

Looking Forwards to Going Backwards: An Assessment of Current Reverse Engineering

Jonathan Francis Roscoe <jjr6@aber.ac.uk>
Department of Computer Science, Aberystwyth University

Abstract—Reverse engineering has been both a stigmatised and valued process in software development for many years; enhancing program comprehension, interoperability, robustness and enabling design recovery.

This introductory survey seeks to understand the uses for reverse engineering, review the developments and influences of techniques over the last twenty-six years and consider how things might change in the coming years.

Index Terms—countermeasures, copyright, design recovery, interoperability, maintenance, malware, piracy, reverse engineering, software auditing, system exploration

I. BACKGROUND

Although arguably a method as old as invention itself, it was not until 1985 that Rekoff [1] formally defined reverse engineering for electronic systems as "the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system ... without the benefit of any of the original drawings ... for the purpose of making a clone of the original hardware system". Shortly after, Chikofsky and Cross [2] defined the application of reversing engineering to software systems, suggesting that "while the hardware objective traditionally is to duplicate the system, the software objective is most often to gain sufficient design-level understanding to aid maintenance, strength enhancement or support replacement", these words are contrasted with those of Rekoff, in that they do not necessarily involve the cloning of a system, merely observation and understanding. From these and related literature [3] we can conclude that the broad objectives of any reverse engineering undertaking are:

To identify system components and their relationships for the creation of high or low level representations of a system.

Chikofsky and Cross define three levels of abstraction that make up the entities of a reverse engineering project:

- Requirements (a specification of the problem the software solves)
- Design (a specification of the solution)
- Implementation (coding, testing and functional deliverable)

They go on to describe "forward engineering", the axiomatic process of transitioning from requirements, to design, to implementation. In contrast, reverse engineering is the transition

from an implementation to its design, through examination of the components serving that form the implementation.

This survey asserts that reverse engineering is an extension of the design phase of development and involves the acquisition of knowledge to the enhancement of system design, this may be an iterative process. Figure 1 highlights the common steps of the software reverse engineering process and their typical role in development.

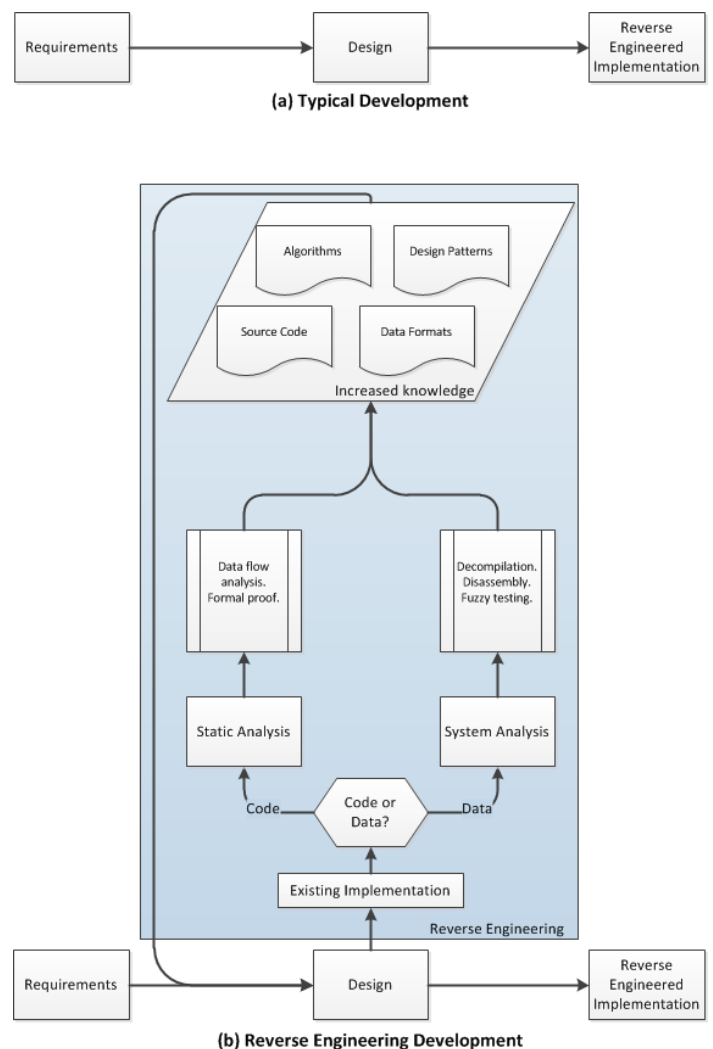


Figure 1. An original diagram: (a) indicates typical development steps (b) demonstrates how reverse engineering extends a typical design process with exploratory stages to enhance comprehension and gain knowledge on an existing implementation.

The results of reverse engineering endeavours may vary from discovering precise algorithms, to recovering design patterns or general system interactions. Depending on available resources and objectives, techniques may make use of various input data including: source code, communication between systems or components and information in memory. As evidenced by the variety of presented papers, reverse engineering can be applied to practically any system at any level, (e.g. hardware, protocol, application, etc).

II. APPLICATION AREAS

Fundamentally, reverse engineering concerns system exploration and design recovery. There are some unfortunate condemnations and reverse engineering and is often perceived as a disreputable activity, however, research suggests it can ease program understanding [4] and is, to a degree a tacit technique everywhere as developers attempt to familiarise themselves with new systems - the mental act of interpreting available information to understand the structure and design of a system is familiar to all developers and many modern IDEs provide facilities for reverse engineering, such as the generation of UML from source code [5].

Although used to varying degrees by all software developers and analysts, reverse engineering techniques have proven themselves extensively in a number of key areas. These application areas may overlap with one another to achieve specific objectives:

A. Interoperability

Reversing has been formally and most legitimately utilised for the visualisation and exploration of software [6]–[8] and is often used to go beyond the documentation of unfamiliar systems (e.g. third party, legacy) to allow developers to fully utilise available functionality that may be neglected from vendor documentation or to overcome limited interoperability with proprietary software. Even in an environment where full disclosure is intended, it is not uncommon for documentation to become outdated [9], forcing new developers to rediscover the system for maintenance.

A common task is the deciphering of file formats to allow interoperability with systems from other vendors, frequently seen in office suites [10]. Users often expect to be able to transfer information from one system to another with relative ease, which can be a significant task when a propriety format has become the de facto industry standard [11]. This can be an important issue for software developers entering an established market.

B. Cryptography

Cryptography has long been a famous field for mathematical reverse engineering [12]–[15] and software implementations are also being attacked. Weak cryptographic algorithms can

be analysed to determine which keys are used to generate output, leading to discovery of the algorithms through which registration keys and similar credentials are authenticated. Such attacks have been proven for real-world systems [16].

The concept of trapdoor (one- way functions) made publically aware by Diffie, Hellman [17] and Merkle went a long way to developing cryptographic systems that were relatively safe from reverse engineering. In recent years however, power analysis has been shown to facilitate the reversal of asymmetric ciphers with considerably greater effectiveness than traditional techniques [18].

C. Security Testing

Reverse engineering is highly effective in testing the security of software. Analysis of completed systems can reveal weak or exploitable points [19] that may reveal further design information [20] or provide those with the inclination to attack and manipulate the software for malicious purposes. Reverse engineering can be a complex process compared to other debugging techniques available to original developers but can be effective in discovering vulnerabilities at runtime using using black box techniques [21] or prior to compilation using static code analysis.

The creation of complex malware, such as trojans and worms is usually facilitated by the reverse engineering of exploits for target systems. "Patch Tuesday", the second Tuesday of each month is when Microsoft releases security related update for the Windows operating system, it is common for a rise in exploitation to occur on the day after as a result of reverse engineering of patches [14], [22].

Whilst malware developers seek to attack software, reversing also provides opportunity for anti-virus and security professionals to discover counter measures. Code obfuscation is a common technique by software developers to attempt to hide the intention and techniques of their software. Techniques have been developed that attack a range of obfuscation techniques including polymorphism and bytecode interpreters [23] enabling the discovery of malware execution steps, inner structure and mode of operation.

DllHijackAuditKit¹ is a tool for detecting vulnerabilities in Windows library files and generating proof of concepts. The tool was created as a means of disturbing vendor complacency and is based on research by Kwon and Su into automated discovery of dynamic component loading vulnerabilities [24]. This is a good example of the many tools available to malicious and professional users.

D. Behaviour Modification

With the correct understanding of binary applications, engineers can modify functionality to allow for cheats in games

¹<http://blog.metasploit.com/2010/08/better-faster-stronger.html>

[25] (which may or may not be malicious), extend functionality or disable components (e.g. bypass anti-virus systems). Although reverse engineering may lead to the creation of "cracks" - patches to binary files that manipulate program execution, reverse engineering "in and of itself does not involve changing the subject system. It is the process of examination, not change or replication" as Chikofsky and Cross [2] explain.

In computer games, the manipulation of memory at runtime and code injection are common ways to enable cheating and modify gameplay. Such techniques have also been used for rapid prototyping and provision of new functionality in legacy systems [26].

E. Competition

In recent years there has been a rise in the use of reverse engineering to develop competing products or services, this is particularly prevalent in web applications where services can be quickly, efficiently and economically deployed to fill a trending market [27], [28]. Though the fast creation of numerous web applications has led to a dramatically increasing number of these applications becoming legacy systems, as the short-lived development period gives way to neglected systems that necessitate further reversing to enable sensible maintenance [29].

Where a new manufacturer is attempting to enter an established market, it is difficult to attract users without the ability to provide adequately equivalent functionality as competitors, and typically, the ability to reuse user-data from competitor products. Reverse engineering is useful in enabling the loading of proprietary formats and understanding established functionality.

F. Copyright Protection

Reverse engineering is frequently employed to bypass copyright protection mechanisms to enable the distribution of counterfeit software [30]–[32]. This may involve behaviour modification, but more typically involves copying proprietary algorithms or understanding key generation to create counterfeit software keys.

DeCSS is a notable piece of software to circumvent copyright protection that gained notoriety due to legal action [33]. Developed with the intention of enabling DVD playback on Linux systems, commercial DVD software was disassembled to isolate a key required for playback. Most interestingly, the release of DeCSS source code led to reverse engineering of the CSS algorithm by further developers and the development of superior techniques and software.

III. TOOLS

Bellay and Gall have surmised from multiple investigations [34], [35] into reverse engineering tools a lack of comprehensibility of output and general dissimilarity between them. Tools

vary in their effectiveness and different analysis techniques are susceptible to different shortcomings. There is no universal tool and no tool can guarantee to be useful for all cases, but there is certainly a lot of choice and an option for most scenarios.

In many cases, reverse engineering tools attack the same problems as conventional development tools, in fact many traditional tools can be used for reverse engineering. These tools may be static and perform analysis on non-volatile data, or dynamic, which analyse system memory to assess changes being effected as an application runs. The two types of analysis often go hand-in-hand [36], [37].

The majority of effort in developing reverse engineering technologies has focussed on understanding and maintaining legacy systems [38], yet many tools have been created and repurposed for a variety of new challenges. The tools can be used to reveal algorithm details, design patterns, protocol information, code quality, exploits and a wealth of other information.

A. System Analysis Tools

One way to determine the interior design of a system is to examine the effect it has on other systems. Monitoring network, disk and memory usage for instance, can yield valuable information pertaining to system activity. System utilities² such as `ls`, `strings`, `tcpdump` and `lsof` can be used to this end. These tools perform dynamic analysis on live systems and tools are more commonly used in the field of digital forensics [39]. Some such tools may involve interacting with software to force the execution of certain paths, by acting as a user to explore the system; whilst others are passive.

B. Disassemblers and Decompilers

Source code is a high-level representation of a set of instructions for a computer system. Code is converted into suitable machine-language through a compilation process, with intermediate representations along the way. Examination of compiled code may be fundamental in validating compiler output for critical systems [40] or examining low level algorithm behaviour (a common task for software "crackers"). Disassemblers enable machine code (executable programs) to be translated into assembly, a representation of machine code that uses mnemonics to represent instructions, facilitating program interpretation.

Although there exists one-to-one mappings between machine code and assembly, there are numerous challenges. Schwarz et al point out that disassembly can be difficult to perform consistently in documentation and exploration of techniques for disassembly [41]. A linear sweep approach allows straightforward disassembly by mapping machine instructions directly to their assembler mnemonic. However, this approach ignores

²for an example of such a collection, see <http://www.gnu.org/software/coreutils/>

Original	Disassembled	Decompiled
<pre>int total = 0; int i; for (i=0; i < 10; i++) total += 1;</pre>	<pre>8048329: mov %esp,%ebp 804832b: sub \$0x8,%esp 804832e: and \$0xfffffffff0,%esp 8048331: mov \$0x0,%eax 8048336: sub%eax,%esp 8048338: movl \$0x0,0xfffffffffc(%ebp) 804833f: movl \$0x0,0xfffffffff8(%ebp) 8048346: cmpl \$0x9,0xfffffffff8(%ebp) 804834a: jle 804834e <main+0x26> 804834c: jmp 8048364 <main+0x3c> 804834e: mov 0xfffffffff8(%ebp),%eax 8048351: mov 0x8049460(,%eax,4),%edx 8048358: lea 0xfffffffffc(%ebp),%eax 804835b: add %edx,(%eax) 804835d: lea 0xfffffffff8(%ebp),%eax 8048360: incl (%eax) 8048362: jmp 8048346 <main+0x1e></pre>	<pre>int local1 = m[r28{0} - 8]; int local2 = m[r28{0} - 12]; local1 = 0; local2 = 0; while (local2 < 9) { local1 += 1; local2++; }</pre>

Figure 2. Example of differences between a small amount of code after disassembly and decompilation. Based on data from the Boomerang documentation (<http://boomerang.sourceforge.net/cando.php>).

program flow control and suffers from an inability to discern embedded data from operations. An alternative method is recursive traversal which determines flow and disassembles all branches of possible execution.

Assembly language is a low-level representation of machine code with little abstraction of instructions. Usually reserved for precise systems development, assembly can be a challenge for a non-specialist programmer to work with. Decompilers take disassembly one step further however, converting machine code into high-level equivalents. This is a considerably less reliable process than disassembly, as compilation is essentially a one way process. However, with knowledge of the formal specification describing the relationship between source and object code, source can be obtained from a compiled application [40]. But compilers introduce many code additions and optimisations, which prevent decompilation leading to identical source code [42].

Original source information is lost during processes of compiling such as translation into intermediate representations. Whilst formal specifications can be used to deduce an equivalent high-level representation from machine code, it will not necessarily correspond to the original source code (although functionality should remain the same). This is exemplified in figure 2. Information not pertinent to operation and not retained in some representation, for example, comments are irretrievable.

Decompilation is most successful when there is substantial debugging information and metadata, such as with Java bytecode [43].

IDAPro³ and Boomerang⁴ are prominent examples of disassemblers and decompilers.

³<http://www.hex-rays.com/idapro/>

⁴<http://boomerang.sourceforge.net/>

C. Debuggers

Debugging tools are highly useful to assist developers in understanding why code behaves the way it does. Program or code slicing is a common technique of static and dynamic debuggers to isolate key components of an application for study. For the reverse engineer, binary debuggers can trace procedures, registers, API calls and other information as well as manipulation of machine code operations and memory at runtime [44] and has proven effective at confirming architectural patterns from running software [45].

Static debuggers (that analyse code rather than running programs) have already proven useful in increasing program comprehension [46] [47].

Common examples of these tools are gdb⁵ and OllyDbg⁶ as well as system calls such as the Unix ptrace.

Both malicious and legitimate developers have been known to prevent specific debuggers from effectively working against applications by exploiting flaws in the analysis software [48].

This class of tools also contains hexeditors and API monitors, that allow observation and modification of binary applications.

D. Code Analysis Tools

If source code is available it is often easy to reveal internal logic, at a higher level, design specifications such as UML diagrams can be created through the use of common development tools and is a common feature of modern IDEs [49]. These reverse engineering tools are differentiated from traditional static analysis tools through their ability to identify high-level

⁵<http://www.gnu.org/software/gdb/>

⁶<http://www.ollydbg.de/>

structural information of a program, as well as specific code-level flaws. Tools such as lint⁷, its derivatives and Fortify⁸ can yield information on potential vulnerabilities in a system.

IV. LEGAL ISSUES

The legal ramifications of reverse engineering in industry are a crucial consideration for any attempt at determining the future popularity and development of techniques. Worldwide, laws on intellectual property with regards to software is inconsistent, misunderstood and often analysed on a case-by-case basis. In the United States, copyright protection exists to ensure the rights of authors of creative work and software is treated as literary work. In the United Kingdom and Europe patents are used for the protection of novel and industrial processes. There has been somewhat slow progress in the development of a solid legal understanding on the rights of software developers, with few notable publications in recent years.

It has been realised that reverse engineering is not a disfavoured practice [50], but one of immense benefit in the the retrieval and discovery of system information. There is, however, significant question as to when the use of reverse engineering to utilise information on private software with commercial intent becomes theft.

The majority of EULAs prohibit the reverse engineering of products. In 2003 a US court case between Bowers and Baystate Technologies found Bowers liable for copyright infringement for reverse engineering which was forbidden by relevant EULA documents. Bowers had released a product to compete with Baystate that incorporated features of the original.

Samuelson [51] discusses the legality of reverse engineering in the United States and observes that reversing is socially and economically beneficial. Samuelson points out, with the aid of McGowan [52] that "reverse engineering furthers copyright's goal of promoting the dissemination and improvement of intellectual property [and] reverse engineering does not deprive authors of returns necessary to induce investment".

To assess the legal issues, we must consider generalised scenarios for reverse engineering:

- Owner increasing understanding (recovery)
- Non-owner increasing understanding (innovation)
- Copycat engineering
- Malicious developer subverting software (licence piracy)

Of these, the first is the only clearly legal scenario - the owner of the intellectual property rights working with the software as they wish. Reverse engineering to understand how a product works and facilitate innovation sounds reasonable, after all, patents are driven by a desire to share and disseminate ideas (patents may effectively render reverse engineering unnecessary). However, it's easy to understand financially motivated developers to be concerned about losing

their edge over competitors. Which is what makes the copycat and malicious developer scenarios unconscionable.

Ohly surmises the issue well with a paper titled "Unfair Competition or Catalyst for Innovation?" [53], concluding that reverse engineering is not unfair (the United States recognises it as an acceptable way to gain information), but that "honest practices" must be observed leading to the redefined problem of identifying "unfair copying".

As Kienle et al [54] have gone on to observe; whilst the various international laws may permit reverse engineering extensively, there is a point before illegality at which ethical and moral concerns should become apparent.

V. CURRENT AND EMERGING WORK

Reverse engineering may occur in multiple contexts, providing the engineer with a different level of system description depending on the focus of their work. This paper identifies two broad categories of reverse engineering - code analysis and system analysis. One does not preclude the other, but typically the objectives of one are the requirements of the other. Both categories of analysis can yield the same results depending on technique; what distinguishes them is the source information used for analysis (source code or data from the system).

A number of significant projects with a variety of conditions and objectives from the past decade are presented here to introduce the reader to the wide variety of research and applications.

A. Data/System Analysis

Data/system analysis tasks, as presented here, is concerned the specification of a system from information available to the end-user as well as information exchange between systems, which includes compiled binaries, data files and in-memory information.

1) *Circumvention of Intrusion Detection Systems (IDS)*: through reverse engineering of network signatures has been documented by Mutz et al [55]. IDSs use signatures to detect network traffic that corresponds to known attacks and then take appropriate measures. If signatures are known, attacks can be crafted so as to go undetected by these systems. In an effort to maintain security and prevent workarounds, manufacturers do not disclose the signatures used by their systems, but this has been demonstrated to be ineffective.

First working with an open-source IDS (Snort), Mutz et al examined rulesets inside the software to determine exactly what characteristics of an attack were triggering suspicion. It was discovered that the software matched specific strings used in proof of concepts as payload for a specific vulnerability; by modifying this string it was simple for the team to execute attacks that evaded detection by the IDS.

A closed-source, commercial product (RealSecure) was also experimented with and required complex reverse engineering

⁷[http://en.wikipedia.org/wiki/Lint_\(software\)](http://en.wikipedia.org/wiki/Lint_(software))

⁸<https://www.fortify.com/>

for results. As rule information was not so readily available, the team investigated the triggers of particular alerts by generating a spread of network traffic targeting vulnerabilities whilst performing a dynamic trace of the binary application as input was received. Tracing was performed using ptrace system calls in Unix. Whilst dynamic tracing generated significant amounts of irrelevant data, after experimentation with filtering, meaningful information could be found. State machine diagrams documenting triggers were generated, enabling the team to design non-suspicious payloads.

Using these techniques the team was able to design evasion techniques for specific attacks where a specific IDS is present. The paper highlights the ease with which specific program processes can be understood and operating knowledge derived. Further research into such techniques would encourage improvement of such systems and likely drive manufacturers to develop more generalised heuristic approaches.

2) *GUI Ripping*: is a term described by Memon et al [56] to define reverse engineering techniques for automated graphical user interface traversal and testing. Whilst GUIs are an invaluable asset for the majority of home and office system applications, currently available techniques for generating use cases are limited and require either human guidance or development overhead through manual model creation.

The GUI Ripping method seeks to improve upon the limitations of traditional techniques by reverse engineering a graphical application as it executes. The method takes advantage of the hierarchical nature of GUI to automatically generate a model of the containing learned information on structure and execution flow behaviour by traversing the structure of the GUI with a depth first approach. This model representation can then be passed to subsequent tools for automated generation of test cases.

The work demonstrates the wide variety and innovative applications of reversing, particularly with regards to those seeking to enhance their own software products.

The GUI Ripping investigation was published in 2003, a significantly more recent project is the GUI Surfer by Silva et al [57]. Whilst the objective of creating a model of the user interface remains the same, the methodology is entirely different and is based upon the analysis of source code rather than a running application.

Code slicing, a recurring technique in reverse engineering, is used to isolate pertinent aspects (graphical widgets), form a graph representation and create a GUI layer leading to a language independent GUI behaviour model. With this information state machines representing execution flow can be created, which have numerous benefits including the ability to recompute a machine with minimum states for refactoring, detect unreachable states (ie. dead code), metrics formation and the generation of test cases.

With seven years between the two publications it is encouraging to see significant improvements in the approach. Silva et al are more innovative in their applications than Memon

et al as they have generated a more effective model, with an effective analysis of states to test the GUI. Continued efforts in this area would be useful in the development of functional testing suites.

3) *Security testing of web applications*: has been carried out by Huang et al [58] using reverse engineering to detect data entry points and monitoring fault injection. Pages of a web application were crawled and where input possible, simulated user events would be applied with the change in dynamic components monitored.

The primary use of reverse engineering was to identify components of possible vulnerability prior to attack, which include targets for cross-site scripting and client side Javascript. The crawling mechanism can be likened to the previously referenced GUI Ripping method. This black box crawling approach is an effective technique given the unavailability (remotely hosted) of web application code and binaries.

Related work investigating web application reverse engineering has been carried out by a number of other teams, for instance, Lucca et al [29] utilise reverse engineering for maintenance and evolution of web applications. There is growing interest in reverse engineering for the examination of web applications, perhaps caused by the relatively economical and rapid deployment of web services that drives a highly competitive market. Whilst the unavailability of code and binary data is a significant limit factor in the dissection of web applications, the expansion of techniques such as this could lead to adequately useful high-level design reversal in the future.

4) *Undocumented file format*: structure and behaviour recovery is a common problem for developers tackled with reverse engineering. Conti et al [59] have approached the problem along with a novel technique for visualisation of information, enabling engineers to perform a visual interpretation of binary data - detecting patterns of specific regularity, irregularity and recurrence (such as steganographic messages hidden within binary files, or format structure).

This new tool utilises the human capacity for visual pattern recognition and is beneficial through its ability to present significantly more data per region of visible screen space than conventional alternatives such as a hexeditor. However, the tool is intended to be a complement to traditional tools, by providing new alternative high-level visualisations of generic binary data.

Whilst the tool is a novel addition to the field; there are some questions over the efficacy. Whilst the information displayed is certainly characteristic of specific structures of data, for the most part this can be deduced by automated tools and does not require human intervention (and consequently, visualisation for analysis).

As open source initiatives are on the increase, we can expect to see less use of proprietary formats and a decrease in the need for reversal of their structure. It is in the interest of developers, even those using proprietary formats, to ensure data is portable.

However, the analysis of files in this manner is beneficial to uncovering hidden data in typical files.

5) *Protocol reverse engineering*: involves extraction of application-level protocol information without an available specification. Cabellero et al [60] have devised and tested a framework for this task that dramatically extends traditional techniques of packet analysis by leveraging information available from the binaries responsible for handling protocol information. This process is referred to as 'shadowing'.

Testing was carried out through the reverse engineering of DNS, HTTP, IRC, ICQ and Samba protocols, all of which are highly popular and well specified. Minimal differences were observed, with differences in handling field information being the reason for any differences.

As with typical approaches, there still exists the limitation that events that do not occur during the reverse engineering process cannot be analysed and are therefore not known in the derived specification. However, the analysis of the program binary leads to more accurate protocol information.

As new network applications are created all the time, it is desirable for many developers to understand how to communicate with them. As the sophistication of protocol analysis increase, we can hope to see a trend towards automisation of the process.

6) *Source code recovery*: is highly desirable in industry for the maintenance of legacy systems, retrieval of lost algorithms but also for the illegitimate replication of software. Emmerick and Waddington [61] worked to recover algorithms from a prototype application written in Visual C++.

The application consisted of multiple compiled executable and library files, some of which were math intensive and representing significant value, hence the requirement for recovery. Source code for an earlier version of the source code was available, though not sufficient for the client's needs.

Decompilers have many practical shortcomings due to the partially irreversible compilation process. The most effective decompilers operate on byte code intended for virtual machines (such as Java and MS CLI languages) due to the extensive metadata that can be commandeered for recovery purposes. The team chose to use the experimental, open-source Boomerang decompiler. The binary to be decompiled was significantly larger than the test cases Boomerang had been used with previously and the commercial IDA Pro was used to make up for shortcomings.

The experimenters were able to select specific portions of the binary files in order to focus on the data of client interest. Boomerang enabled recovery of class names and hierarchy as well as source code fragments. Sufficient code was recovered to allow functional testing of procedures and the available source code from a previous prototype was used as a reference for the decompiled code. Significant algorithm outlines were recovered and differences were minor, such as the use of C *malloc()* calls vs C++ *new()* calls.

Although the authors recognise that the workload was significant and the amount of recovery of the overall application was low, they were successful in their objective and provided a beneficial service in recovering important code. This project is an encouraging example of commercial reverse engineering applications particularly with regards to decompilation, a process which often bears little fruit. If the processes required for decompilation of larger programs can be understood and enhanced with future, similar projects then we can expect to see an increase in the effectiveness of decompiler tools.

7) *UML recovery*: is a common means for developers to create high-level specifications after the fact. Traditionally, this has been achieved through analysis of code (see section V-B1), but Briand and Leduc have implemented this through dynamic analysis [49] for the purpose of system comprehension and quality assurance. The research was specific to Java RMI with a focus on multithreading however has many translatable concepts.

The approach has a benefit with regards to investigation of quality assurance in relation to code based techniques. By generating traces during application execution a model was built and a sequence diagram generated. As the system dealt with runtime data rather than original source code, the generated diagrams could make a truer comparison with the original design diagrams to determine the presence of discrepancies in the system.

Further work has been carried out by Costagliola et al [62] and involves the use of visual language to identify design patterns. UML recovery is one of the most significant uses of software reverse engineering as it pertains to the maintenance and interoperability of legacy systems, though most work is focussed on code based reversing of UML and modern development typically necessitate good practices such as revision control, so future legacy systems will be more available we can expect to see a decline in the need for binary reversing in this fashion.

B. Code Analysis

Code analysis is the analysis of semantic information embedded within source code and can be performed when code is available to the engineer and typically involves the generation of a high-level specification of a system.

Reverse engineering of code is a less studied task, since the need for reversing tends to arise from a lack of available code, documentation and system information. Nevertheless, such analysis is common in software development as it ties in to fields such as metrics [63].

1) *UML recovery*: is a feature of many modern IDEs and involves the generation of UML diagrams (class, sequence, etc) from source code (see section V-A7 for an example of UML recovery based on runtime analysis). UML diagram generation is perhaps the most significant application of reverse engineering of available code and enhances high-level program understanding and aid quality assurance - helping to

for comparison to original formal design specifications or to ensure software will behave as expected.

Successful has been carried out on creating formal proofs derived from code by Gannod and Cheng [71] who have gone on to apply their work to a NASA command and control system. The work outlined in their publication was limited to "straight-line code" (code with no iteration or recursion); recursion is a particular complexity for formal methods.

For systems of such significance that they necessitate a formal specification, the ability to reverse engineer proofs would be of significant benefit to ensure there are no discrepancies between specification and implementation, particularly in an environment where formal annotations in code may be unavailable.

Less formal means of reverse engineering quality assurance has been investigated in the form of "code smell" detection. Typical tools look at reducing security issues, however, code smells describe general bad programming practices. Emden and Moonen [72] presented an implementation for Java, enabling automatic code inspection suitable for continuous integration. Schumacher et al [73] published more recently with a tool to generate code metrics as part of the code smell detection process.

6) *Web application understanding*: is a difficult task, as the majority of code, executables and system components are complicated through multi-layered and multi-tiered designs with a demanding market placing pressure on fast delivery over verbose design. Development for the web is certainly one of the most significant aspects of modern software engineering but one most lacking in solid documentation. Lucca et al [74] have defined an approach for this difficult task of reverse engineering application design by analysing source code.

The results were deemed adequate and effective by a team of experienced web application engineers, although some aspects required human intervention to understand some aspects.

This work can be contrasted with the work presented in section V-A3 relating to black box security testing of web applications which is carried out from the client side, oblivious to code.

C. Counter Measures

In scenarios where the protection of copyright theft is a particular concern (ie. for commercially released products), original developers can implement techniques to mitigate reverse engineering attacks on their software to protect trade secrets or protect against piracy. Naumovich and Memon [75] and similarly, Collberg and Thomborson [76] investigated the area extensively to isolate the following techniques used to prevent software piracy:

- **Application Servers** - code is executed on a trusted server, remote to the user. Potential issues of scalability, performance and network requirements.
- **Licence Keys** - a very popular technique that is heavily invested in, however reverse engineering is trite and key

generation or overriding of key-checking (aka cracking, nopping) is common.

- **Hardware Solutions** - a once popular but increasingly less common solution, a typical example is the requirement of an authentic CD when software is in use. As with licence keys, software cracking is a common way of preventing checking mechanisms.
- **Tamperproofing** - to prevent modification of software code various techniques such as checksums, aging, cryptography and guards [77] can be employed simultaneously, to varying degrees of success.
- **Obfuscation** - renders code incomprehensible, but can be easy to reverse.
- **Watermarking** - the embedding of ownership marks into software. Not preventative, but does allow for identification.
- **TCPA** - the Trusted Computing Platform Alliance is a collaboration of technology manufacturers to provide a hardware and OS based trust system, allowing public key authentication of computer systems. This endeavor is only just emerging and is currently a long way off due to technical and logistic problems.

With the release of new commercial software, reverse engineering and piracy counter measures become more sophisticated and effective. However, the collaboration of skilled individuals across the Internet, combined with the relative worth of such products makes the effectiveness of such methods short lived and invariably worked around.

There will always be those seeking new and more effective ways to reverse engineer software as well as those seeking to prevent it.

VI. THE FUTURE

Web of Knowledge yielded 50 results for an in-title search of "reverse engineering" and topic of "software" in the last four years. This number may seem low, but we must consider the specificity of the search, many reversing techniques are developed tacitly as a part of larger projects and the area is not usually under dedicated study. The removal of the in-title parameter with topic and addition of keyword criteria for "reverse engineering" produces hundreds of results, demonstrating significant usage of techniques, if not dedicated research.

As a science, reverse engineering is developing slowly but surely. Publications can provide an indication of productivity, collaboration and progress in a research field. In a meta-study by Hassan and Holt [78] a small-world network of authors from the Working Conference on Reverse Engineering (WCRE) was observed. The active researchers are a relatively small group compared to other areas of software engineering. This suggests a lack of fresh thought to the field and perhaps goes some way to explain why there is less significant progress over recent years, however, the meta-survey notes evolution in emerging terms.

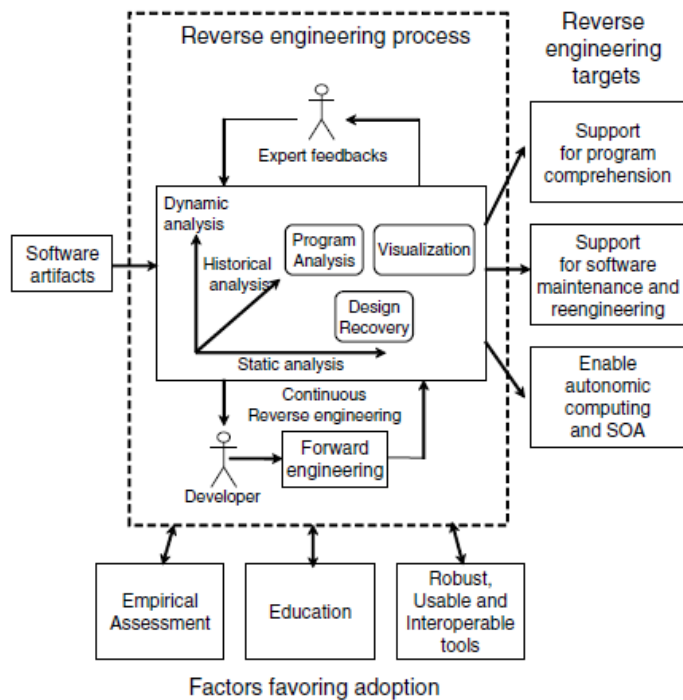


Figure 4. The role of reverse engineering Source: "New Frontiers of Reverse Engineering" [79]

Canfora et al [79] considered the roles that reverse engineering might have in the future. Figure 4 presents the targets and roles of reverse engineering as the authors imagine will come to form in the future. One of the most significant suggestions is that reverse engineering techniques will be expanded to involve not only code analysis and data/live system analysis but also historical artifact evolution. The dynamic nature of modern software will require reverse engineering to consider the history of software as well. The authors also envision that in addition to traditional goals, reverse engineering will be incorporated in the development of autonomic systems for the purpose of service discovery and reconfiguration.

Hall [80] proposes that reverse engineering must develop further beyond low-level analysis and contribute re-engineering of an application domain. This has been done to some extent with high-level design recovery but not for enterprise and distributed systems, which are increasingly popular in the form of web applications.

The methods and techniques of reverse engineering are powerful weapons for difficult system maintenance and its friendly uses should be promoted, but we must be judicious in our usage to prevent them becoming burdensome [81].

The majority of major recent works seemed to be focussed on the automation and usability issues of reverse engineering tools [82]–[84], which is an encouraging sign for the future of practical tools as whilst many tools provide usual functionality, they are generally too disparate and offer limited value in isolation. Placing the power of reverse engineering into the hands of the common developer would facilitate comprehension amongst teams and provide more reliable quality assurance.

On the other hand, there are those who must work to prevent the reverse engineering of applications and as tools become more advanced we will continue to see the development of counter measures to reverse engineering techniques.

Tonella et al [85] studied the trends in software engineering with suggestions for research directions that might improve the field. These include the refinement of a taxonomy to build knowledge, a framework for empirical studies and a means of benchmarking. The authors contend that empirical research thus far has been limited by inconsistency and mixed understandings of the field, sentiments shared by Canfora et al [79].

One area unfortunately lacking in results is disassembly. The compilation of source code is, in many cases, a one way process which makes adequate recovery of source code difficult to achieve. Nevertheless, the literature has shown that disassemblers can provide valuable industrial services so it is possible and desirable.

Reverse engineering of applications where documentation and source code is unavailable can be particularly challenging, and is a barrier to some practical uses. Whilst it is typically the nature of programming for the end-result to be designated for machine execution only, there are certainly a large number of legitimate and illegitimate users who would appreciate the ability to effectively reverse engineer compiled code. We can expect to see continued attempts at enabling this.

VII. CONCLUSIONS

Reverse engineering concerns the discovery and understanding of a system beyond what is available on the surface.

A variety of literature has been presented and a number of issues and benefits presented. Through the various references we can see a blurred line between reverse engineering and conventional engineering, this highlights the ubiquitous and flexible nature of reversing. In this paper the topic has been recognised as present in all aspects of computer science to varying degrees, hopefully the reader has come to share this view point.

Reverse engineering is a substantial area for research and practical use. Whilst there will always be some contention regarding the use of reverse engineering in relation to copyright and IP theft, techniques are employed to varying extents by most developers and are particularly beneficial for software maintenance.

Reversing will surely continue to be utilised by unscrupulous programmers, and as frameworks and tools are developed to make the task of discovering the functionality and processes of a system, we could expect to see a rise in malicious use. This is the state of any useful tool however, and the same tool will provide us with an ability to overcome the abuse. Understanding how reverse engineers can gather information on an application is important for protecting commercial property from software piracy.

Ali [86] observes an unfortunate lack of interest in reverse engineering despite a rise in need for software maintainers and encourages that undergraduate students study the field. Computer science students are often given assignments with an objective to write some small piece of software; a more realistic assignment is the modification of an existing system a task requiring increased understanding and adaptability but is seldom featured in class activities.

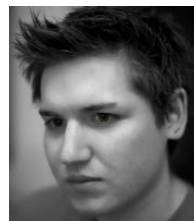
It would be desirable to see an increase in research of reverse engineering as it applies to the common engineer. This would encourage increased usage and improve awareness of the potential of reversing, driving fresh innovation and development in the field.

REFERENCES

- [1] M. G. Rehoff, "On Reverse Engineering," in *IEEE Transactions on Systems, Man, and Cybernetics*, 1985, pp. 244–252. I
- [2] E. J. Chikofsky and J. H. C. II, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, pp. 13–17, 1990. I, II-D
- [3] M. L. Nelson, "A survey of reverse engineering and program comprehension," *CoRR*, vol. abs/cs/0503068, 2005. I
- [4] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller, "Programmable reverse engineering," *International Journal of Software Engineering and Knowledge Engineering*, vol. 4, pp. 501–520, 1994. II
- [5] L. Briand, Y. Labiche, and Y. Miao, "Towards the reverse engineering of uml sequence diagrams," *Reverse Engineering, Working Conference on*, vol. 0, p. 57, 2003. II
- [6] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong, "Reverse engineering: a roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 47–60. II-A
- [7] B. Price, "A Principled Taxonomy of Software Visualization," *Journal of Visual Languages & Computing*, vol. 4, no. 3, pp. 211–266, Sep. 1993. [Online]. Available: <http://dx.doi.org/10.1006/jvlc.1993.1015> II-A
- [8] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl, "A reverse-engineering approach to subsystem structure identification," *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, pp. 181–204, 1993. [Online]. Available: <http://dx.doi.org/10.1002/smr.4360050402> II-A
- [9] S. Deelstra, M. Sinnema, and J. Bosch, "Experiences in software product families: Problems and issues during product derivation," in *Software Product Lines*, ser. Lecture Notes in Computer Science, R. L. Nord, Ed. Springer Berlin / Heidelberg, 2004, vol. 3154, pp. 120–122. II-A
- [10] R. Weir, "Opendocument format: The standard for office documents," *IEEE Internet Computing*, vol. 13, pp. 83–87, March 2009. [Online]. Available: <http://dx.doi.org/10.1109/MIC.2009.42> II-A
- [11] W. Treese, "Politics and the technology of file formats," *netWorker*, vol. 10, pp. 15–17, March 2006. II-A
- [12] G. R. Ariel Futoransky and A. Waissbein, "Cryptography in forensics and reverse-engineering," *FIRST Technical Colloquium. October 6, 2005. Palacio San Martín, Buenos Aires, Argentina.*, 10 2–5. II-B
- [13] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, "Reformat: Automatic reverse engineering of encrypted messages," in *Computer Security – ESORICS 2009*, ser. Lecture Notes in Computer Science, M. Backes and P. Ning, Eds. Springer Berlin / Heidelberg, 2009, vol. 5789, pp. 200–215. II-B
- [14] L. Hively, F. Sheldon, and anna Squicciarini, "A vision for scalable trustworthy computing," *IEEE Security and Privacy*, vol. 99, no. PrePrints, 2010. II-B, II-C
- [15] B. Schneier, "Cryptographic design vulnerabilities," *Computer*, vol. 31, pp. 29–33, September 1998. [Online]. Available: <http://portal.acm.org/citation.cfm?id=619033.621071> II-B
- [16] K. Nohl, D. Evans, S. Starbug, and H. Plötz, "Reverse-engineering a cryptographic rfid tag," in *Proceedings of the 17th conference on Security symposium*. Berkeley, CA, USA: USENIX Association, 2008, pp. 185–193. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1496711.1496724> II-B
- [17] W. Diffie and M. E. Hellman, "New directions in cryptography," 1976. II-B
- [18] F. Amiel, B. Feix, and K. Villegas, "Power analysis for secret recovering and reverse engineering of public key algorithms," in *Proceedings of the 14th international conference on Selected areas in cryptography*, ser. SAC'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 110–125. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1784881.1784889> II-B
- [19] B. Guha and B. Mukherjee, "Network security via reverse engineering of tcp code: vulnerability analysis and proposed solutions," in *Proceedings of the Fifteenth annual joint conference of the IEEE computer and communications societies conference on The conference on computer communications - Volume 2*, ser. INFOCOM'96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 603–610. II-C
- [20] M. Popa, M. Macrea, and L. Mihiu, "Reverse engineering analyze for microcontrollers' assembly language projects," in *Advances in Systems, Computing Sciences and Software Engineering*, T. Sobh and K. Elleithy, Eds. Springer Netherlands, 2006, pp. 333–338. II-C
- [21] S. Sparks, S. Embleton, R. Cunningham, and C. Zou, "Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting," *Computer Security Applications Conference, Annual*, vol. 0, pp. 477–486, 2007. II-C
- [22] P. Silberman and M. Zaidman, "Paimeidiff: keeping microsoft honest on patch tuesdays," *J. Comput. Small Coll.*, vol. 22, pp. 64–64, January 2007. II-C
- [23] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," *Security and Privacy, IEEE Symposium on*, vol. 0, pp. 94–109, 2009. II-C
- [24] T. Kwon and Z. Su, "Automatic detection of unsafe component loadings," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 107–118. [Online]. Available: <http://doi.acm.org/10.1145/1831708.1831722> II-C
- [25] S. D. Webb and S. Soh, "A survey on network game cheats and p2p solutions." II-D
- [26] G. Hoglund and G. Mcgraw, *Exploiting Software : How to Break Code*. Addison-Wesley Professional, Feb. 2004. [Online]. Available: <http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0201786958> II-D
- [27] G. A. Di Lucca, A. R. Fasolino, and P. Tramontana, "Reverse engineering web applications: the ware approach," *J. Softw. Maint. Evol.*, vol. 16, pp. 71–101, January 2004. II-E
- [28] R. Patel, F. Coenen, R. Martin, L. Archer, W. C. Nx, R. Patel, F. Coenen, R. Martin, and L. Archer, "Reverse engineering of web applications: A technical review," 2007. II-E
- [29] D. L. Di, M. D. Penta, G. Antoniol, and G. Casazza, "An approach for reverse engineering of web-based applications," in *Proceedings of WCRE '01*, 2001, pp. 231–240. II-E, V-A3
- [30] S. Handa, "Reverse engineering computer programs under canadian copyright law," *McGill Law Journal*, 1995. II-F
- [31] S. McClure, J. C. Foster, and M. Price, *Sockets, Shellcode, Porting, & Coding: Reverse Engineering Exploits And Tool Coding For Security Professionals*. Syngress Publishing, 2005. II-F
- [32] A. ul Iman, M.; Ishaq, "Anti-reversing as a tool to protect intellectual property," *Engineering Systems Management and Its Applications (ICESMA), 2010 Second International Conference*, 2010. II-F

- [33] K. Eschenfelder and A. Desai, "Software as protest: The unexpected resiliency of US-based DeCSS posting and linking," *INFORMATION SOCIETY*, vol. 20, no. 2, pp. 101–116, APR-JUN 2004. II-F
- [34] B. Bellay and H. Gall, "An evaluation of reverse engineering tool capabilities," *Journal of Software Maintenance*, vol. 10, pp. 305–331, September 1998. [Online]. Available: <http://portal.acm.org/citation.cfm?id=294168.294262> III
- [35] —, "A comparison of four reverse engineering tools," *Reverse Engineering, Working Conference on*, vol. 0, p. 2, 1997. III
- [36] M. E. Mit and M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *In WODA 2003: ICSE Workshop on Dynamic Analysis*, 2003, pp. 24–27. III
- [37] S. DeRosa, "Reverse engineering malware," *University of Advancing Technology*. III
- [38] E. Buss, R. D. Mori, R. De, M. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, E. Merlo, H. Müller, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S. Tilley, J. Troster, and K. Wong, "Investigating reverse engineering technologies: The cas program understanding project," *IBM Systems Journal*, vol. 33, pp. 477–500, 1994. III
- [39] R. Leigland and A. W. Krings, "A formalization of digital forensics," *International Journal of Digital Evidence*, vol. 3, 2004. III-A
- [40] P. T. Breuer and J. P. Bowen, "Generating decompilers," in *Information and Software Technology Journal*. John Wiley & Sons, 1998, pp. 131–138. III-B
- [41] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *In Proc. IEEE 2002 Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 2002, pp. 45–54. III-B
- [42] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," Australia, Tech. Rep., 1994. III-B
- [43] J. Miecznikowski and L. J. Hendren, "Decompiling java bytecode: Problems, traps and pitfalls," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC '02. London, UK: Springer-Verlag, 2002, pp. 111–127. [Online]. Available: <http://portal.acm.org/citation.cfm?id=647478.727938> III-B
- [44] J. Vanegue, T. Garnier, J. Auto, S. Roy, and R. Lesiank, "Next generation debuggers for reverse engineering," *4th Annual Hackers To Hackers Conference*. III-C
- [45] T. Systä, "Dynamic reverse engineering of java software," in *Proceedings of the Workshop on Object-Oriented Technology*. London, UK: Springer-Verlag, 1999, pp. 174–175. [Online]. Available: <http://portal.acm.org/citation.cfm?id=646779.705453> III-C
- [46] R. Kollman, P. Selonen, E. Stroulia, T. Systä, and A. Zundorf, "A study on the current state of the art in tool-supported uml-based static reverse engineering," in *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 22–. [Online]. Available: <http://portal.acm.org/citation.cfm?id=882506.885156> III-C
- [47] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from java source code," *Automated Software Engineering, International Conference on*, vol. 0, pp. 123–134, 2006. III-C, V-B1
- [48] P. F. (Microsoft), "Anti-unpacker tricks," *Virus Bulletin, Technical Features*, June 2009. III-C
- [49] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the reverse engineering of uml sequence diagrams for distributed java software," *IEEE Transactions on Software Engineering*, vol. 32, pp. 642–663, 2006. III-D, V-A7
- [50] T. Lewis, "Reverse engineering of software: An assessment of the legality of intermediate copying," *Los Angeles Entertainment Law Review*, 2000. IV
- [51] P. Samuelson, "Reverse engineering under siege," *Commun. ACM*, vol. 45, pp. 15–20, October 2002. [Online]. Available: <http://doi.acm.org/10.1145/570907.570919> IV
- [52] D. McGowan, "Contracting, fair competition, and article 2b: Some reflections on federal competition policy," vol. 13, 1998. IV
- [53] A. Ohly, "Reverse engineering: Unfair competition or catalyst for innovation?" in *Patents and Technological Progress in a Globalized World*, ser. MPI Studies on Intellectual Property, Competition and Tax Law, J. Drexl, R. M. Hilty, W. Schön, J. Straus, W. P. z. W. u. Pyrmont, M. J. Adelman, R. Brauneis, J. Drexl, and R. Nack, Eds. Springer Berlin Heidelberg, 2009, vol. 6, pp. 535–552. IV
- [54] H. M. Kienle, D. German, and H. Muller, "Legal concerns of web site reverse engineering," in *Proceedings of the Web Site Evolution, Sixth IEEE International Workshop*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 41–50. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1025133.1026456> IV
- [55] D. Mutz, C. Kruegel, W. Robertson, G. Vigna, and R. A. Kemmerer, "Reverse engineering of network signatures," in *IN PROCEEDINGS OF THE AUSCERT ASIA PACIFIC INFORMATION TECHNOLOGY SECURITY CONFERENCE, GOLD*, 2005, pp. 1–86 499. V-A1
- [56] A. Memon, I. Banerjee, and A. Nagarajan, "Gui ripping: Reverse engineering of graphical user interfaces for testing," *Reverse Engineering, Working Conference on*, vol. 0, p. 260, 2003. V-A2
- [57] J. C. Silva, C. Silva, R. Goncalo, J. Saraiva, and J. C. Campos, "The GUISurfer Tool: Towards a Language Independent Approach to Reverse Engineering GUI Code," in *EICS 2010: PROCEEDINGS OF THE 2010 ACM SIGCHI SYMPOSIUM ON ENGINEERING INTERACTIVE COMPUTING SYSTEMS*, ACM SIGCHI. 1515 BROADWAY, NEW YORK, NY 10036-9998 USA: ASSOC COMPUTING MACHINERY, 2010, Proceedings Paper, pp. 181–186, 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems, Berlin, GERMANY, JUN 19-23, 2010. V-A2
- [58] Y.-W. Huang, C.-H. Tsai, T.-P. Lin, S.-K. Huang, D. Lee, and S.-Y. Kuo, "A testing framework for web application security assessment," *Computer Networks*, vol. 48, no. 5, pp. 739 – 761, 2005, web Security. V-A3
- [59] G. Conti, E. Dean, M. Sinda, and B. Sangster, "Visual reverse engineering of binary and data files," in *Visualization for Computer Security*, ser. Lecture Notes in Computer Science, J. Goodall, G. Conti, and K.-L. Ma, Eds. Springer Berlin / Heidelberg, 2008, vol. 5210, pp. 1–17. V-A4
- [60] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 317–329. V-A5
- [61] M. Van Emmerik and T. Waddington, "Using a decompiler for real-world source recovery," in *Proceedings of the 11th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 27–36. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1038267.1039035> V-A6
- [62] G. Costagliola, A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery by visual language parsing," *Software Maintenance and Reengineering, European Conference on*, vol. 0, pp. 102–111, 2005. V-A7
- [63] S. Demeyer, S. Ducasse, and M. Lanza, "A hybrid reverse engineering approach combining metrics and program visualization," in *Proceedings of the Sixth Working Conference on Reverse Engineering*, ser. WCRE '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 175–. [Online]. Available: <http://portal.acm.org/citation.cfm?id=832306.837044> V-B
- [64] R. Naik and A. Bahulkar, "A programmable analysis and transformation framework for reverse engineering," *Electronic Notes in Theoretical Computer Science*, vol. 94, pp. 39 – 49, 2004, proceedings of the International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM 2003). 3, V-B2
- [65] R. K. Keller, R. Schauer, S. Robitaille, and P. Pag?, "Pattern-based reverse-engineering of design components," *Software Engineering, International Conference on*, vol. 0, p. 226, 1999. V-B1
- [66] S. Matzko, P. J. Clarke, T. H. Gibbs, B. A. Malloy, J. F. Power, and R. Monahan, "Reveal: a tool to reverse engineer class diagrams," in *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, ser. CRPIT '02. Darlinghurst, Australia, Australia: Australian

- Computer Society, Inc., 2002, pp. 13–21. [Online]. Available: <http://portal.acm.org/citation.cfm?id=564092.564095> V-B1
- [67] G. Canfora, A. Cimitile, and U. de Carlini, “A logic-based approach to reverse engineering tools production,” *IEEE Trans. Softw. Eng.*, vol. 18, pp. 1053–1064, December 1992. [Online]. Available: <http://portal.acm.org/citation.cfm?id=147665.147671> V-B3
- [68] C. Dahn and S. Mancoridis, “Using program transformation to secure c programs against buffer overflows,” *Reverse Engineering, Working Conference on*, vol. 0, p. 323, 2003. V-B4
- [69] N. Jovanovic, C. Kruegel, and E. Kirda, “Pixy: A static analysis tool for detecting web application vulnerabilities (short paper),” *Security and Privacy, IEEE Symposium on*, vol. 0, pp. 258–263, 2006. V-B4
- [70] M. Zitser, R. Lippmann, and T. Leek, “Testing static analysis tools using exploitable buffer overflows from open source code,” *SIGSOFT Softw. Eng. Notes*, vol. 29, pp. 97–106, October 2004. V-B4
- [71] G. C. Gannod and B. H. C. Cheng, “Strongest postcondition semantics as the formal basis for reverse engineering,” in *Proceedings of the Second Working Conference on Reverse Engineering*, ser. WCRE '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 166–. [Online]. Available: <http://portal.acm.org/citation.cfm?id=832303.836920> V-B5
- [72] E. Van Emden and L. Moonen, “Java quality assurance by detecting code smells,” in *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 97–. [Online]. Available: <http://portal.acm.org/citation.cfm?id=882506.885134> V-B5
- [73] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, “Building empirical support for automated code smell detection,” in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10. New York, NY, USA: ACM, 2010, pp. 8:1–8:10. [Online]. Available: <http://doi.acm.org/10.1145/1852786.1852797> V-B5
- [74] G. A. Di Lucca, A. R. Fasolino, and P. Tramontana, “Reverse engineering web applications: the ware approach,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 1-2, pp. 71–101, 2004. [Online]. Available: <http://dx.doi.org/10.1002/smr.281> V-B6
- [75] G. Naumovich and N. Memon, “Preventing piracy, reverse engineering, and tampering,” *Computer*, vol. 36, pp. 64–71, July 2003. V-C
- [76] C. S. Collberg, I. C. Society, C. Thomborson, and S. Member, “Watermarking, tamper-proofing, and obfuscation - tools for software protection,” *Software Engineering, IEEE Transactions on*, vol. 28, pp. 735–746, 2002. V-C
- [77] H. Chang and M. J. Atallah, “Protecting software code by guards,” in *Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*, ser. DRM '01. London, UK, UK: Springer-Verlag, 2002, pp. 160–175. [Online]. Available: <http://portal.acm.org/citation.cfm?id=647777.734775> V-C
- [78] A. E. Hassan and R. C. Holt, “The small world of software reverse engineering,” in *Proceedings of the 11th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 278–283. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1038267.1039059> VI
- [79] G. Canfora Harman and M. Di Penta, “New Frontiers of Reverse Engineering,” in *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 326–341. [Online]. Available: <http://dx.doi.org/http://dx.doi.org/10.1109/FOSE.2007.154>, VI
- [80] P. Hall, “Overview of reverse engineering and reuse research,” *Information and Software Technology*, vol. 34, no. 4, pp. 239 – 249, 1992. VI
- [81] I. D. Baxter and M. Mehlich, “Reverse engineering is reverse forward engineering,” *Reverse Engineering, Working Conference on*, vol. 0, p. 104, 1997. VI
- [82] H. M. Kienle and H. A. Mueller, “The Tools Perspective on Software Reverse Engineering: Requirements, Construction, and Evaluation,” in *ADVANCES IN COMPUTERS, VOL 79*, ser. Advances In Computers. 525 B STREET, SUITE 1900, SAN DIEGO, CA 92101-4495 USA: ELSEVIER ACADEMIC PRESS INC, 2010, vol. 79, pp. 189–290. VI
- [83] H. Lee, H. Youn, and E. Lee, “Automatic detection of design pattern for reverse engineering,” in *SERA 2007: 5th ACIS International Conference on Software Engineering Research, Management, and Applications, Proceedings*, Kim, HK and Tanaka, J and Malloy, B and Lee, R and Wu, C and Baik, DK, Ed., Int Assoc Comp & Informat Sci; IEEE Comp Soc; IEEE. 10662 LOS VAQUEROS CIRCLE, PO BOX 3014, LOS ALAMITOS, CA 90720-1264 USA: IEEE COMPUTER SOC, 2007, Proceedings Paper, pp. 577–583, 5th ACIS International Conference on Software Engineering Reseach, Management and Applications held in Conjunction with 1st International Workshop on Advanced Internet Technology and Applications, Busan, SOUTH KOREA, AUG 20-22, 2007. VI
- [84] P. F. Mihancea, “Towards a Reverse Engineering Dataflow Analysis Framework for Java and C plus,” in *PROCEEDINGS OF THE 10TH INTERNATIONAL SYMPOSIUM ON SYMBOLIC AND NUMERIC ALGORITHMS FOR SCIENTIFIC COMPUTING*. 10662 LOS VAQUEROS CIRCLE, PO BOX 3014, LOS ALAMITOS, CA 90720-1264 USA: IEEE COMPUTER SOC, 2009, Proceedings Paper, pp. 285–288, 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, ROMANIA, SEP 26-29, 2008. VI
- [85] P. Tonella, M. Torchiano, B. Du Bois, and T. Systä, “Empirical studies in reverse engineering: state of the art and future trends,” *Empirical Software Engineering*, vol. 12, pp. 551–571, 2007. VI
- [86] M. R. Ali, “Why teach reverse engineering?” *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1–4, July 2005. VII



Jonathan Francis Roscoe is a Software Engineering MEng student at Aberystwyth University, Wales.

A firm believer in the importance of well engineered systems; academic interests include architectural patterns, development methodologies and software vulnerabilities.

He wasn't planning to add a bibliography to this paper until he saw guidelines for bibliographies in the IEEE Transactions style manual.