
000100  00 00 00 00 e0 00 02 01 0b 01 0a 00 00 08 00 00 00 0c 00 00 00 00 00 00 ae 12 00 00 00 10 00 00
000120  00 20 00 00 00 00 40 00 00 10 00 00 00 02 00 00 05 00 01 00 00 00 00 00 05 00 01 00 00 00 00 00
000140  00 60 00 00 00 04 00 00 f9 d5 00 00 03 00 40 81 00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00
000160  00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 24 22 00 00 3c 00 00 00 00 40 00 00 b4 01 00 00
000180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 50 00 00 48 01 00 00 d0 20 00 00 1c 00 00 00
```

Figure 2: Printf.exe Header Section.



| VA | Data | Description |
|---|---|---|
| 00400108 | 010B | Magic |
| 0040010A | 0A | Major Linker Version |
| 0040010B | 00 | Minor Linker Version |
| 0040010C | 00000800 | Size of Code |
| 00400110 | 00000C00 | Size of Initialized Data |
| 00400114 | 00000000 | Size of Uninitialized Data |
| 00400118 | 000012AE | Address of Entry Point |
| 0040011C | 00001000 | Base of Code |
| 00400120 | 00002000 | Base of Data |
| 00400124 | 00400000 | Image Base |
| 00400128 | 00001000 | Section Alignment |
| 0040012C | 00000200 | File Alignment |
| 00400130 | 0005 | Major O/S Version |
| 00400132 | 0001 | Minor O/S Version |
| 00400134 | 0000 | Major Image Version |
| 00400136 | 0000 | Minor Image Version |
| 00400138 | 0005 | Major Subsystem Version |
| 0040013A | 0001 | Minor Subsystem Version |
| 0040013C | 00000000 | Win32 Version Value |
| 00400140 | 00006000 | Size of Image |
| 00400144 | 00000400 | Size of Headers |

Figure 3: Printf.exe Parameter's from PE View.



```
L0040175B:
            push    esi
            lea     eax,[ebp-08h]
            push    eax
            call    [KERNEL32.dll!GetSystemTimeAsFileTime]
            mov     esi,[ebp-04h]
            xor     esi,[ebp-08h]
            call    [KERNEL32.dll!GetCurrentProcessId]
            xor     esi,eax
            call    [KERNEL32.dll!GetCurrentThreadId]
            xor     esi,eax
            call    [KERNEL32.dll!GetTickCount]
            xor     esi,eax
            lea     eax,[ebp-10h]
            push    eax
            call    [KERNEL32.dll!QueryPerformanceCounter]
            mov     eax,[ebp-0Ch]
            xor     eax,[ebp-10h]
            xor     esi,eax
            cmp     esi,edi
            jnz     L004017A1
            mov     esi,BB40E64Fh
            jmp     L004017B1
L004017A1:
```

Figure 4: PEExplorer View of Printf.exe Assembly Code.

*2.1.1 Host Application.* As the binary executable can only run in an OS environment, all the OS related interfaces must be removed and the necessary bare PC interfaces must be plugged into the code. In order to provide the bare PC interfaces, a bare PC host application should be running to load and run the program. Since the host application is compiled and linked separately from the binary executable, a special task is needed to provide a host environment for the binary executable. The BMC architecture includes a main task, a receive task and application or protocol-related tasks such as an HTTP task [1], a TLS task [4] and a SQLite task [5]. A running task suspends and returns control to the main task when it has to wait for an event and it resumes when the event occurs. The task chosen to provide a host environment for the binary executable in this case is a SQLite task. The implementation of this mechanism is shown in Fig. 5. The binary executable runs (Task1), whenever the state flag is set to 100 and the original task runs whenever the state flag is set to 101. The original task SQLite is preserved using this mechanism and the binary executable runs under the existing task thus avoiding new task creation. The SwitchEIP() function will switch from the original task to the binary executable as shown in Fig. 5 by modifying the EIP (program counter) in the TSS. The binary executable layout in memory obtained using the PEView tool ry is shown in Fig. 6.

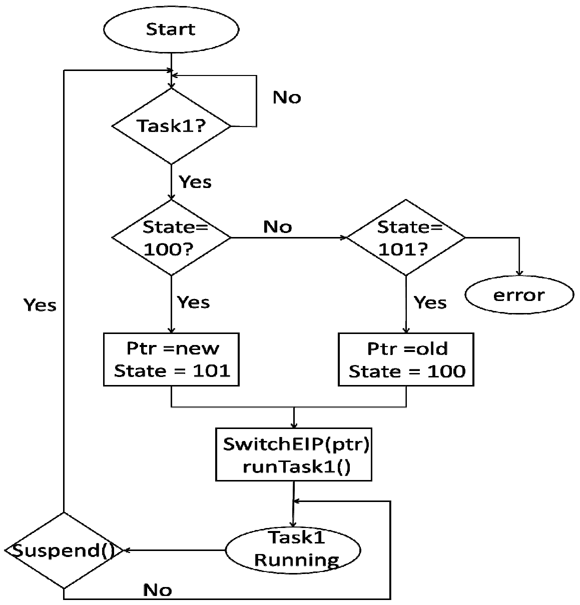


fig6

**Figure 5: The SwitchEIP() Function.**

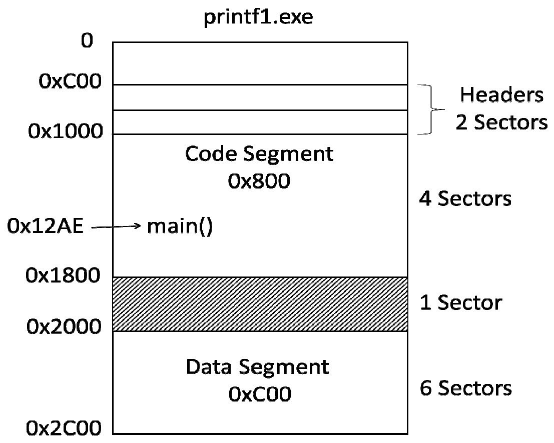


**Figure 6: Binary Executable Map.**

*2.1.2 Transformation Process.* The binary executable is disassembled to generate its assembly code. At an assembly level, all OS related calls must be realized through "call" functions. There are internal and external (OS related) calls in the assembly program. Internal calls have no effect on the transformation process. The PEExplorer tool statically helps to identify these calls.

In order to run the target binary code on a bare PC, we need to replace all system calls with essential and equivalent bare PC interfaces. The major problem is to understand the functionality of the system calls and replace them in the binary code. Some calls may be replaced with equivalent interfaces, others may not have a direct equivalent, and some may not be required. This problem is not same as that encountered in a virtual machine, where OS calls can be mapped from one OS to another. The BMC paradigm assumes no OS concepts and is not a centralized approach. We use a manual approach for the transformation process to help identify the relevant design issues. Although the external calls must be understood, the functionality of the assembly code is not relevant for the binary transformation process.

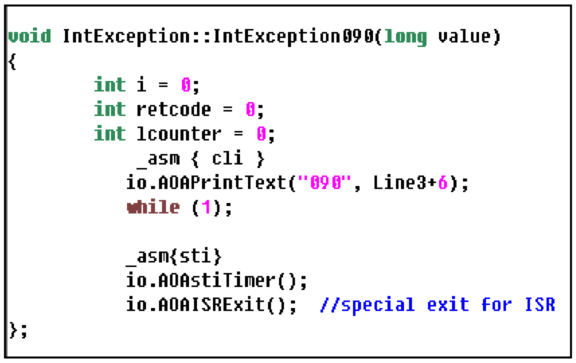
The following steps are executed manually to transform the binary executable.

1. Using PEExplorer, identify "call" locations (system calls and DLLs) in the binary code
2. Understand call parameters, format of assembly listing and parameters passing
3. Implement equivalent calls in the BMC as needed or use existing calls
4. Replace OS calls with the BMC calls in the binary code

The entry point to the code as shown in Fig. 2 (0x12AE) is used to compute the actual entry point (0x12AE – 0x1000 + 0x400 = 0x6AE), which is used to start the program. 0x1000 is the executable start point after linking and 0x400 is size of the header section.

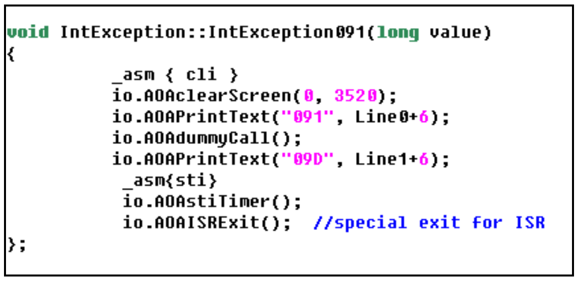
Instead of replacing all system calls at once, a hill-climbing methodology is adopted by moving from one call to the next in the code and dealing with one call at a time. We use interrupts to handle the external calls. When a given interrupt is invoked, the control is returned to an application program. When a current call is resolved, the next call is replaced with an INT 90 in the binary executable. The code is shown in Fig. 7. When the program runs, the interrupt prints a message and executes while (1). This will assure that the program control reached this particular call, where there is an interrupt. If it is not a required call for BMC, this call will be replaced with another interrupt INT 91 (in Fig. 8), which will use a dummy call and simply return to the next instruction. If it is a required call, INT 92 is used to substitute the system call with the BMC call (Fig. 9). Note that the hex value (0x12345678) printed was passed through the stack. Its address in the stack is captured as shown in Fig. 9 and the offset to the BMC call is also computed. Passing parameters in the stack needs to be mapped to the BMC call parameters as illustrated here. Parameter capturing is the most difficult part of this transformation.

A screenshot of the running program is shown in Fig. 10. This figure shows the same binary executable running in a loop indicating that a binary executable can run while the host program is running. In this example, INT 91 was used in 12 places and INT 92 was used in just one call. The "I am here" message is printed in a dummy function call as shown in the figure. INT 91 and 92 are shown in the figure, but INT 90 is not shown because that run is in a different screen output. The rest of the information on the screen is used for debugging purposes.



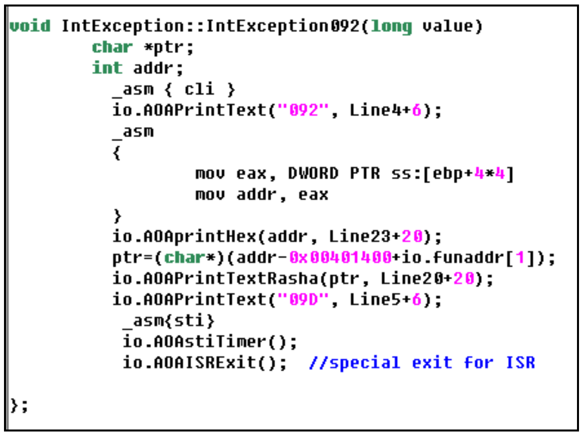
```
void IntException::IntException090(long value)
{
        int i = 0;
        int retcode = 0;
        int lcounter = 0;
            _asm { cli }
           io.AOAPrintText("090", Line3+6);
           while (1);

           _asm{sti}
           io.AOAstiTimer();
           io.AOAISRExit();  //special exit for ISR
};
```

**Figure 7: Int 90.**



```
void IntException::IntException091(long value)
{
          _asm { cli }
          io.AOAclearScreen(0, 3520);
          io.AOAPrintText("091", Line0+6);
          io.AOAdummyCall();
          io.AOAPrintText("09D", Line1+6);
           _asm{sti}
           io.AOAstiTimer();
           io.AOAISRExit();  //special exit for ISR
};
```

**Figure 8: Int 91.**

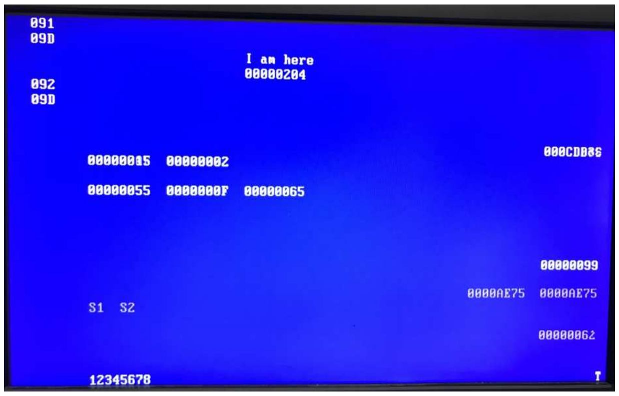


```
void IntException::IntException092(long value)
        char *ptr;
        int addr;
          _asm { cli }
          io.AOAPrintText("092", Line4+6);
          _asm
          {
                  mov eax, DWORD PTR ss:[ebp+4*4]
                  mov addr, eax
          }
          io.AOAprintHex(addr, Line23+20);
          ptr=(char*)(addr-0x00401400+io.funaddr[1]);
          io.AOAPrintTextRasha(ptr, Line20+20);
          io.AOAPrintText("09D", Line5+6);
           _asm{sti}
           io.AOAstiTimer();
           io.AOAISRExit();  //special exit for ISR

};
```

**Figure 9: Int 92.**

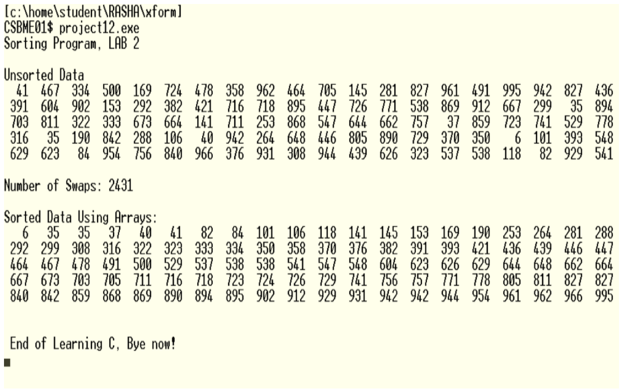
## 2.2 Sort Application (bSort)

The binary transformation process described for the simple print application can also be used to transform applications with a larger number of system calls. We consider the example of a standard bubble sort "bSort" program written in Visual Studio (VS). The executable has 6656 bytes or 13 sectors. The assembly listing of this program is 13 pages. The output of this executable running in a DOS window is shown in Fig. 11. As with the bPrintf binary executable, PEView, PEExplorer and Objdump are used to help with transforming the program to run on a bare PC. The entry point for this program is located at 0x14A0. Details of this example are omitted as it is similar to the bPrinf program. The bare PC output for bSort is shown in Fig. 12. In this application, we used many interrupts to replace system calls, where each system call is handled with one interrupt. The parameter passing strategy is same as in the bPrintf application. A tool that can automatically perform the binary transformation steps for simple applications such as print and sort will be developed in the future.



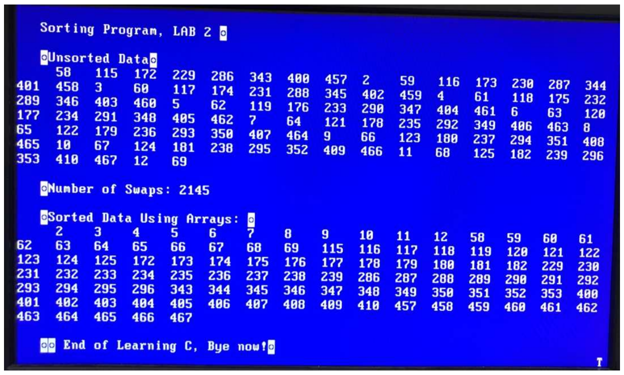
```
091
09D
                          I am here
                          00000204
092
09D



     00000015  00000002                                          000CDD86

     00000055  0000000F  00000065




                                                                 00000099
                                                       0000AE75  0000AE75
     S1  S2
                                                                 00000062


     12345678                                                           T
```

**Figure 10: Bare PC display for Printf.exe.**



```
[c:\home\student\RASHA\xform]
CSBME01$ project12.exe
Sorting Program, LAB 2

Unsorted Data
  41  467  334  500  169  724  478  358  962  464  705  145  281  827  961  491  995  942  827  436
 391  604  902  153  292  382  421  716  718  895  447  726  771  538  869  912  667  299   35  894
 703  811  322  333  673  664  141  711  253  868  547  644  662  757   37  859  723  741  529  778
 316   35  190  842  288  106   40  942  264  648  446  805  890  729  370  350    6  101  393  548
 629  623   84  954  756  840  966  376  931  308  944  439  626  323  537  538  118   82  929  541

Number of Swaps: 2431

Sorted Data Using Arrays:
   6   35   35   37   40   41   82   84  101  106  118  141  145  153  169  190  253  264  281  288
 292  299  308  316  322  323  333  334  350  358  370  376  382  391  393  421  436  439  446  447
 464  467  478  491  500  529  537  538  538  541  547  548  604  623  626  629  644  648  662  664
 667  673  703  705  711  716  718  723  724  726  729  741  756  757  771  778  805  811  827  827
 840  842  859  868  869  890  894  895  902  912  929  931  942  942  944  954  961  962  966  995


 End of Learning C, Bye now!
■
```

**Figure 11: bSort running in DOS window.**



```
    Sorting Program, LAB 2 ◘

    ◘Unsorted Data◘
      58   115  172  229  286  343  400  457  2    59   116  173  230  287  344
401  458  3    60   117  174  231  288  345  402  459  4    61   118  175  232
289  346  403  460  5    62   119  176  233  290  347  404  461  6    63   120
177  234  291  348  405  462  7    64   121  178  235  292  349  406  463  8
65   122  179  236  293  350  407  464  9    66   123  180  237  294  351  408
465  10   67   124  181  238  295  352  409  466  11   68   125  182  239  296
353  410  467  12   69

    ◘Number of Swaps: 2145

    ◘Sorted Data Using Arrays: ◘
      2    3    4    5    6    7    8    9    10   11   12   58   59   60   61
62   63   64   65   66   67   68   69   115  116  117  118  119  120  121  122
123  124  125  172  173  174  175  176  177  178  179  180  181  182  229  230
231  232  233  234  235  236  237  238  239  286  287  288  289  290  291  292
293  294  295  296  343  344  345  346  347  348  349  350  351  352  353  400
401  402  403  404  405  406  407  408  409  410  457  458  459  460  461  462
463  464  465  466  467

    ◘◘ End of Learning C, Bye now!◘
                                                                              T
```

**Figure 12: bSort running in bare PC window.**

## 2.3 Network Interface Device Driver (bNIC)

The bPrintf and bSort executables provided useful insights into the binary transformation process. However, each involves only a relatively small number of system calls and a small binary executable. We now consider the transformation of a more complex application: a bare PC gigabit Ethernet Intel NIC device driver, which has a large binary executable and a much larger number of external calls than the sort application.

Fig. 13 shows the system architecture used for this application. A bare PC Web server has an Ethernet NIC device driver that is closely integrated with the application program [7]. The Ethernet NIC device driver (object file EtherObj.obj) and the Web server code is a single monolithic executable. The Ethernet calls are made from various functions as indicated in the figure. The Ethernet calls are intertwined with the Web server code eliminating all layers in the network protocols. The above system is a working Web server with its Ethernet device driver.

In order to demonstrate the binary transformation of an Ethernet NIC device driver, a separate binary executable is created with EtherObj.obj and Ethernet.obj, where Ethernet.obj is a test file for this driver. The test file simply instantiates the EtherObj class and initializes the parameters. Many debug statements are inserted in this driver file to ensure the invocation of this driver after transformation. There is no other Web server code in the test file. A dummy Ethernet.exe is created to demonstrate the transformation process with an executable.

The challenge in this transformation is to cut off the Web server connections to original EtherObj.obj file and connect with the Ethernet.exe file. As shown in Fig. 13, connections to the Ethernet driver come from several methods including createOtherTasks() and StartTransfer(). In the Web server code, a connection can be a function call or a variable reference. In Fig.14, TDLPointer and SendInPtr are variables in the EO class, and FormatEthPacket() is a member function. Functions and variables are located in different segments in the binary executable and thus computing their addresses in memory is different.

A total of 27 instances in the test.exe (application program) are affected during the transformation process. In a conventional system, a sophisticated dynamic linker tool can resolve these references. As a similar tool has not been developed yet for BMC applications, a manual approach is currently used for this purpose.

*2.3.1 Function Calls.* Given an external function call in test.exe, it is replaced with the binary executable call as shown in Fig. 15. When a given function is called, it is necessary to identify the memory location of the call and the address of the called function in memory. For a ColdReset() function call in the program, Fig. 15 illustrates how to compute these addresses. Other function call addresses can be computed in a similar manner. In this application, test.exe is the executable to run the Web server.

The test.map file shows all the function addresses located in the test.exe file. The ColdReset() function is located within another function called a parent function. The parent function in this case is located at 0x3b2bf in the test.map file. The offset of the ColdReset() function within the parent function is 0x649, which can be obtained by looking at the parent function list file (e.g. File.lst). The ColdReset() function is located in the program at 0x3b908 (0x3b2bf+0x649), and in memory at 0x3ad08 (0x3b908 – 0x1000 – 0x400). Microsoft executables start at 0x1000 and they have 2 sectors of header files (0x400). When the ColdReset() function is called, the program counter is at (0x3b908 + 5=0x3b90d) as the current instruction is 5 bytes long. Currently, the hex code at the location of ColdReset() at address 0x3ad08 is 0xfffc6c61, which is pointing to the old ColdReset() call in the Ethernet NIC device driver. If we want to invoke the binary executable in the Ethernet.exe for this function, we need to change the address at this location.

The Ethernet.exe binary is loaded at a dummy address 0x11e1. The ColdReset() function in Ethenet.exe is located at 0x1361 (according to Ethernet.map file). So the actual address of ColdReset() function in the binary executable is 0x1361+0x11e1 = 0x2542. This implies the offset address for calling binary executable ColdReset() function is 0x2542-0x3b90D = 0xfffc6c35. Thus, the existing address at location 0x3ad08 has to be replaced with this offset address, shown in the figure in little endian form consistent with the Intel architecture. This illustrates the process for replacing a single call in a regular program with a call to the desired binary executable. The example shows that a binary executable can be transformed to work with a regular application. The transformation process is not the same as using DLLs in an OS environment, since BMC applications do not support DLLs and there is no centralized kernel or OS running in the system.

In this case, the binary transformation only involved BMC calls. The transformation methodology for OS to BMC calls will in principle be the same. However, the interactions between a conventional application (such as an OS-based NIC driver) and the OS will involve several additional layers resulting in a considerably more complex binary transformation.

*2.3.2 Variables.* Given a variable address, it is replaced with the binary executable address as shown in Fig. 16. An example of SendInPtr variable illustrates this process. The SendInPtr variable is located in the parent function at 0x4cfa2 address. The SendInPtr offset in the File.lst file is 0x87. So the SendInPtr is located in memory at 0x4c439 (0x4cfa2 + 0x87 – 0x1000 – 0x400). The SendInPtr in the Ethernet.map for the new location is at 0x00031028. Thus, we need to replace the data at 0x4c439 with the 0x00031028 value. This calculation is simpler than the function call as there is no need to compute the address offset and use the current program counter location.

The addresses generated for functions and variables are dependent on some parameters as shown in Figs. 15 and 16. Some of these parameters change when an application is modified and recompiled. In order to deal with this problem, we developed a spreadsheet to compute these addresses for all 27 functions with some parameters. It is also possible to put the spreadsheet equations in the code to dynamically compute these addresses and plug them into the binary executable at appropriate locations.

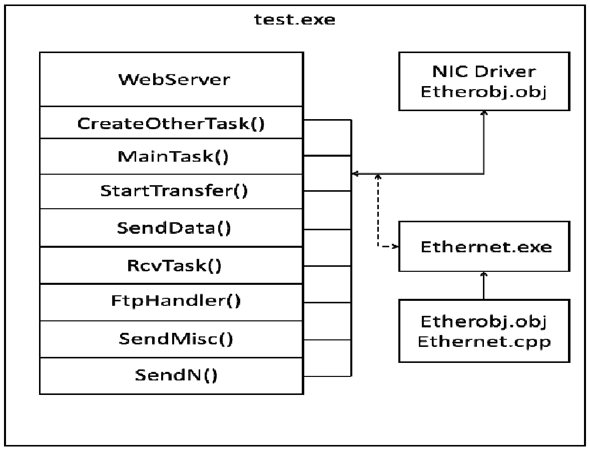


**Figure 13: Web server connections.**

```
x = EO.TDLPointer + EO.SendInPtr * 16 - ADDR_OFFSET;
    EO.SendInPtr++;
    retcode = EO.FormatEthPacket(send_buffer, TCPPack_size+20, IP_TYPE , TargetMAC,
    InPtr, sendtype, delegflag, delegatemac, currenttask);
```

**Figure 14: EO variables.**

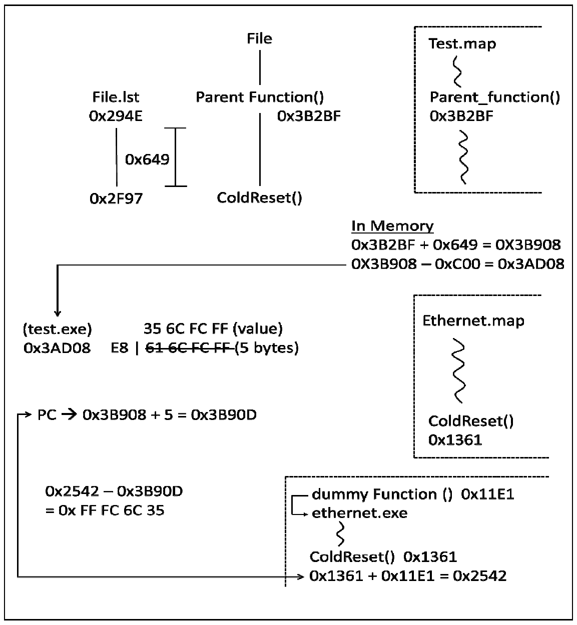


Fig 15

**Figure 15: Computing the ColdReset() address.**

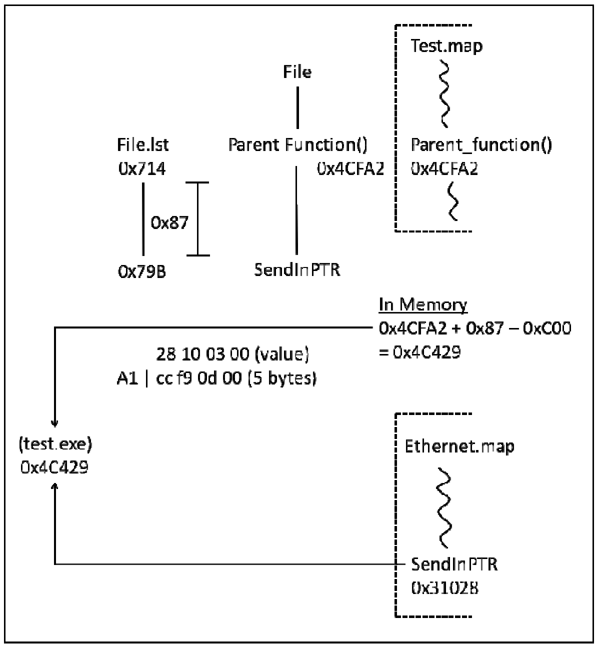
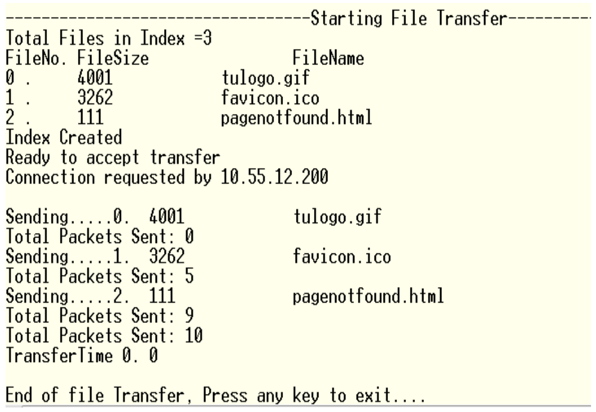


**Figure 16: Computing the SendInPtr address.**

## 2.4 Functional Operation after Transformation

After plugging in the necessary calls to the NIC driver for all external calls, we tested the Web server for proper operation. Initially, as shown in the screenshot of Fig. 17, the Web server acts as a client and transfers files from a Windows server. Once the file transfer is complete, the Web server is ready to serve client requests. The "tulogo.gif" page is shown in Fig. 18 as an example to illustrate successful completion by the server of a client request.

This approach demonstrates that binary code can be dynamically linked and transformed to work with an existing BMC application such as a Web server. The technique used here serves as a first step towards developing a general methodology to perform binary transformations of existing OS based executables.



```
---------------------------------Starting File Transfer---------
Total Files in Index =3
FileNo. FileSize              FileName
0 .     4001           tulogo.gif
1 .     3262           favicon.ico
2 .     111            pagenotfound.html
Index Created
Ready to accept transfer
Connection requested by 10.55.12.200

Sending.....0.  4001            tulogo.gif
Total Packets Sent: 0
Sending.....1.  3262            favicon.ico
Total Packets Sent: 5
Sending.....2.  111             pagenotfound.html
Total Packets Sent: 9
Total Packets Sent: 10
TransferTime 0. 0

End of file Transfer, Press any key to exit....
```

**Figure 17: Bare PC display for file transfer.**

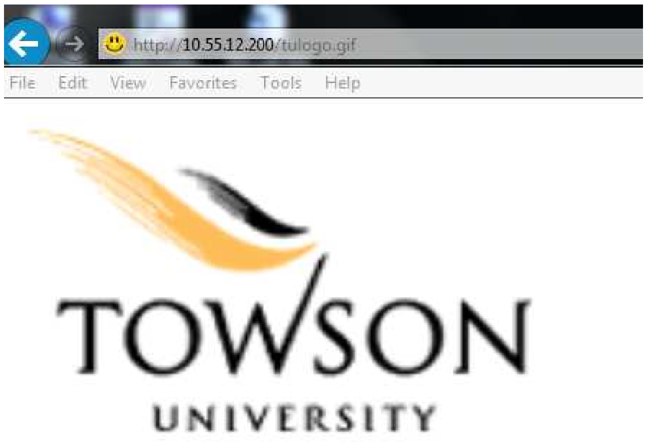

http://10.55.12.200/tulogo.gif
File Edit View Favorites Tools Help
TOWSON
UNIVERSITY

**Figure 18: Browser display for a client request.**

## 3 DISCUSSION

The transformation process in these three examples highlights several issues that require further research. For example, there is a need for user driven semi-automatic or fully automated transformation tools, which can do the transformation process dynamically at run time. Also, the host BMC application should provide a running BMC environment for foreign binary executables. Understanding external system calls and replacing them with the appropriate BMC calls is a difficult problem. Finding equivalent BMC functions and sometimes replacing them with dummy functions is currently done using an ad-hoc approach. More formal methods for replacing external system calls will need to be investigated in future. Additional problems that were discovered during our transformation experiments with OS-based binary executables relate to module intertwining. Finally, the transformation process should not require an understanding of the binary code, as otherwise the problem becomes unmanageable. It should be noted that since BMC systems are different from conventional systems, it is not possible to directly apply standard binary reverse engineering techniques used to map between different OS environments (for example, techniques enabling Windows binaries to run in Linux).

The binary transformation process presented here lays a foundation for future research that will enable current OS based applications to run on bare machines or bare PCs with no OS support. This makes the transformed applications independent of computing platforms such as Windows and Linux. As an OS or kernel serves as middleware, eliminating it will result in a flat model where general-purpose applications directly communicate to the hardware. The transformation concept can also be applied to pervasive devices in a variety of operating environments. Once the transformation process is automated, BMC interfaces will provide common computer interfaces to the hardware reducing the current heterogeneity among hardware and architectures.

## 4 RELATED WORK

The BMC paradigm is related to research efforts attempting to eliminate the OS or reduce its footprint [11-15]. The essential difference between these approaches and the BMC approach is that the latter enables general-purpose applications to run directly on the bare PC hardware without any OS or kernel support. Unlike porting an application from one OS platform to another, transforming the source code of OS-based applications to run on a bare PC has proved to be difficult [6]. This is the first attempt to transform binary executables from conventional OS-based applications to run on a bare PC.

Similar work in an OS-based environment is essentially reverse engineering. For example, due to the difficulty of writing numerous network device drivers for Linux from scratch, the NDIS wrapper [16] was developed. This wrapper has been used to enable legacy Windows drivers to run on Linux by implementing the necessary parts of the Windows kernel and linking the driver. RevNIC [17] adopts a different approach to reverse engineering a binary NIC driver. It consists of a tool that takes the driver binary, uses reverse engineering techniques to derive the driver logic, and produces device driver code that matches the original driver interactions with the hardware. RevNIC has been used to reverse engineer a few Windows drivers to run on different OSs. While the approach is innovative, there do not appear to be examples of using RevNIC to reverse engineer drivers other than those discussed in the work by the original authors. Finally, an example showing how to reverse engineer a driver for a USB device is given in [18]. As discussed in the previous section, since BMC applications run on a bare PC with no OS or kernel support, binary transformations of OS executables is different from conventional reverse engineering.

## 5 CONCLUSION

We investigated binary transformations that enable an existing binary executable for a conventional OS-based application to be run as or within a BMC application on a bare PC. Transformed binaries include a printing application, a sorting application, and a binary network interface driver. We described the steps in the transformation process for each binary and provided the relevant internal implementation details. We also discussed issues to be addressed in future research on binary transformations and the significance of this work. While the current work is preliminary, it will serve as foundation for future research targeting the binary transformation of complex OS-based applications. We are presently investigating the feasibility of applying the transformation techniques used for the BMC NIC binary to OS-based binary NIC drivers.

## REFERENCES

[1] L. He, R. K. Karne, and A. L. Wijesinha, "The design and performance of a bare PC Web server," International Journal of Computers and Their Applications, IJCA, Vol. 15, No. 2, June 2008, pp. 100-112.

[2] H. Chang, R. K. Karne, and A. Wijesinha, "Migrating a Bare PC Web Server to a Multi-core Architecture," 40th Annual IEEE International Computer Software and Applications Conference (COMPSAC), 2016, pp. 216-221.

[3] B. Rawal, R. K. Karne, and A. L. Wijesinha. "Mini Web server clusters for HTTP request splitting," IEEE Conference on High Performance, Computing

and Communications (HPCC), 2011, pp. 94-100.

[4] P. Appiah-Kubi, R.K. Karne and A.L. Wijesinha, “A Bare PC TLS webmail Server,” International Conference on Computing, Networking and Communications, (ICNC), 2012, pp. 156-160.

[5] W. Thompson, R. Karne, A. Wijesinha, and H. Chang, “Interoperable SQLite for a bare PC,“ Beyond Databases, Architectures and Structures Conference (BDAS), 2017, pp. 177-188.

[6] U. Okafor, R. K. Karne, A. L. Wijesinha and B. Rawal, "Transforming SQLite to Run on a Bare PC," 7th International Conference on Software Paradigm Trends (ICSOFT), 2012, pp. 311-314.

[7] F. Almansour, R. K. Karne, A. L. Wijesinha, R. Karne, and B. S. Rawal, “Ethernet bonding on a bare PC web server with dual NICs,” 33rd Annual ACM Symposium on Applied Computing (SAC), 2018, pp. 1116-1121.

[8] WJRSofware-PEView, http://wjradburn.com/software/, [Accessed 9-24-18].

[9] ObjDump, https://sourceforge.net/projects/objdump/, [Accessed 9-24-18].

[10] PE Explorer, http://www.pe-explorer.com, [Accessed 9-24-18].

[11] D. R. Engler and M.F. Kaashoek, “Exterminate all operating system abstractions,” Fifth Workshop on Hot Topics in Operating Systems, USENIX, 1995, p. 78.

[12] J. Lange, et. al, “Palacios and Kitten: new high performance operating systems for scalable virtualized and native supercomputing,” 24th IEEE International Parallel and Distributed Processing Symposium, 2010.

[13] GitHub – ReturnInfinity/BareMetal-OS, https://github.com/ReturnInfinity/BareMetal-OS, [Accessed 9-24-18].

[14] Linux Kernel Tinification, https://tiny.wiki.kernel.org/, [Accessed 9-24-18].

[15] A Minimal Rust Kernel, https://os.phil-opp.com/minimal-rust-kernel/, [Accessed 9-24-18].

[16] NDISWrapper, http://ndiswrapper.sourceforge.net, [Accessed 9-24-18].

[17] V. Chipounov and G. Candea “Reverse engineering of binary device drivers with RevNIC,” 5th ACM European Conference on Computer Systems (EUROSYS), 2010.

[18] B. Everard, “Drive it yourself: USB Car,” LINUXVOICE, March 2015. https://www.linuxvoice.com/drive-it-yourself-usb-car-6/, [Accessed 9-24-18].