EPiC Series in Computing

Volume 69, 2020, Pages 151–160

Proceedings of 35th International Conference on Computers and Their Applications

Transformation Methodology of Binary Executables to Run on Bare Machines

N. Ordouie[1], R. Almajed[2], R. Karne[1], A. Wijesinha[1], J. Weymouth[1], and N. Soundararajan[1]

[1] Towson University, Towson, USA

[2] American University in the Emirates, Dubai, UAE

nordouie@towson.edu, rasha.almajed@aue.ae, {rkarne, awijesinha, jweymouth, nsoundararajan}@towson.edu

Abstract

This paper identifies challenges involved in the transformation of binary executables to run on bare machines such as PCs. It also addresses why we want to transform binary executables to run on bare machines. Text processing applications such as "vi," "word," and "notepad" are chosen to illustrate the need for transformation because these editors are the most commonly used across many operating system platforms, including Windows and Linux. They have much functionality in common to provide a general text processing application. Why not consolidate these standard functions and develop a generic text processing application? How do you make these editors without any platform dependencies? Transforming these applications to run on bare PCs or bare machines by using source or a binary level transformation will address these challenges. A binary transformation methodology described here lays the groundwork for further research in this area and provides some insight into the transformation process.

1. Introduction

As the operating systems (OS) have grown in size and complexity over the years, researchers have turned their attention to move some of the OS functionalities to applications and reduce the size of the OS, resulting in a lean operating system. The Exokernel [1], Tiny-OS [2], IO_Lite [3], Palacio and Kitten [4], Bare Metal [5], OS-Kit [6], Raw Architecture [7] are some examples of minimizing the size of complexity of OS or kernel. This concept is extended further to the bare machine computing (BMC) paradigm, which requires no OS or kernel system [8, 9]. A bare computing box means there is no OS or kernel and no persistent storage (hard disk). This computing box has no value, ownership, or firewalls. Any user can use any computing device without a username or password to the device. The BMC approach also requires a BMC programming paradigm, where application programs directly communicate to hardware without any assistance from any middleware. This approach makes the programming task difficult by making applications to deal with systems programming. However, it

offers immense benefits by avoiding open systems and layered models that result in heterogeneity, complexity, and rapid obsolescence. The BMC paradigm provides an alternate solution to these problems and offers a more straightforward and inherently secure architecture for selective application domains. The BMC paradigm demonstrates these attributes in many applications [10-16]. The BMC approach is a non-evolutionary paradigm, which needs mapping or transformation of existing applications to run on bare machines. The source-code level transformation is demonstrated for SQLite [17] to run on a bare PC. The binary code transformation process was initially attempted for some simple applications to run on bare machines [18]. There is abundant work done at the binary level to optimize compilers, reverse engineering, and instruction-level insight [19-23]. There are many tools [24-28] available for binary transformation and analysis. This paper takes the work already done [18] to the next level and provides a standard methodology to handle the transformation process and address design issues.

2 Background

An application is mostly written in a high-level language, compiled, linked, and loaded to run on a computer. The compiling process generates an executable that can run on an underlying architecture. However, this executable cannot run without assistance from an OS. A given executable also has a dependency on an underlying CPU architecture. A common assumption is that many other applications are also running on the computer at the same time in a multiprocessing environment. The executable is a passive entity until the OS provides process facilities to load and run on the machine. During the execution of this process, the OS provides all hardware services as needed, and schedules the execution whenever possible. The OS also controls exceptions, interrupts, and inter-process communications.

An executable makes internal and external calls. The external calls are OS-dependent, since they lead to system calls, either by directly calling the appropriate APIs, or indirectly through a dynamically linked library. The executable requires the replacement of external calls with the equivalent calls in BMC to communicate directly with the hardware and run on a bare machine. There is also a need to load the application program before it starts running on a machine.

An application is specific to a given domain, vendor, user needs, platform, time of releases, and software and hardware capabilities. These parameters generate heterogeneity among applications. The heterogeneity spurs across many versions and products, and does not follow any object-oriented principles such as extensibility, upward compatibility, and reuse. Consequently, in today's world, we make obsolete products that result in a dumping and waste problem [29]. The root cause of this problem is the practice of designing environment-dependent applications; when an environment changes, the product changes. To slow down the obsolescence of applications, and reduce their heterogeneity, we must design them without any dependency on environments. For a simple example of an application without any dependency on the environment, consider the following. A text processing application is a generic application that has been there for a long time, and it will be there forever. We design, and use Microsoft Word, Notepad, and "vi" for text processing, but they are, by definition, environment-sensitive. In addition, they have many common features such as user interface, file manipulation interface, editing and parsing commands, memory needs, as well as some unique features such as spell checking, word search, and formatting. Some have unique features of their own. However, one application cannot use another's unique features. We propose some novel ideas that allow these disparate applications to be amalgamated at run time. Then, a text-processing application can become independent of any computing environment. If we build all applications without any dependency on environments, we will have efficiency in computing and applications.

3. Text Processing Illustration

This section presents some studies made on three common applications "vi," "Microsoft Word," and "Notepad" enabling them to be compared as text processing tools.

3.1 "Vi"

The "vi" editor chosen here is a Vim version 7.2 on Windows 10. The size of the executable is 1,421,312 bytes. When this application runs, it requires 239 external calls that are OS-dependent. Table 1 illustrates the system characteristics of this application. Of 4172 internal calls that perform some specific functions in the editor, the 239 external calls are exported by seven DLLs. In some cases, the binary needs only a few functions from a DLL. While this application is running, it may have to load all seven DLLs at some point during the execution.

The seven DLLs, in turn, are dependent on other DLLs, as shown in Table 1. These DLLs depend on 357 other DLLs. Some of the DLL dependency trees are multilevel and very complex. Moreover, the size of the code for DLLs also increases rapidly as the number of DLLs increases. The DLL approach addresses the problems of modularity, flexibility, and open systems, but fails to deal with the complexity of loading DLLs, resolving the linking problem at run time, and security vulnerabilities. Table 2 shows the type of system calls and the number of calls in each category for this application.

Table 1: Vi Editor System Characteristics – part 1

Library	Number of Exported Calls Used	Code Size (Bytes)	Number of DLL Interface Dependencies	Total Functions Exported
ADVAPI32.dll	1	645,320	33	~1802
KERNEL32.dll	162	720,200	91	~1362
SHELL32.dll	2	21,399,576	81	~332
GDI32.dll	19	155,112	12	~726
comdlg32.dll	2	935,424	42	~29
olde32.dll	3	1,376,688	~60	~400
USER32.dll	50	1,639,560	38	~849
External Calls	**239**			
Internal Calls	**4172**			
Total	**4411**	26,871,880	357	

3.2 "Microsoft Word"

Table 3 shows nine types of DLLs for running Microsoft word. At run time, it uses 55 external calls. Table 4 shows the breakdown of these 55 external calls, and illustrates the type of functions needed at runtime.

3.3 "Notepad"

This application has 226 external calls and 28 types of DLLs that export these calls. Table 5 shows the types of system calls.

Table 2: Vi Editor System Characteristics – part 2

Type of Calls	Number
Console	27
Data-Exchange	8
Diagnostics	4
Dynamic-Link-Library	5
Exception-Handling	2
Execution	24
File	34
Keyboard-and-Mouse	4
Memory	20
Miscellaneous	73
Resource	1
Storage	6
Synchronization	8
System-Info	8
Windowing	15

Table 4: Microsoft Word System Characteristics – part 2

Type of Calls	Number
Diagnostics	2
Dynamic-Link-Library	4
Exception-Handling	2
Execution	5
File I/O	1
Memory	4
Misc.	28
Registry	3
Synchronization	1
System-Info	5

Table 3: Microsoft Word System Characteristics – part 1

Library	Number of Exported Calls Used	Code Size (Bytes)	Number of DLL Interface Dependencies	Total Functions Exported
ADVAPI32.dll	3	645,320	33	~1802
KERNEL32.dll	23	720,200	91	~1362
VCRUNTIME	4	366,128	6	71
api-ms-win-crt-heap-l1-1-0	1	19,264		27
api-ms-win-crt-locale-l1-1-0	2	18,752		20
api-ms-win-crt-math-l1-1-0	1	28,992		320
api-ms-win-crt-runtime-l1-1-0	17	22,840		107
api-ms-win-crt-stdio-l1-1-0	3	24,384		159
api-ms-win-crt-string-l1-1-0	1	24,384		178
External Calls	**55**			
Internal Functions	**28**			
Total	**83**	**1,870,264**	**130**	

Table 5: Notepad System Characteristics

Library	Number of Exported Calls Used	Code Size (Bytes)
ADVAPI32.dll	14	645,320
KERNEL32.dll	66	720,200
api-ms-win-core-com-l1-1-0.dll	9	15,668
... Some rows removed....		
GDI32.dll	22	155,112
USER32.dll	74	1,639,560
Msvcrt.dll	33	636,936
COMCTL32.dll	2	664,984
COMDLG32.dll	11	935,424
Ntdll.dll	1	1,943,136
PROPSYS.dll	2	1,793,472
SHELL32.dll	7	21,399,576
SHLWAPI.dll	5	326,496
WINSPOOL.DRV	3	522,240
Urlmon.dll	1	1,808,896
External Calls	**226**	
Internal Functions	**9**	
Total	**235**	33,415,516

Call Type	Count
Cryptography	3
Data-Exchange	3
Dynamic-Link-Library	7
Desktop	1
Diagnostics	9
Exception-Handling	3
Execution	10
File	17
Hooking	2
Keyboard-and-Mouse	7
Memory	19
Network	1
Registry	5
Resource	3
Security	2
Shell	1
Synchronization	9
System-Info	4
Windowing	21
Misc.	144

A variety of observations made from the study of the three text processing tools is as follows:

- At run time, each application requires a large number of DLLs
- At run time, the code size can increase dramatically to load additional DLLs
- DLLs can have multiple levels of dependency on other DLLs. This causes further aggravation in DLLs and their required storage
- Loading and linking DLLs at runtime can slow down the operation
- Some DLL links can be broken and compromised
- DLL code must be trusted; otherwise, they could introduce security vulnerabilities into the application and the entire system
- Managing DLLs creates complexity during run-time
- Reducing the code size in application by using DLLs results in complexity at run time
- There is no inter-DLL sharing possible because of independent compilation
- Each application has some unique features that other applications may not have.

The above observations leads to some fascinating insights. If we use the BMC paradigm, we only need to provide external calls, as shown in the above examples. These external calls amount to 520 for all three applications. For “vi”, only half of the external calls were necessary for running on a bare PC. If we assume 40% commonality among the external calls of these applications, then we only need to provide 312 calls. These 312 calls are direct hardware interfaces and do not have any other level of dependency. By providing direct hardware interfaces for an application at run time, instead of managing thousands of DLLs and large amounts of dynamic storage areas, we make the application run time simpler with less code. Such an application also uses less memory and becomes inherently more secure.

4 Transformation Methodology

The transformation of binary executables to run on bare machines poses some significant challenges. We address these challenges by developing a three-step process

During the first step, as shown in Fig.1, a bare PC application is set up to host a blob of binary executable. The binary executable blob is loaded in a bare PC application address space so that it can utilize direct hardware interfaces (HAPI) available in the bare PC application, and avoid dynamic linking at runtime. A separate task in the bare PC application runs the binary executable. When the binary executable is finished running, it returns to the bare PC application.

The second part of the methodology involves preparing the binary executable to run on a bare machine. Fig. 1 provides a brief overview of the methodology for the binary transformation process. A more detailed description is not relevant for this paper.

There are many other design issues in preparing a binary executable to run on a bare machine. Current OS and compiler related tools cannot be used at run time in BMC, as they are OS-dependent. Some of the design issues are briefly described as follows:

1. **Finding the right system call:** There are hundreds of system calls in today's OS environment, as described in the above examples. Finding the right system call is a significant problem in the transformation process. There are four possible choices in finding a system call.
    a. There is an equivalent system call in BMC
    b. There is a similar system call in BMC but requires some modification
    c. It requires a new system call in BMC
    d. The system call does not apply to BMC.
2. **Stack Problems:** For each system call, the stack parameters and layout are different. Parameters passed through the stack require stack pointer adjustment.
3. **Program control returning to the correct place after the call:** The stack stores the return address. As the number of parameters and their references is unknown until the run time, the current stack pointers may change due to parameter retrieval in the call. Recovering the correct return address and returning to the correct place is crucial for proper operation.
4. **Not applicable calls**: When an external call does not apply to BMC, it can use a dummy call and return. Sometimes, the dummy call may have to pass valid parameter values, so that after returning, it resumes the correct code flow. In order to do this, we may have to add an artificial JMP instruction to skip the code, requiring editing the code dynamically.
5. **Miscellaneous issues**: During a return from an external call, it is possible to modify the binary code if a code flow graph is available at run time. Using IDA Pro [30], we can create a code flow graph, store it in a file, and use it at the call time to make appropriate changes in the binary code. This modification of binary code is a revolutionary idea that will enable the modification of the flow control at run time.

The third part of the methodology is to build a tool that automatically prepares a binary executable to run on a bare machine. That endeavor is not part of this paper.

The "**vi**" editor has 239 external calls. Fig. 2 illustrated a DLL function call replacement. For each external call, there is a location address and its return address in memory. The return address is simply the call address plus the size of the call instruction. The "retn" parameter in the call should have an

appropriate return value to find the return address in the stack. An example to demonstrate the external call "GetVersionExA()" is shown in the figure. A view of the stack memory after replacing the external call with the BMC equivalent call is shown in Fig. 3. At run time, the transformation tool must compute a return address for each call and set up the "retn" value dynamically in the binary code.

A preliminary analysis of the "vi" binary executable shows that about half of the external calls are unnecessary for running on a bare PC. Some of these calls are not relevant for BMC execution.

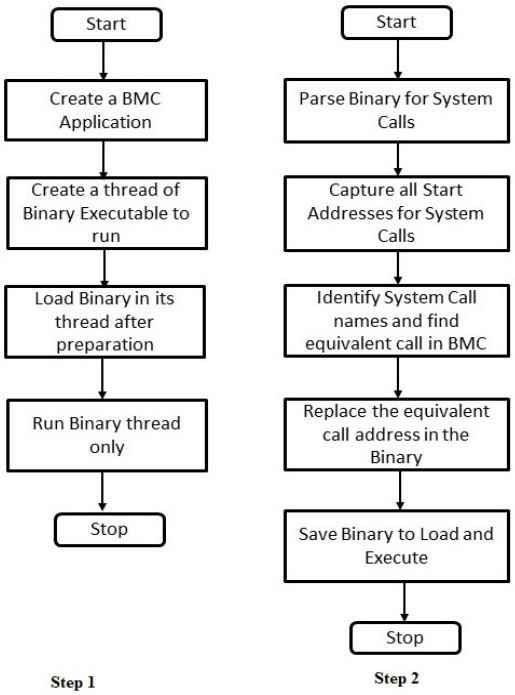

Figure 1: Methodology

5 Justification for Transformation

The BMC paradigm proposed a revolutionary and non-evolutionary approach to develop a new set of applications that are simple, stable, application-centric, and inherently more secure. To exploit the BMC paradigm, many existing applications can run on bare machines. Source code transformation [17] has challenges since the open-source code is unavailable for many applications. In a typical application, intertwined header files make the transformation process harder. It is not easy to untangle the dependencies and generate an amalgamated file for a given application. An alternate approach is to explore binary level transformation, which avoids the problems faced in source code transformation. There is no need to understand the binary code functionality if the system calls are replaced correctly and adequately. When an OS-based binary executable is transformed to run on bare machines or bare PCs, it has immense benefits and broader impacts in computing. Some of these benefits are listed below.

- The application becomes independent of the OS or kernel.
- Long term archiving is a possible use of the transformed binary executable [30].

- Applications become simple.
- Applications avoid interfacing with too many DLLs.
- It reduces code size and dynamic links at run-time.
- It reduces vulnerabilities due to other unnecessary DLLs and links.
- It allows the abstraction of applications (e.g., text processing).
- It allows sharing code at the binary level from one domain to another (standard functions can be shared).

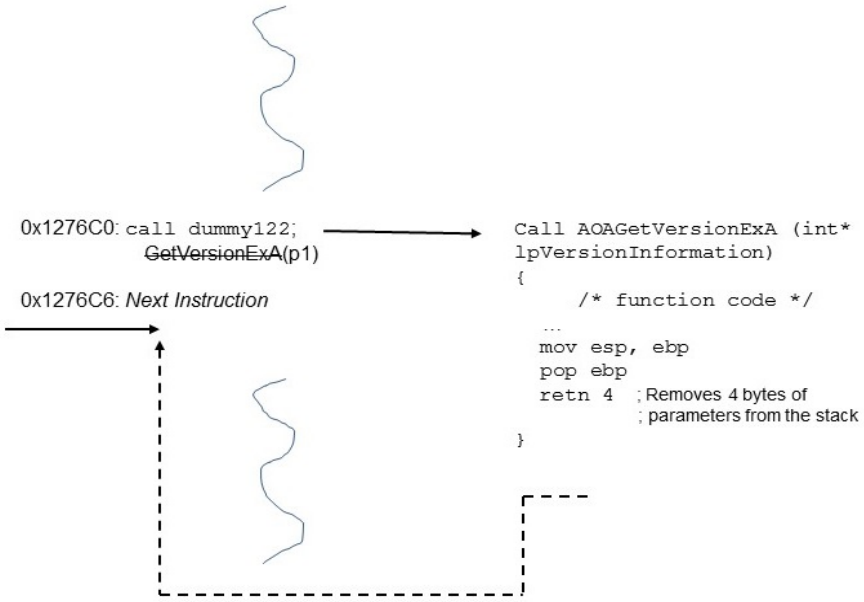

Figure 2: External Call Flow Control

6 Conclusion

This paper shows the complexity of a binary executable at runtime as it interfaces with dynamically linked modules and illustrates how a complex run-time environment will not hinder simplifying the application. It also shows a novel way to abstract environment-sensitive applications and create an abstract level application that is independent of environments, thus inheriting stability and avoiding obsolescence. Furthermore, it describes a generic methodology for transformation that can be used to construct an automated tool to transform binary executables. The concepts presented here open doors to share binary code with diverse applications and domains at runtime. More research is needed to construct automated tools that can perform the binary transformation of existing executables for running on bare machines. Long term archival will be feasible when binary transformation becomes practical. This work serves as a foundation for future binary transformations to run applications on bare machines.

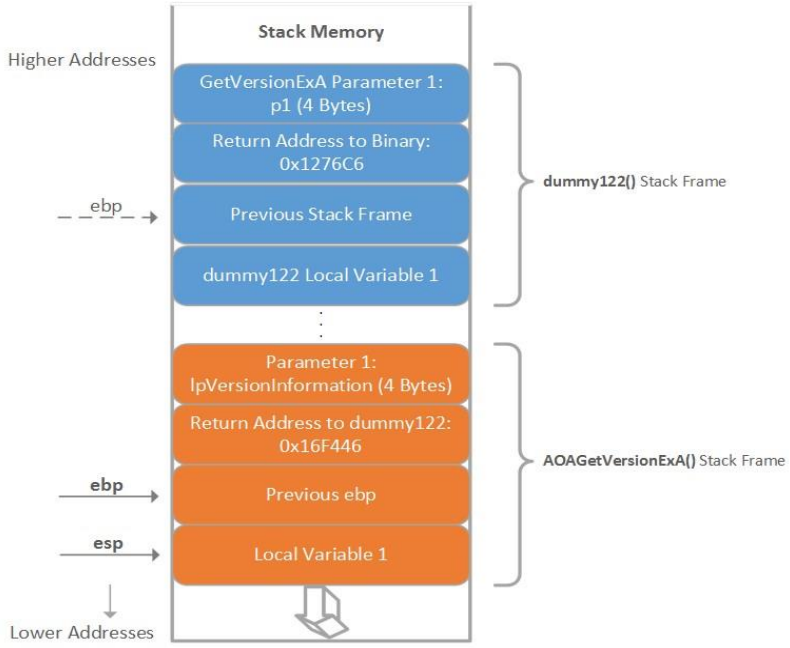

Figure 3: Stack Memory after External Call Replacement

References

[1] D. R. Engler. The Exokernel Operating System Architecture, Ph.D. thesis, MIT, October 1998.

[2] TinyOS is an open-source operating system designed for wireless embedded sensor networks http://www.tinyos.net/.

[3] V. S. Pai, P. Druschel, and W. Zwaenepoel. W. IO-Lite: A Unified I/O Buffering and Caching System. In Proc. ACM Transactions on Computer Systems, Volume 18, Issue 1, pp. 37 – 66. February 2000.

[4] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, R. Brightwell, Palacios, and Kitten: New High-Performance Operating Systems for Scalable Virtualized and Native Supercomputing, Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS), April 2010.

[5] T. Venton, M. Miller, R. Kalla, and A. Blanchard. "A Linux-based tool for hardware bring-up, Linux development, and manufacturing." IBM Systems Journal, Vol. 44, No. 2, pp. 319-329, 2005.

[6] http://www.cs.utah.edu/flux/oskit. The OS Kit Project.

[7] Raw Architecture Workstation, http://groups.csail.mit.edu/cag/raw/purpose/.

[8] https://en.wikipedia.org/wiki/Bare_machine_computing.

[9] https://orion.towson.edu/~karne.

[10] G. H. Ford, R. K. Karne, A. L. Wijesinha, and A. Patrick. The Performance of Bare Machine Email Server, 21st International Symposium on Computer Architecture and High-Performance Computing, SBAC-PAD 2009, October 28-31, Sao Paulo, Brazil, p143-p150.

[11] 6. P. Appiah-Kubi, A. L. Wijesinha and R. Karne. The Design and Performance of a Bare PC Webmail Server, The 12th IEEE International Conference on High-Performance Computing and Communications, AHPCC 2010, Sept 1-3, 2010, Melbourne, Australia, p521-526.

[12] B. Rawal, R. Karne, and A. L. Wijesinha, "Mini Web Server Clusters based on HTTP Request Splitting" HPCC 2011: The 13th IEEE International Conference on High-Performance Computing and Communications, Sep 2, 2011 - Sep 4, 2011, Banff, Canada.

[13] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, L. He, and S. Girumala. A Peer-to-Peer Bare PC VoIP Application", IEEE Consumer and Communications and Networking Conference (CCNC), Las Vegas, Nevada, January 2007.

[14] 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.

[15] H.Alabsi, R. K. Karne, A. Wijesinha, R. Almajed, B. Rawal, and F. Almansour, A Novel SQLite-Based Bare PC Email Server, 15nth International Conference, BDAS2019, Ustron, Poland, May 28-31, 2019, p341-p353.

[16] S.Almutairi, R. Karne, A. Wijesinha, H. Chang, R. Almajed, H. Alabsi, W. Thompson, and N. Soundararajan An API for Bare Machine Computing Applications, IEEE SoutheastCon, April 2019.

[17] U. Okafor, R. K. Karne, A. L. Wijesinha and B. Rawal, "Transforming SQLITE to Run on a Bare PC," In Proceedings of the 7th International Conference on Software Paradigm Trends, pages 311-314, Rome, Italy, July 2012.

[18] R.Almajed, R. K. Karne and A.L. Wijesinha, Binary Transformation of Applications to run on Bare PCs, The 34th ACM/SIGAPP Symposium On Applied Computing, April 8-12, 2019, Limassol, Cyprus.

[19] Srinivasan, V., Vartanian, A., and Reps, T., Model-assisted machine-code synthesis. In *PACMPL 1(OOPSLA)*: 61:1-61:26 (2017). [abstract; PDF; slides]

[20] Prateek Saxena, "Static Binary Analysis And Transformation For Sandboxing Untrusted Plugins," M.S Thesis, SUNY Stony Brook, 2007.

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

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

[23] Gogul Balakrishnan and Thomas Reps, "Analyzing Memory Accesses in x86 Executables", Comp. Sci. Dept., University of Wisconsin; {bgogul,reps}@cs.wisc.edu.

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

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

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

[27] https://reverseengineering.stackexchange.com/questions/19179/older-versions-of-ida-free

[28] IDA Pro Disassembler, http://www.datarescue.com/idabase/, [Accessed on 10-4-2019].

[29] http://news.bbc.co.uk/2/hi/technology/5107642.stm. 30 million PCs being dumped each year in the US alone." BBC News: PC users.

[30] M. Peterson. SNIA (Storage Networking Industry Association), "The Coming Archive Crisis," November 2006.