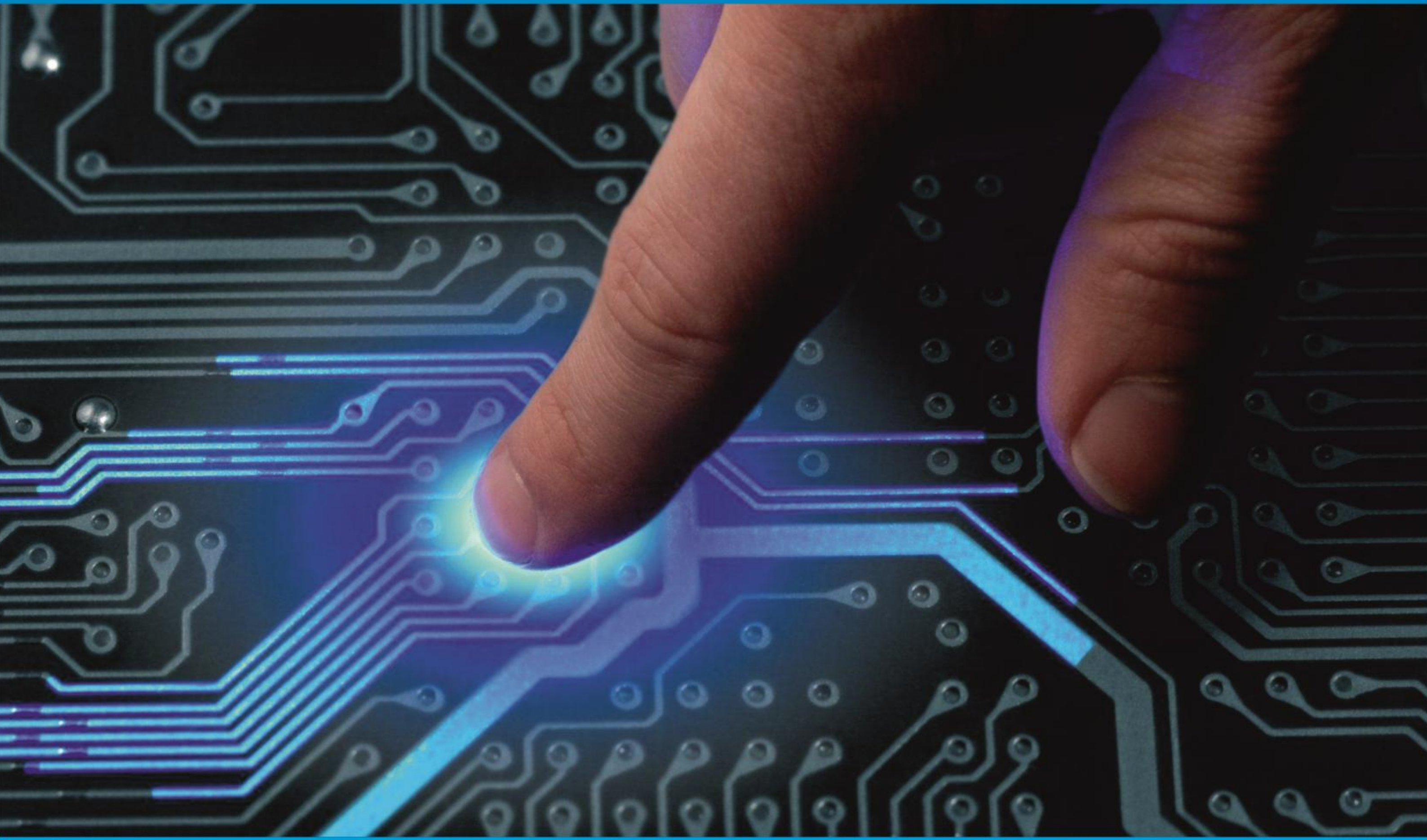
IJIRCCE

e-ISSN: 2320-9801 | p-ISSN: 2320-9798



INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN COMPUTER & COMMUNICATION ENGINEERING

Volume 9, Issue 5, May 2021

INTERNATIONAL STANDARD SERIAL NUMBER INDIA

Impact Factor: 7.488

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.488 |

|| Volume 9, Issue 5, May 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0905032 |

A Novel Survey on Reverse Engineering Techniques for Improved System Security

Nagesh[1], Saurabh[2], Prof. Rajeshwari Gundla[3]

Student, School of Engineering, Ajeenkya D Y Patil University, Pune, Maharashtra, India [1]

Student, School of Engineering, Ajeenkya D Y Patil University, Pune, Maharashtra, India [2]

Assistant Professor, School of Engineering, Ajeenkya D Y Patil University, Pune, Maharashtra, India [3]

ABSTRACT: Reverse Engineering is a point on the challenging task of understanding heritage program code without having suitable documentation. Using a transformational forward engineering view, the difficulty is caused by design resolution made during system development. Such decisions “hide” the program functionality and performance requirements in the final system by putting in repeated clarification through layers of generalization, and information-spreading optimizations, both of which change representations and force single program entities to serve multiple purposes. Following the transformational procedure, we can use the transformations of a forward engineering methodology and apply them “backwards” to reverse engineer code to a more abstract identification. This paper presents reverse engineering program understanding theories and the reverse engineering technology. The demand by all business sectors to modify their information systems to the web has created a tremendous need for methods, tools, and infrastructures to evolve and make use of existing applications efficiently and cost-effectively. Reverse engineering has become the most promising technology to fight this legacy systems problem. The difference between reverse engineering and standard scientific research is that with reverse engineering the antiquity being investigated is man-made, dissimilar to scientific research where it is a natural phenomenon. The concept has been around because long before computers or modern technology, and likely dates back to the days of the industrial insurrection. It is very similar to scientific research, in which a researcher is endeavoring to work out the “blueprint” of the atom or the human mind.

KEYWORDS: Reverse Engineering, Code Reverse Engineering, Data Reverse Engineering

I. INTRODUCTION

Engineering practice is inclined to focus on the design and implementation of a product without considering its lifetime. The concept of computers automatically finding useful information is an exciting and promising feature of just about any application intended to be of practical use [2]. Although, the major effort in software engineering organizations is spent after development on maintaining the systems to remove existing errors and to modify them to change requirements. Unfortunately, mature systems frequently have incomplete, incorrect or even nonexistent design documentation. This makes it difficult to understand what the system is doing, why it is doing it, why the work is performed, and why it is coded that way [5]. As a consequence, mature systems are hard to modify and the tempering is difficult to validate. Chikofsky and Cross defined reverse engineering to be analyzing a subject system to identify its current components and their province, and to extract and create system generality and design information. Current reverse engineering technology centers attention on regaining information by using analysis tools and by extracting programs bottom-up by recognizing plans in the source code [8]. Reverse engineering is the procedure of taking out the knowledge or design blueprints from anything man-made [20]. Reverse engineering is normally conducted to obtain missing knowledge, ideas, and design philosophy, such information is unavailable. In some cases, the information is owned by someone who isn't prepared to share them. In other cases, the information has been lost or demolished [1]. Traditionally, reverse engineering has been about taking shrink-wrapped products and physically anatomizing them to uncover the secrets of their design. Some secrets were then typically used to make similar or better products. In some industries, reverse engineering involves inspecting the product under a microscope or taking it apart and figuring out what each section does [23]. Not too long ago, reverse engineering was actually a justly popular hobby, expert by a large number of people even if it wasn't referred to as reverse engineering. Remember how in the early days of modern electronics, many people were so amazed by modern appliances like radio and television that it became common practice to take them apart and see what goes on inside? That was reverse engineering [29]. Of course, advances in the

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.488 |

|| Volume 9, Issue 5, May 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0905032 |

electronics industry have made this practice far less applicable. Modern digital electronics are so miniaturized that nowadays you really would not be able to see much of the fascinating stuff by just opening the box.

II. LITERATURE SURVEY

1. Software Reverse Engineering: Reversing

Software is one of the most complex and intriguing technologies around us this day, and software reverse engineering is about opening up a program's “box,” and looking inside [27]. Of course, we will not need any screwdrivers on this journey. Just like software engineering, software reverse engineering is an only virtual operation, involving only a CPU, and the human mind. Software reverse engineering essentially requires a combination of skills and a thorough understanding of computers and software development, but like most valuable subjects, the only real prerequisite is a strong nosiness and desire to learn. Software reverse engineering integrates some arts: code breaking, puzzle solving, programming, and logical analysis [37]. The process is used by a diversity of different people for a variety of different purposes, many of which will be discussed all over this book.

2. Reversing Applications

It would be fair to say that in most industries reverse engineering for the purpose of developing a competitor product is the most well-known application of reverse engineering. The interesting thing is that it really isn't as popular in the software industry as one would anticipate [17]. There are several reasons for this, but it is essentially because software is so complex that in many cases reverse engineering for ruthless purposes is thought to be such a complex process that it just doesn't make sense financially [36]. So, what are the usual applications of reverse engineering in the software world? usually speaking, there are two groups of reverse engineering applications: security-related and software development–related. The following segment presents diverse reversing applications in both categories.

2.1 Security-Related Reversing

For some people the connection between security and reversing might not be right away clear. Reversing is related to some different aspects of computer security [5]. For example, reversing has been employed in encryption research—a researcher reverses an encryption product and assesses the worth of the level of security it provides. Reversing is also laboriously used in connection with malicious software, on both ends of the barrier: it is used by both malware developers and those developing the antidotes. Eventually, reversing is very popular with crackers who use it to examine and eventually beat various copy protection schemes [7]. All of these applications are converse about in the sections that follow.

2.1.1 Malicious Software

The Internet has completely changed the computer industry in general and the security-related feature of computing in particular. Malicious software, such as viruses and worms, spreads so much quickly in a world where millions of users are connected to the Internet and use email daily. Just 10 years ago, a virus would usually have to copy itself to a floppy and that diskette would have to be loaded into a further computer in order for the virus to spread [47]. The infection process was justly slow, and defense was much simpler because the channels of infection were few and required human involvement for the program to spread. That is all ancient history—the Internet has created an essential connection between almost every computer on earth. Nowadays modern worms can spread automatically to millions of computers without any human involvement [59]. Reversing is used extensively in both ends of the hostile software chain. Developers of hostile software often use reversing to discover vulnerabilities in operating systems and other software [26]. Such vulnerabilities can be used to penetrate the system's defense layers and allow infection—usually over the Internet. After infection, culprits sometimes employ reversing techniques to locate software vulnerabilities that allow a malicious program to gain access to responsive information or even take full control of the system. At the other end of the chain, developers of antivirus software dissect and examine every malicious program that falls into their hands. They use reversing methods to trace each step the program takes and assess the harm it could cause, the expected rate of infection, how it could be removed from infected systems, and whether infection can be stay away from altogether [34]. Serves as an introduction to the world of malicious software and demonstrates how reversing is used by antivirus program writers. Demonstrates how application vulnerabilities can be located using reversing methods.

2.1.2 Reversing Cryptographic Algorithms

Cryptography has always been based on confidentiality: Alice sends a message to Bob and encrypts that message using a secret that is only known to her and Bob. Cryptographic algorithms can be severely divided into two groups:

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.488 |

|| Volume 9, Issue 5, May 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0905032 |

constricted algorithms and key-based algorithms. Restricted algorithms are the kind some kids play with; writing a letter to a friend with each letter moved several letters up or down. The secret in constricted algorithms is the algorithm itself. Once the algorithm is revealing, it is no longer secure. Restricted algorithms provide very poor security because reversing makes it very difficult to maintain the confidentiality of the algorithm. Once reversers get their hands on the encrypting or decrypting program, it is only a matter of time before the algorithm is uncovered. Because the algorithm is the secret, reversing can be seen as a way to break the algorithm [55]. In other hand, in key-based algorithms, the secret is a key, some numeric value that is used by the algorithm to encrypt and decrypt the message. In key-based algorithms users encrypt messages using keys that are kept personal. The algorithms are usually made public, and the keys are kept private legitimate recipients [23]. This almost makes reversing pointless because the algorithm is already familiar. In order to decipher a message encrypted with a key-based cipher, you would have to either obtain the key Try all possible amalgamation until you get to the key Look for a flaw in the algorithm that can be employed to take out the key or the original message Still, there are cases where it makes sense to reverse engineer private implementations of key-based ciphers [35]. Even when the encryption algorithm is well-known, specific implementation details can often have an unpredicted impact on the overall level of security offered by a program. Encryption algorithms are delicate, and minor execution errors can sometimes completely invalidate the level of security offered by such algorithms. The only way to really know for sure whether a security product that instrument an encryption algorithm is truly secure is to either go through its source code or to reverse it.

2.1.3 Digital Rights Management

Nowadays, computers have turned most types of copyrighted materials into digital information. Music, films, and even books, which were once only available on physical equal mediums, are now available digitally. This trend is a mixed blessing, on condition that huge benefits to consumers, and huge complications to copyright owners and content providers [37]. For consumers, it means that materials have increased in quality, and become easily reachable and simple to manage. For providers, it has enabled the administration of high-quality content at low cost, but more importantly, it has made controlling the flow of such content an impossible mission. Digital information is extremely fluid. It is very easy to move on all sides and can be very easily duplicated [28]. This fluidity means that once the copyrighted materials reach the hands of purchasers, they can be moved and duplicated so easily that piracy almost becomes common practice. Traditionally, software companies have dealt with freebooting by embedding copy protection technologies into their software. These are additional pieces of software embedded on top of the vendor's software product that attempt to prevent or restrict users from copying the program [36]. In recent years, as digital media transmute a reality, media content providers have developed or obtained technologies that control the distribution of such content such as music, movies, etc. These technologies are impartially called digital rights management (DRM) technologies.

DRM technologies are conceptually very alike to traditional software copy protection technologies discussed above [62]. The difference is that with software, the thing which is being kept safe is active or “intelligent,” and can decide whether to make itself available or not. Digital media is a passive element that is generally played or read by another program, making it more difficult to control or control usage. Throughout this book I will use the term DRM to narrate both types of technologies and specifically refer to media or software DRM technologies where relevant [46]. This topic is most related to reverse engineering because crackers routinely use reverse-engineering techniques while attempting to defeat DRM technologies. The reason for this is that to conquer a DRM technology one must understand how it works. By using reversing techniques, a cracker can learn the internal secrets of the technology and discover the simplest possible tempering that could be made to the program in order to disable the protection [56]. I will be discussing the subject of DRM technologies and how they relate to reversing in more profundity.

2.1.4 Auditing Program Binaries

One of the strengths of open-source software is that it is often inherently more reliable and secure [4]. Regardless anyhow s of the real security it provides, it just feels much safer to run software that has often been inspected and approved by thousands of unbiased software engineers. inessential to say, open-source software also provides some real, touchable quality benefits [6]. With open-source software, having open access to the program's source code means that definite vulnerabilities and security holes can be discovered very early on, often before malicious programs can take advantage of them. With protected software for which source code is unavailable, reversing becomes a workable alternative for searching for security vulnerabilities [31]. Of course, reverse engineering cannot make protected software nearly as accessible and legible as open-source software, but strong reversing skills enable one to view code and assess the various security risks it poses. I will be exhibiting this kind of reverse engineering

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.488 |

|| Volume 9, Issue 5, May 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0905032 |

2.2 Reversing in Software Development
Reversing can be thoroughly useful to software developers. For instance, software developers can employ reversing techniques to discover how to interoperate with unconfirmed or partially documented software. In other cases, reversing can be used to conclusive the quality of third-party code, such as a code library or even an operating system [21]. Finally, it is sometimes possible to use reversing skillfulness for extracting valuable information from a competitor's product for the purpose of upgrading your own technologies. The applications of reversing in software development are talked through in the following sections.

2.2.1 Achieving Interoperability with Proprietary Software
Interoperability is where comparable software engineers can arrange from reversing almost daily. When working with a proprietary application library or operating system API, documentation is almost always inadequate [44]. Nonetheless of how much trouble the library vendor has taken to make sure that all feasible cases are covered in the documentation, users almost always find themselves scratching their heads with unanswered questions [54]. Most designers will either be constant and keep trying to somehow get things to work, or contact the vendor for answers. Any other way, those with reversing skills will often find it remarkably easy to deal with such situations [56]. Using reversing it is practicable to resolve many of these issues in very little time and with a relatively small effort. Demonstrate several different software's for reversing in the context of carry out compatibility.

2.2.2. Developing Competing Software
As I've earlier mentioned, in most companies this is by far the most popular application of reverse engineering. Software tends to be far more complex than most products, and so reversing an entire software product in order to create a competing product just doesn't make any sense [48]. It is usually much more accessible to design and develop an outcome from scratch, or simply license the more complex components from a third party rather than develop them in-house. In the software industry, even if a competitor has an unpatented technology (and I'll get into patent/trade-secret issues later in this chapter), it would never make sense to reverse engineer their entire product. It is commonly easier to self-reliant develop your own software [52]. The anomaly is highly complex or unique designs/algorithms that are very challenging or costly to develop. In such cases, most of the application would still have to be developed independently, but highly complex or unusual components might be reversed and reimplemented in the new product. The lawful aspects of this type of reverse engineering are talked about in the legal section.

2.2.3. Evaluating Software Quality and Robustness
Just as it's possible to audit a program binary to gauge its security and vulnerability, it's also possible to undertake and sample a program binary so as to urge an estimate of the overall quality of the coding practices utilized in the program [32]. The necessity is extremely similar: open-source software is an open book that permits its users to gauge its quality before committing thereto. Software vendors that do not publish their software's ASCII text file are essentially asking their customers to “just trust them.” It's like buying a second-hand car where you only can't crop up the hood. you've got no idea what you're really buying. the necessity for having source-code access to key software products like operating systems has been made clear by large corporations; several years ago, Microsoft announced that enormous customers purchasing over 1,000 seats may obtain access to the Windows ASCII text file for evaluation purposes. Those that lack the purchasing power to convince a serious corporation to grant them access to the merchandise 's ASCII text file must either take the company's word that the product is well built, or resort to reversing [21]. Again, reversing would never reveal the maximum amount about the product's code quality and overall reliability as taking a glance at the ASCII text file, but it is often highly informative. There are not any special techniques required here [58]. As soon as you're comfortable enough with reversing that you simply can fairly quickly re-evaluate code, you'll use that ability to undertake and evaluate its quality. This book provides everything you would like to try.

III. LOW-LEVEL SOFTWARE

Low-level software (also referred to as system software) may be a generic name for the infrastructure of the software world. It encompasses development tools like compilers, linkers, and debuggers, infrastructure software like operating systems, and low-level programming languages like programming language [61]. It's the layer that isolates software developers and application programs from the physical hardware. The event tools isolate software developers from processor architectures and assembly languages, while operating systems isolate software developers from specific hardware devices and simplify the interaction with the top user by managing the display, the mouse, the keyboard, and so on [43]. Years ago, programmers always had to figure at this low level because it had been the sole possible thanks to writing software—the low-level infrastructure just didn't exist. Nowadays, modern operating systems and

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.488 |

|| Volume 9, Issue 5, May 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0905032 |

development tools aim at isolating us from the small print of the low-level world. This greatly simplifies the method of software development, but comes at the value of reduced power and control over the system. So to become an accomplished reverse engineer, you want to develop a solid understanding of low-level software and low-level programming. That's because the low-level aspects of a program are often the sole thing you've got to figure out as a reverser—high-level details are nearly always eliminated before a software program is shipped to customers [57]. Mastering low-level software and therefore the various software-engineering concepts is simply as important as mastering the particular reversing techniques if one is to become an accomplished reverser. A key concept about reversing which will become painfully clear later during this book is that reversing tools like disassemblers or decompiled never actually provide the answers—they merely present the knowledge. Eventually, it's always up to the reverser to extract anything meaningful from that information. so as to successfully extract information during a reversing session, reversers must understand the varied aspects of low-level software. So, what exactly is low-level software? Computers and software are built layers upon layers. At the rock bottom layer, there are many microscopic transistors pulsating at incomprehensible speeds. At the highest layer, there are some elegant looking graphics, a keyboard, and a mouse—the user experience [31]. Most application developers use high-level languages that take effortlessly understandable commands and carry out them. for example, commands that make a window, load an internet page, or display an image are incredibly high-level, meaning that they translate to thousands or maybe many commands within the lower layers. Reversing requires a solid understanding of those lower layers. Reversers must literally remember anything that comes between the program ASCII text file and therefore the CPU. The subsequent sections introduce those aspects of low-level software that are mandatory for successful reversing [59].

3.1. Assembly Language

Assembly language is the lowest level within the software chain, which makes it incredibly suitable for reversing—nothing moves without it. If software performs an operation, it must be visible within the programming language code [12]. programming language is the language of reversing. To master the planet of reversing, one must develop a solid understanding of the chosen platform's programming language. Which brings us to the foremost basic point to recollect about assembly language: it's a category of languages, not one language. Every computer platform has its own programming language that's usually quite different from all the remainder. Another important concept to urge out of the way is machine language (often called code, or object code). People sometimes make the error of thinking that machine language is “faster” or “lower-level” than programming language. That's a misconception: machine language and programming language are two different representations of an equivalent thing [32]. A CPU reads machine language, which is nothing but sequences of bits that contain an inventory of instructions for the CPU to perform. programming language is just a textual representation of these bits—we name elements in these code sequences so as to form them human-readable [63]. rather than cryptic hexadecimal numbers we will check out textual instruction names like MOV(Move), XCHG (Exchange), and so on. Each programming language command is represented by variety, called the order code, or opcode. code is actually a sequence of opcodes and other numbers utilized in reference to the opcodes to perform operations. CPUs constantly read code from memory, decode it, and act to support the instructions embedded in it. When developers write code in programming language (a fairly rare occurrence these days), they use an assembler program to translate the textual programming language code into code, which may be decoded by a CPU [33]. Within the other direction and more relevant to our narrative, a disassembler does the precise opposite. It reads code and generates the textual mapping of every instruction in it. This is often a comparatively simple operation to perform because the textual programming language is just a special representation of the thing code. Disassemblers are a key tool for reversers and are discussed in additional depth later during this chapter. Because programming language may be a platform-specific affair, we'd like to settle on a selected platform to specialize in while studying the language and practicing reversing. I've decided to specialize in the Intel IA-32 architecture, on which each 32-bit PC is predicated. This choice is a simple one to form, considering the recognition of PCs and of this architecture [42]. IA-32 is one among the foremost common CPU architectures within the world, and if you're planning on learning reversing and programming language and haven't any specific platform in mind, accompany IA-32. The architecture and programming language of IA-32-based CPUs are introduced.

3.2. Compilers

So, considering that the CPU can only run machine language, how are the favored programming languages like C++ and Java translated into machine code? A document containing instructions that describe the program during an application-oriented language is fed into a compiler [37]. A compiler may be a program that takes a source file and generates a corresponding machine language file. counting on the application-oriented language, this machine language can either be a typical platform-specific code that's decoded directly by the CPU or it is often encoded during a special platform-independent format called bytecode (see the subsequent section on bytecodes). Compilers of traditional (nonbytecode-based) programming languages like C and C++ directly generate machine-readable code from the textual ASCII text file. What this suggests is that the resulting code, when translated to programming language by a disassembler, is actually a machine-generated programming language program [47]. Of course, it's not entirely machine-generated, because the software developer described to the compiler what needed to be wiped out the application-oriented language [64]. But the small print of how things are administered are taken care of by the compiler, within the resulting code. This is often a crucial point because this code isn't always easily understandable, even in comparison to a man-made programming language program—machines think differently than citizenry [27]. The most important hurdle in deciphering compiler-generated code is the optimizations applied by latest compilers. Compilers employ a spread of techniques that minimize code size and improve execution performance. The matter is that the resulting optimized code is usually counterintuitive and difficult to read. As an example, optimizing compilers often replace straightforward instructions with mathematically equivalent operations whose purpose is often faraway from obvious initial glance. Significant portions of this book are dedicated to the art of deciphering machine-generated programming language [43]. we'll be studying some compiler basics and proceed to specific techniques which will be wont to extract meaningful information from compiler-generated code.

3.3 Virtual Machines and Bytecodes
Compilers for high-level languages like Java generate a bytecode rather than a code. Bytecodes are almost like object codes, except that they're usually decoded by a program, rather than a CPU. the thought is to possess a compiler, generate the bytecode, and to then use a program called a virtual machine to decode the bytecode and perform the operations described in it [52]. Of course, the virtual machine itself must at some point convert the bytecode into standard code that's compatible with the underlying CPU. There are diverse vital benefits to using bytecode-based languages. One significant advantage is platform independence [21]. The virtual machine is often ported to different platforms, which enables running an equivalent binary program on any CPU as long because it features a compatible virtual machine. Of course, no matter which platform the virtual machine is currently running on, the bytecode format stays an equivalent. This suggests that theoretically software developers don't get to worry about platform compatibility. All they need to do is provide their customers with a bytecode version of their program. Customers must successively obtain a virtual machine that's compatible with both the precise bytecode language and with their specific platform. The strategy should then run on the user's platform with no changes or platform-specific work. This book primarily focuses on reverse engineering of native executable programs generated by native machine language compilers [43]. Reversing programs written in bytecode-based languages is a completely different process that's often much simpler compared to the method of reversing native executables. focuses on reversing techniques for programs written for Microsoft's. NET platform, which uses a virtual machine and a low-level bytecode language.

3.4 Operating Systems
An operating system is a program that manages the computer, including the hardware and software applications. An operating system takes care of many different tasks and can be seen as a kind of coordinator between the different elements in a computer [53]. Operating systems are such a key element in a computer that any reverser must have a good understanding of what they do and how they work. As we'll see later, many reversing techniques revolve around the operating system because the operating system serves as a gatekeeper that controls the link between applications and the outside world. Chapter 3 provides an introduction to modern operating system architectures and operating system internals, and demonstrates the connection between operating systems and reverse-engineering techniques.

IV. THE REVERSING PROCESS

How does one begin reversing? There are really many different approaches that work, and I'll try to discuss as many of them as possible throughout this book. For starters, I usually try to divide reversing sessions into two separate phases [15]. The first, which is really a kindly of wide-ranging observation of the earlier program, is called system-level reversing. System-level reversing approach helps determine the common structure of the program and sometimes even locate areas of interest within it. Once you intuit a general understanding of the layout of the program and determine areas of special interest within it you can proceed to more in-depth work using code-level reversing techniques. Code-level techniques provide detailed information on a selected code chunk [39]. The following sections describe each of the two techniques.

4.1 System-Level Reversing
System-level reversing involves running many tools on the program and applying various operating system assistance to obtain information, inspect program executables, track program input and output, and so forth [43]. Most of this

statistic comes from the operating system, because by definition every interaction that a scheme has with the outside world must go through the operating system. This is the motive why reversers must comprehend operating systems—they can be used during reversing sessions to obtain a wealth of information about the target program being investigated. I will be discussing operating system basics and proceed to introduce the various tools commonly used for system-level reversing.

4.2 Code-Level Reversing
Code-level reversing is basically a kind. Extracting design concepts and algorithms from a program binary may be a complex process that needs a mastery of reversing techniques alongside a solid understanding of software development, the CPU, and therefore the OS. Software is often highly complex, and even those with access to a program's well-written and well properly-documented ASCII text file are often amazed at how difficult it is often to grasp. Deciphering the sequences of low-level instructions that structure a program is typically no mean feat [65]. But fear not, the main target of this book is to supply you with the knowledge, tools, and techniques needed to perform effective code-level reversing. Before covering any actual techniques, you want to become conversant in some software-engineering essentials. Code-level reversing observes the code from a really low-level, and we'll be seeing every little detail of how the software operates [52]. Many of those details are generated automatically by the compiler and not manually by the software developer. This, which sometimes makes it difficult to know how they relate to the program and to its functionality. A that's why reversing requires a solid understanding of the low-level aspects of software, including the link between high-level and low-level programming constructs, programming language, and therefore the inner workings of compilers and the way they operate also will be very helpful. Compilers and other software-engineering essentials These topics are discussed.

V. THE TOOLS

Techniques that want to aid program understanding are often grouped into three categories: unaided browsing, leveraging corporate knowledge and knowledge, and computer-aided techniques like reverse engineering [27]. Unaided browsing is actually “HumanWare ": the engineer manually flips through ASCII text file in printed form or browses it online, perhaps using the filing system as a navigation aid. This approach has inherent limitations supporting the quantity of data that an engineer could also be ready to keep track [33]. Leveraging corporate knowledge and knowledge are often accomplished through mentoring or by conducting informal interviews with personnel intimate with the topic system. This approach is often very valuable if there are people available who are related to the system because it has evolved over time. They carry important details in their heads about architecture decisions, major changes over time, and troublesome subsystems.[43] However, leveraging corporate knowledge and knowledge isn't always possible. The first designers may have left the corporate. The software may be acquired from another company. Or the system may have had its perversions out-sourced. In these situations, computer-aided reverse engineering is important. A reverse-engineering environment can manage the complexities of program understanding by helping the engineer extract high-level information from low-level artifacts, like ASCII text files. This frees engineers from tedious, manual, and error-prone tasks like code reading, searching, and pattern matching by inspection [56].

5.1 System-Monitoring Tools
System-level reversing challenges a diversity of tools that sniff, monitor, explore, and otherwise expose the program being reversed [21]. These tools usually display information gathered by the operating system about the application and its environment. Because almost all communications in the middle of a program and the outside world go through the operating system, the operating system can usually be leveraged to extract such information. System-keep and eye tools can monitor networking activity, file accesses, registry access, and so on. There are also tools that expose a program's use of operating system objects such as mutexes, pipes, events, and so forth. Many of these tools will be discussed.

5.2 Disassemblers
As I explained earlier, disassemblers are programs that take a program's executable binary as input and creates textual files that have the assembly language code for the entire program or parts of it [44]. This is a fairly elementary process considering that assembly language code is simply the textual mapping of the object code. Disassembly is a processor-specific process, but some disassemblers support multiple CPU armature [16]. A high-quality disassembler is a key component in a reverser's toolkit, yet some reversers prefer to just use the built-in disassemblers that are embedded in certain low-level debuggers.

5.3 Debuggers

If you have ever attempted even the only software development, you've presumably used a debugger. The essential idea behind a debugger is that programmers can't really envision everything their program can do. Programs are usually just too complex for a person to actually predict every single potential outcome [51]. A debugger may be a program that permits software developers to watch their program while it's running. The 2 most elementary features during a debugger are the power to line breakpoints and therefore the ability to trace through code. Breakpoints allow users to pick a particular function or code line anywhere within the program and instruct the debugger to pause program execution once that line is reached. When the program reaches the breakpoint, the debugger stops (breaks) and displays the present state of the program [61]. At that time, it's possible to either release the debugger and therefore the program will continue running or to start out tracing through the program. Debuggers allow users to trace through a program while it's running (this is additionally referred to as single-stepping). Tracing means the program executes one line of code then freezes, allowing the user to watch or maybe alter the program's state. The user can then execute the subsequent line and repeat the method. this enables developers to look at the precise flow of a program at a pace more appropriate for human comprehension, which is a few billion times slower than the pace the program usually runs in [28]. By installing breakpoints and tracing through programs, developers can watch a program closely because it executes a problematic section of code and checks out to work out the source of the matter. Because developers have access to the ASCII text file of their program, debuggers present the program in source-code form, and permit developers to line breakpoints and trace through source lines, albeit the debugger is really working with the machine language underneath. For a reverser, the debugger is nearly as important because it is to a software developer, except for slightly different reasons. reversers use debuggers in disassembly mode [38]. In disassembly mode, a debugger uses a built-in disassembler to disassemble code on the fly. Reversers can step through the disassembled code and essentially “watch” the CPU as it's running the program one instruction at a time. Even as with the source-level debugging performed by software developers, reversers can install breakpoints in places of interest within the disassembled code then examine the state of the program. for a few reversing tasks, the sole thing you're getting to need may be a good debugger with good built-in disassembly capabilities. having the ability to step through the code and watch because it is executed is basically a useful element within the reversing process.

5.4. Decompilers

Decompilers are subsequently intensified from disassemblers. A decompiler takes an executable computer file and attempts to supply readable application-oriented language code from it. The thought is to undertake and reverse the compilation process, to get the first source file or something almost like it [31]. On the overwhelming majority of platforms, actual recovery of the first ASCII text file isn't possible. There are significant elements in most high-level languages that are just omitted during the compilation process and are impossible to recover [43]. Still, decompilers are powerful tools that in some situations and environments can reconstruct a highly readable ASCII text file from a program binary. Chapter 13 discusses the method of decompilation and its limitations, and demonstrates just how effective it is often.

VI. REVERSE ENGINEERING PERMISSIBLE

The legal debate around reverse engineering has been happening for years. it always revolves round the question of what social and economic impact reverse engineering has on society as an entire [25]. Of course, calculating this type of impact largely depends on what reverse engineering is employed for. The subsequent sections discuss the legal aspects of the varied applications of reverse engineering, with a stress on the us. It should be noted that it's never getting to be possible to accurately predict beforehand whether a specific reversing scenario goes to be considered legal or not—that depends on many factors [47]. Always seek legal counsel before going yourself into any high-risk reversing project. the subsequent sections should provide general guidelines on what sorts of scenarios should be considered high risk.

6.1. Interoperability

obtaining two programs to conduct and interoperate is never an easy task. Even within a one product mature by a single group of people, there are frequently interfacing problems caused when attempting to get individual components to interoperate [63]. Application annexation is so complex and the programs are so sensitive that these things rarely function properly on the first attempt. When an application creator wishes to develop software that communicates with a component developed by another company, there are large amounts of information that must be exposed by the other party regarding the interfaces [22]. A software platform is any program or hardware device that programs can run on top for a software platform developer, the decision of whether to publish or to not publish the details of the platform's

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.488 |

|| Volume 9, Issue 5, May 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0905032 |

software interfaces is a critical one. On one hand, exposing application interfaces means that another developer will be able to develop an application that runs on top of the platform. This could drive sales of the platform upward, but the vendor might also be offering their own application that uses the platform. Publishing application interfaces would also create new conflict for the vendor's own applications [57]. The various legit characteristics that affect this type of reverse engineering such as copyright laws, trade secret protections, and patents are talked about in the following sections.

6.2. Competition

When used for interoperability, reverse engineering clearly benefits society because it simplifies (or enables) the event of latest and improved technologies. When reverse engineering is employed within the development of competing products, things are slightly more complicated [34]. Opponents of reverse engineering usually claim that reversing stifles innovation because developers of latest technologies have little incentive to take a position in research and development if their technologies are often easily “stolen” by competitors through reverse engineering. This brings us to the question of what exactly constitutes reverse engineering for the aim of developing a competing product. The foremost extreme example is to directly steal code segments from a competitor's product and embed them into your own. This is often a transparent violation of copyright laws and is usually very easy to prove. A more complicated example is to use some quite complicated process to a program and recompile its output during a way that generates a binary with identical functionality but with seemingly different code [43]. This is often almost like the previous example, except that during this case it'd be much more difficult to prove that code had actually been stolen. Finally, a more relevant (and ethical) quite reverse engineering during a competing product situation is one where reverse engineering is applied only to small parts of a product and is merely used for the gathering of data, and not code [8]. In these cases, most of the merchandise is developed independently with no use of reverse engineering and only the foremost complex and unique areas of the competitor's product are reverse engineered and reimplemented within the new product.

6.3. Copyright Law

Copyright laws aim to guard software and other property from any quite unauthorized duplication, and so on. The simplest example of where copyright laws apply to reverse engineering is within the development of competing software [54]. As I described earlier, in software there's a really fine line between directly stealing a competitor's code and reimplementing it. One thing that's generally considered a violation of copyright law is to directly copy protected code sequences from a competitor's product into your own product, but there are other, much more indefinite cases [33]. How does copyright law affect the method of reverse engineering a competitor's code for the aim of reimplementing it in your own product? Within the past, opponents of reverse engineering have claimed that this process violates copyright law due to the creation of intermediate copies during the reverse-engineering process. Consider the decompilation of a program as an example. so as to decompile a program, that program must be duplicated a minimum of once, either in memory, on disk, or both. The thought is that albeit the particular decompilation is legal, this intermediate copying violates copyright law. However, this claim has not been delayed in courts; there are several cases including Sega v. Accolade and Sony v. Connectix, where intermediate copying was considered use, primarily because the ultimate product didn't actually contain anything that was directly copied from the first product. From a technological perspective, this makes perfect sense—intermediate copies are always created while software is getting used, no matter reverse engineering [31]. Consider what happens when a program is installed from an optical media like a DVD-ROM onto a hard-drive—a copy of the software is formed. This happens again when that program is launched—the executable file on disk is duplicated into memory so as for the code to be executed.

6.4. License Agreement Considerations

In light of the fact that other than the DMCA there are no rules that directly forbid or restrict revoke, and that the DMCA only applies to DRM products or to application that contains DRM mechanization, Software vendors add anti-reverse-engineering clauses to shrink-wrap software license agreements [41]. That's that very lengthy document you're always told to “accept” when installing practically any software package within the world. It should be noted that in most cases just employing a program provides the legal equivalent of signing its license agreement (assuming that the user is given a chance to view it). The main legit query around reverse-engineering clauses in license agreements is whether they are enforceable [45]. In the U.S., there doesn't seem to be one, authoritative answer to the present question—it all depends on the precise circumstances during which reverse engineering is undertaken. In the European Union this problem has been certainly explained by the Directive on the Legal Protection of Computer Programs [EC1][31]. This direction defines that decompilation of application programs is permissible in cases of interoperability. The directive overrides any shrink-wrap license agreements, a minimum of during this matter.

VII. PROPOSED ANALYSIS APPROACH

The past years saw the development of several program analysis techniques and tools. Some of them rely on static analysis techniques, while recent years have seen an in-creasing use of dynamic analysis as an effective way to complement static analysis: even though dynamic analysis can be costly and incomplete, it is required to deal with many reverse engineering issues where just static analysis does not suffice [47]. Currently, several analysis and transformation toolkits are available to help reverse engineers in their tasks. For example, the Design Maintenance System (DMS) [9] developed by Semantic Designs, the TXL, or the Strategic Toolkit. This tool provides facilities for parsing the source code and doing rule-based transformations. been found that the presence of clones is not necessarily bad smell and a harmful phenomenon: provided that maintainers are aware of their presence, clones constitute a widely adopted appliance to facilitate software development and reuse [28]. Duplicate erase, on the other hand, may be hazardous and under-sired. Aspect oriented coding depicts new frontiers of software development. It addresses the problem of crosscutting matter, i.e., of features spread across many modules, with a new modularization unit, the aspect, that encapsulates them. To support the maintenance of crosscut-ting concerns, as well as to refactor them into aspects, it is needed to identify them into the source code. With this in intellect, several points mining approaches have been developed [43]. They are on the analysis of method fan-in, or dynamic analysis of execution traces.

VIII. FUTURE SCOPE

Future trends of Reverse Engineering In this section we will discusses future reverse engineering trends related to the three main areas of reverse engineering Future trends in program analysis One of the key challenges of program analysis for today and tomorrow is to deal with high dynamicity. Many Coding languages widely used today allow for high energetically, which constitutes a powerful development mechanism, but makes analysis more difficult [18]. For example, languages like Java introduce the concept of reflection and the studying software evolution, for analyzing the developers 'behavior or to correlate different software characteristics —e.g., faultiness with metric profiles — recently soft-ware repository information has been used as an alternative or complementary way of software analysis. For example, Geigerite. Related clones with software repositories co-changes, and camphorate. with crosscutting concerns. From the last 10 years, software analysis moved from a single dimension to two dimensions and, finally, today's opportunity is to add a third dimension consisting in the historical analysis of data extracted from software repositories [26]. Change diffs between file revisions, change coupling relationships between files being co-changed during the same time window, but also relationships between bug reports and code repositories, and the change rationale as documented in problem fixing reports and in CVS messages, add up to valuable sources of information supportive to static and dynamic analysis.

Future trends in design recovery
Forthcoming research related to design recovery needs to be able to deal with design paradigms which analysts and software architects are currently using [32]. While a lot of work has been finished to extract UML documentation from the source code, a lot still needs to be done in specific for what regards the extraction of dynamic diagrams and also of Object Constraint Language (OCL) pre- and post-conditions. On the other hand, there is the requirement to develop new innovative design recovery approaches and tools for new software architectures, that have characteristics of being extremely dynamic, highly distributed, self-configurable and heterogeneous [6]. We will discuss this problem for the analysis of Service Oriented Architectures (SOA). We highlighted the important role played by WA reverse engineering within the last ten years. Web applications are now moving towards Web 2.0, and this may have non-negligible effects on reverse engineering [22]. For example, there'll be the necessity for techniques ready to support the migration of existing WAs — composed of multiple pages interacting with the users by means of HTML forms — to-wards Web 2.0 applications where the user interacts with a single page. Further projects will be linked to the capability of reverse engineering to deal with the number of new technologies being used into new WAs.

IX. CASE STUDY

Sega versus Accolade: In 1990 Sega Enterprises, a well-known Japanese gaming company, released their Genesis gaming console. The Genesis's programming interfaces weren't published. The thought was for Sega and their licensed affiliates to be the sole developers of games for the console. Accolade, a California-based game developer, was curious about developing new games for the Sega Genesis and in porting a number of their existing games to the Genesis platform. Accolade explored the choice of becoming a Sega licensee, but quickly abandoned the thought because Sega required that each one games be exclusively manufactured for the Genesis console. rather than becoming a Sega licensee Accolade decided to use reverse engineering to get the small print necessary to port their games to the Genesis platform. Accolade reverse engineered portions of the Genesis console and a number of other of Sega's game cartridges. Accolade engineers then used the knowledge gathered in these reverse-engineering sessions to supply a document that described their findings. This internal document was essentially the missing documentation describing the way to develop games for the Sega Genesis console. Accolade successfully developed and sold several games for the Genesis platform, and in October of 1991 was sued by Sega for infringement of copyright. The first claim made by Sega was that copies made by Accolade during the reverse-engineering process (known as “intermediate copying”) violated copyright laws. The court eventually ruled in Accolade's favor because Accolade's games didn't actually contain any of Sega's code, and since of the general public benefit resulting from Accolade's work (by way of introducing additional competition within the market). This was a crucial landmark within the legal history of reverse engineering because during this ruling the court essentially authorized reverse engineering for the aim of interoperability.

X. CONCLUSION

Choosing reverse engineering technology in industry has been very slow. South reverse engineering is performed to maintain legacy code. Therefore, it should not be focused on program understanding but on system maintenance instead. This should be done in a way that frees us from reverse engineering a system again and again because of modifications made to its code over time. Researchers will continue to develop technology and tools for generic reverse engineering tasks, particularly for data reverse engineering but future research ought to focus on ways to make the process of reverse engineering more repeatable, defined, managed, and optimized [33]. For large evolving systems, forward and reverse engineering processes have to be integrated and achieve the same appreciation for product and process improvement for long-term evolution as for the initial development phases. It is tough to predict all needing things of the reverse engineers and, that's why, tools must be developed that are end-user programmable. Pervasive scripting is one successful strategy to allow the user to codify, customize, and automate continuous understanding activities and, at the same time, integrate the reverse engineering tools into personal development process and environment. Infrastructures for tool integration have evolved dramatically whereas it is expected that control, data, and presentation integration technology will continue to advance at amazing rates [14]. Even if we perfect reverse engineering technology, there are inherent high costs and risks in evolving legacy systems. Developing strategies to control these costs and risks is a key research direction for the future. The premise that reverses engineering needs to be applied continuously throughout the lifetime of the software and that it is important to understand and potentially reconstruct the earliest design and architectural decisions has major tool design implications. Reverse engineering has to be focused on design recovery [38]. As we need sophisticated tool support to perform system maintenance efficiently, the recovered design has to be based on formal methods.

REFERENCES

[1] M. Lehman., Programs, life cycles and laws of software evolution, Proceedings of IEEE Special Issue on Software Engineering, 68(9), 1980, 1060–1076.

[2] V.R. Basili and H.D. Mills., Understanding and Documenting Programs. IEEE Transactions on Software Engineering,(8), 1982, 270-283.

[3] T. Guimaraes., Managing Application Program Maintenance Expenditures. Communications of the ACM,26(10), 1983, 739-746.

[4] J. Ning., A Knowledge-based Approach to Automatic Program Analysis. PhD thesis, Department of Computer Science, University of Illinois at Urbana, Champaign, 1989.

[5] E. Chikofsky and J. Cross., Reverse engineering and design recovery: A taxonomy,IEEE Software, 7(1),1990, 13–17.

[6] C. Rich and L. Wills., Recognizing a Program's Design: A Graph Parsing Approach, IEEE Software7(1), 1990, 82-89.

[7] I.D. Baxter. Transformational Maintenance by Reuse of Design Histories. PhD thesis, University of California, Irvine, 1990.

[8] V. Kozaczynski, J.Q. Ning, and A. Engberts., Program Concept Recognition and Transformation, IEEE Transactions on Software Engineering,18(12), 1992, 1065-1075.

[9] I.D. Baxter., Design Maintenance Systems. Communications of the ACM,35(4), 1992, 73-89.

[10] P. Tonella and R. Fiutem and G. Antonoil and E. Merlo. Augmenting Pattern-Based Architectural Recovery with Flow Analysis: Mosaic -A Case Study, Working Conference on Reverse Engineering,1996,198-207.
[11] T. Munakata., Knowledge discovery. Communications of the ACM,42(11), 1999, 26–29.
[12] M. Blaha. Reverse Engineering of Vendor Databases, Working Conference on Reverse Engineering (WCRE-98), Honolulu, Hawaii, USA, 1998,183–190.
[13] S. R. Tilley., Coming attractions in program understanding II: Highlights of 1997 and opportunities for 1998.Technical Report CMU/SEI-98-TR-001, Carnegie Mellon Software Engineering Institute, 1998.
[14] R. Clayton, S. Rugaber, and L. Wills., The knowledge required to understand a program, Proceedings of the 5thWorking Conference on Reverse Engineering (WCRE-98), Honolulu, Hawaii, USA, 1998,69–78.
[15] J. Singer and T. Lethbridge., Studying work practices to help tool design in software engineering, Proceedings of the 6thInternational Workshop on Program Comprehension (WPC- 98), Ischia, Italy, 1998,173–179.[16]. A. Blackwell., Questionable practices: The use of questionnaire in psychology of programming research, The Psychology of Programming Interest Group Newsletter, 1998.
[16] R. Kasman and S. Carrie,,re.,Playing detective: Reconstructing software architecture from available evidence. Journal of Automated Software Engineering, 6(2),1999,107–138.
[17] Zhang D, Wang J and Yang Y 2014, Design 3D garments for scanned human bodies, Journal of Mechanical Science and Technology, 28 (7) 2479-2487
[18] K. Wong. Reverse Engineering Notebook, PhD thesis, Department of Computer Science, University of Victoria, 1999.
[19] B. Boehm. Software Engineering Economics. (Prentice-Hall, New York,1981).
[20] C. Rich and R.C. Waters., The Programmer’s Apprentice. (ACM Press,1990).
[21] H.A. Partsch., Specification and Transformation of Programs: A Formal Approach to Software Development. (Springe, 1990).
[22] J. Nielsen.,Usability Engineering. (Academic Press, New York, 1994).
[23] A. Umar., Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies. (Prentice Hall, New York, 1997).
[24] L. Bass, P. Clements, and R. Kazma., Software Architecture in Practice. (Addison-Wesley, 1997).B. Shneiderman.Designing the User Interface: Strategies for Effective Human-Computer Interaction. (Third Edition, Addison-Wesley, 1998).
[25] B. Shneider man.,: Designing the User Interface: Strategies for Effective Human-Computer Interaction. (Third Edition, Addison-Wesley, 1998).
[26] S. R. Tilley., The Canonical Activities of Reverse Engineering. (Baltzer Science Publishers, The Netherlands.2000).
[27] Draghici G 1991, Ingeneria integrate a producer [Integrated product engineering], Ed. Euro bit, Timișoara, România.
[28] Raja V and Fernandes K 2008, Reverse Engineering an Industrial Perspective, Springer Series in Advanced Manufacturing, UK.
[29] Gibson I, Rosen D and Stucker B 2010, Additive Manufacturing Technologies. Rapid Prototyping to Direct Digital Manufacturing, Springer
[30] Geomagic Available: http://www.geomagic.com/en/products/design/overview. Accessed 12 April 2015
[31] Rapid Form Available: http://www.rapidform.com/products/xor/overview/ Accessed 12 January 2015
[32] CATIA Available: http://www.3ds.com/products-services/catia/solutions-by-industry/ Accessed 1May 2015
[33] Chang K and Chen C 2011, 3D Shape engineering and design parameterization, Computer - Aided Design and Applications. 8(5) 681-692.
[34] Duret A, Remy S and Du Cellier G 2010, Knowledge based Reverse Engineering – An approach for Reverse Engineering of a mechanical part. ASME Journal of Computing and Information Science in Engineering. 10(4) 044501-1-044501-4.
[35] Disrupt A, Remy S and Du Cellier G and Eymard B 2008, From a 3D point cloud to an engineering CAD model: a knowledge-product-based approach for reverse engineering. Virtual and Physical Prototyping,3(2)51-59.
[36] Pescara R and Oancea G 2012, CAD modeling of part assemblies using reverse engineering technique, Advanced Materials Research, 591-593 7-10. [11] Wang C 2010, An analysis and evaluation of fitness for shoe lasts and human feet. Computers in industry, 61(6) 532-540.

[37] Scanner COMET L3D Available: http://www.steinbichler.com/products/3d-scanning/comet-l3d, Accessed 15 February 2015 (13) Geomagic (2012) Tutorial Geomagic Studio 2012.

[38] Catia Documentation Available: http://www.catiadesign.org/_doc/v5r14/. Accessed 15 June 2014

[39] Albu A 2011, Contributed la elaborate constructive is ethnologic a calapoadelor in industrial de calamine [Contributions to the constructive and technological designing of the last like in the footwear industry] PhD Thesis, University of Oradea, Romania

[40] Maureena G and Mihai A 2003, Bazile protectories încălţămintei [Footwear design basics]. Ed. Perform antica, Iasi, Romania

[41] Zhang D, Wang J and Yang Y 2014, Design 3D garments for scanned human bodies, Journal of Mechanical Science and Technology, 28 (7) 2479-2487

[42] Várady T, Martin RR, Cox J. Reverse engineering of geometric models – an introduction. Aided Des 1997;29(4):255–68.

[43] Mutawalli S. Review of reverse engineering approaches. In: 23rd International conference on computers and industrial engineering, vol. 35 (1–2), 1998. p. 25–8.

[44] K. H. Lee, H. Woo, and T. Suk, "Data reduction methods for reverse engineering", International Journal of Advanced Manufacturing Technology, 17(10), pp. 735–743, 2001.

[45] K. H. Lee and H. Woo, "Accurate part shape acquisition for reverse engineering", Proceedings of 25th International Conference on Computers and Industrial Engineering, New Orleans, Louisiana, pp. 52–55, 29–31 March 1999.

[46] Yau HT, Mend CH. Automated CMM path planning for dimensional inspection of dies and molds having complex surfaces. Int J Mach Tool Manufacture 1995;35(6):861–76.

[47] Huang MC, Tai CC. The pre-processing of data points for curve fitting in Reverse Engineering. Int J Adv Manufact Technol 2000; 16:635–42.

[48] Tai CC, Huang MC. The processing of data points based on design intent in reverse engineering. Int J Mach Tools Manufact 2000; 40:1.

[49] Tai CC, Huang MC. The processing of data points based on design intent in reverse engineering. Int J Mach Tools Manufact 2000; 40:1.

[50] Lokesh, K., Jain, P.K.: choice of rapid prototyping technology. Advanced production Engineering & Management, vol. 5, 2 (2010), 75-84 [10] Sobh. T, Owen. J. Jaynes (1995) "Industrial Inspection and Reverse Engineering," Computer Vision and Image Understanding 61 (4), 468-474 [11] Varadi'd, Martin. R. (1997) "Reverse Engineering of Geometric Models," Computer-Aided Design 29 (4), 255- 268

[51] Draghici G 1991, Ingeneria integrate a producer [Integrated product engineering], Ed. Euro bit, Timișoara, România.

[52] Raja V and Fernandes K 2008, Reverse Engineering an Industrial Perspective, Springer Series in Advanced Manufacturing, UK.

[53] Gibson I, Rosen D and Stucker B 2010, Additive Manufacturing Technologies. Rapid Prototyping to Direct Digital Manufacturing, Springer

[54] Geomagic Available: http://www.geomagic.com/en/products/design/overview. Accessed on 12 April 2015

[55] Rapid Form Available: http://www.rapidform.com/products/xor/overview/. Accessed 12 January 2015

[56] CATIA Available: http://www.3ds.com/products-services/catia/solutions-by-industry/ Accessed 1May 2015

[57] Chang K and Chen C 2011, 3D Shape engineering and design parameterization, Computer - Aided Design and Applications. 8(5) 681-692.

[58] Disrupt A, Remy S and Du Cellier G 2010, Knowledge based Reverse Engineering – An approach for Reverse Engineering of a mechanical part. ASME Journal of Computing and Information Science in Engineering. 10(4) 044501-1-044501-4.

[59] Disrupt A, Remy S and Du Cellier G and Eymard B 2008, From a 3D point cloud to an engineering CAD model: a knowledge-product-based approach for reverse engineering. Virtual and Physical Prototyping, 3(2) 51-59.

[60] Pescara R and Oancea G 2012, CAD modeling of part assemblies using reverse engineering technique, Advanced Materials Research, 591-593 7-10.

[61] Wang C 2010, An analysis and evaluation of fitness for shoe lasts and human feet. Computers in industry, 61(6) 532-540.

[62] Scanner COMET L3D Available: http://www.steinbichler.com/products/3d-scanning/comet-l3d, Accessed 15 February 2015

[63] Geomagic (2012) Tutorial Geomagic Studio 2012.

[64] Catia Documentation Available: http://www.catiadesign.org/_doc/v5r14/. Accessed 15 June 2014

[65] Albu A 2011, Contribution la elaborate constructive is ethnologic a calapoadelor in industrial de calamine [Contributions to the constructive and technological designing of the last like in the footwear industry] PhD Thesis, University of Oradea, Romania

[66] Maureena G and Mihai A 2003, Bazile protectories încălţămintei [Footwear design basics]. Ed. Perform antica, Iasi, Romania