Reverse Code Engineering – State of the Art and Countermeasures

Reverse Code Engineering – Stand der Technik und aktuelle Gegenmaßnahmen

Carsten Willems, Ruhr-University Bochum,
Felix C. Freiling, Friedrich-Alexander University Erlangen-Nuremberg

Summary Reverse Code Engineering (RCE) is, loosely speaking, the process of analyzing a piece of code in order to understand it. RCE is often used to analyze proprietary, binary programs, and in the last few years this research area has evolved a lot. In this article, we survey and structure the area of reverse code engineering. We focus on different techniques to recover both the control and data flow of a given binary program, for which no source code is available. Furthermore, we also discuss analysis techniques for malicious software (short: malware), which is commonly protected to resist analysis. We present the current state of the art of such protection techniques, while dividing them into active and passive measures. Our survey focusses on reverse engineering of binary native code for the Intel/AMD x86 architecture, and we thus disregard analysis of byte-code like Java or .NET. Nevertheless, most of the techniques presented in this article can be transferred to other architectures and operating system as well. ►►►
Zusammenfassung Reverse Code Engineering (RCE) ist die Analyse von Binärprogrammen mit dem Ziel, deren Semantik zu verstehen. Traditionell wird dabei vor allem proprietäre Software untersucht, für die kein Sourcecode verfügbar ist. In letzter Zeit hat es jedoch eine enorme Ausweitung auch auf andere Einsatzgebiete gegeben. In diesem Bericht werden die verschiedenen Bereiche und Einsatzgebiete vorgestellt und eine Strukturierung vorgenommen. Dabei wird im ersten Teil auf die verschiedenen Methoden zur Rückgewinnung von Kontroll- und Datenfluss unbekannter Software eingegangen. Im zweiten Teil werden ausführlich die verschiedenen Schutzmaßnahmen behandelt, die von Programmen eingesetzt werden, um sich einer solchen Analyse zu entziehen. Der Fokus liegt dabei in der Analyse von Binärcode für die Intel/AMD x86 Architektur. Daher wird das Gebiet der heutigen Bytecode-Sprachen wie Java or .NET vernachlässigt. Die vorgestellten Methoden und Verfahren lassen sich jedoch problemlos auch auf andere Hardware- und Software-Architekturen übertragen.

Keywords D [Software]; reverse code engineering, program analysis, software protection ►►► **Schlagwörter** RCE, Programm Analyse, Schutz der Software

1 Introduction

The term *engineering* generally refers to a constructive process, in which hardware/software systems are built and brought into operation for commercial or private purposes. Many such systems exist today for which we do not have access to the details of their construction process, for example because of lost documentation or legal requirements (intellectual property protection). This has promoted the advent of a corresponding *deconstructive* process commonly known as *Reverse Engineering* (RE) or simply *reversing*. In general, reverse engineering refers to the process of "analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction" [16]. While the term originally was introduced with respect to hardware devices and machines, nowadays it usually relates to *reverse code engineering* or *protocol/fileformat reversing* [12; 22; 25; 26; 44; 77; 85; 88]. Protocol reverse engineering tries to recover the structure of a secret network protocol. Similarly, fileformat reverse engineering tries to reconstruct non public file specifications from raw data files.

In the past, RE has often been associated with shady and illegitimate uses in the context of software piracy. This bad image is probably due to public advertising and legal campaigns of large companies trying to protect their intellectual property. Today, there are an increasing number of totally legitimate applications of RE, such as closed source auditing, vulnerability research and the analysis of malware. Famous examples for successful applications of RE include the recovery and recreation of the original IBM BIOS, which empowered the enormous success of the IBM PC compatible computers, the reversing of the Microsoft SMB/CIFS protocol which lead to the development of *Samba* [77], the disclosure of the Sony Rootkit by *Russinovich* [66], and the recent reversing of the *Stuxnet* worm [32].

These successes have given rise to an active academic research community that, however, is still conceptually dispersed and therefore hard to enter. In this article, we attempt to lower the entrance barrier to RE research by surveying and structuring this exciting research area. We focus on techniques to recover the control and data flow of programs for which no source code is available and which, in the case of malware, often are further protected to resist the analysis. More specifically, we concentrate our survey around RE of binary *native* code for the Intel/AMD x86 architecture. We therefore disregard RE of byte-code like Java or .NET. Nevertheless, most of the techniques discussed in the following can be transferred to other architectures and operating system as well. We also disregard the legal aspect of RE, a research area of its own.

In the following, we put RE into the broader context of *program analysis* in Sect. 2. We then cover the different RE approaches and techniques in Sect. 3 and, in Sect. 4, take a detailed look at the countermeasures against RE taken by protected software.

2 Brief Introduction to Program Analysis

Reverse Engineering is strongly related to *program analysis* [55], which tries to statically predict approximations about the run-time behavior of a program or its reachable states. One of its requirements is computability and, hence, often under- or over-approximations are used. It is mainly used to support compiler optimization, perform automated program verification and assist in security research. Its main fields are *control flow analysis*, *data flow analysis*, and *type recovery*.

When observing the control flow, it is examined which code can be reached from or leads to which other parts of the program. In data flow analysis one is interested in the dependency and relationship between different data structures. Often the semantics of the used operations on the data are not considered and only data propagation is observed. On machine level the CPU only operates on integer and float values. Therefore, all complex data structures have to be mapped onto those atomic data types during compilation. Recovering the original data structures is essential for understanding the analyzed program and, hence, performed during data type recovery. The data type for memory access has to be identified, e. g., by observing the access patterns [2; 71], inferring from the prototypes of called functions [43], or by applying unsupervised machine learning methods on memory images of running processes [24].

An essential model used in program analysis are *Control Flow Graphs* (CFG). Their nodes consist of *basic blocks*, which constitute sequentially executed instructions and have exactly one single entry and one exit node. The connecting edges represent the control flow between them and, thus, two nodes are connected if there is a branch from one to the other. Besides the CFG also *Call Graphs* (CG) are used, in which the nodes represent different functions of an application and the edges the calls between them. Constructing such graphs is a non-trivial task for executables that are only available in *binary* form. In a first step the program has to be disassembled, i. e., the particular machine instructions and encodings have to be resolved. This is exceedingly complicated on CISC architectures, since those often use a variable length instruction set and the instruction boundaries have to be determined first. In a second step the function boundaries have to be identified, which is impeded by overlapping and cross-jumping functions as well as by optimization techniques that improve computing performance by selectively placing frequently used function chunks near to each other. A further problem are indirect branches, since their effective destination is not known. Data flow techniques can determine their possible destinations, e. g., *Value Set Analysis* (VSA) [2] can be applied to statically compute the set of possible values that a certain memory location or register may contain during runtime.

Another related structure is the *Program Dependence Graph* (PDG), which comes in different forms, e. g., *Code Dependence Graph* (CDG) or *Data Dependence Graph* (DDG). CDGs and DDGs are used for mapping numerous program analysis problems to the realm of graph theory. One powerful related method is *slicing* [8; 41; 86], in which program complexity is reduced by eliminating all instructions that do not affect the reaching of a particular program state. Modern slicing techniques are based on PDGs and constitute a reachability problem. In *backward slicing* one wants to know what variables or operations influence the value of a variable at one given statement. In *forward slicing* it is verified which variables or statements are affected by a variable value at some given point. Traditionally slicing was restricted to one-procedure programs, but newer techniques allow *inter-procedural* slicing as well [38; 41; 61]. *Differential slicing* [39] takes two different runs of the same binary into account and compares them towards their different input values. By that, the input and control flow differences can be identified that distinguish the runs. Finally, *dynamic slicing* [42] reduces the complexity by taking the concrete values from one particular run into account.

Hence, several conditional statements can be ignored and removed from the graph since the tested variable values are known.

3 State of the Art in Reverse Engineering

While fundamental limits of program analysis arise from the undecidability of the *halting problem* [78] and *Rice's theorem* [62], RE suffers from additional problems beyond that. First of all, when reversing programs, the source code normally is not available. Even if it is at hand, it cannot be trusted due to compiler optimizations [3] and, therefore, the assembly output has to be taken into account. Typical difficulties that complicate RE of binaries are:

- compilation imposes information loss,
- lack of symbol names and comments,
- compiler optimizations complicate resulting code,
- hardware knowledge is necessary to understand code,
- data structures have become chunks of bytes, and
- programs try to protect from analysis.

In general there are two different approaches: *static* and *dynamic analysis*. *Static analysis* tries to model all possible program behaviors and, hence, operates on a high level of abstraction. For that purpose the program under observation is not executed, but all examinations are performed in an abstract way, mostly by using CFG, CG or PDGs. By that, the analysis is sound and covers all possible program states, while on the other hand, it is very complex and time-consuming and may encounter serious problems in the case of obfuscated and protected binaries.

In contrast to static analysis, *dynamic analysis* has its origin in program testing and profiling and takes actual data values from one or more program executions into account. Since concrete values are at hand, no abstraction and approximation has to be used. Nevertheless, no complete view of an application can be generated, since only a subset of all possible execution paths is observed. To that end, dynamic analysis is incomplete. One special form of dynamic analysis is *behavior analysis*, in which the executable is seen as a *black box* and only its runtime effects on the *environment* are observed, e. g., what files have been created or what processes have been started. Though this kind of analysis only gives a very high level view of a binary, it often is helpful and sufficient.

3.1 Static Analysis

Prior to any static analysis, the executable binary file has to be *disassembled*, e. g., the raw byte stream has to be transformed into valid assembler instructions and (unformatted) data regions. Two different disassembling algorithms are used mainly nowadays: *linear sweep* and *recursive traversal*. Linear sweep algorithms start with the first byte of the code section and consecutively analyze each successive instruction. The method is simple and fast, but it has some serious drawbacks which arise due to variable instruction size and data or garbage that is embedded into the code stream. It easily happens that this algorithm erroneously interprets data as code and, accordingly, propagates disassembling errors throughout all the following regular instructions. Recursive traversal addresses this problem by not using a strictly sequential approach, but by starting to disassemble at the known code entry-points and recursively following each branch-instruction. By that, only valid code is observed and unaligned instructions or embedded data do not disturb the process. The main disadvantage of this method is the assumption that each jump target can be identified by static analysis, which is not always possible, e. g., for indirect calls. An improvement to this algorithm is *speculative disassembly* which tries to also parse the left out gaps between the reachable code regions, using linear sweep for that. Most of the popular RE tools use recursive traversal, e. g., *IDA Pro* [69] or *OllyDbg* [89], but there also exist established tools that use linear sweep, e. g., *WinDbg* [48] or the GNU tool *objdump* [9].

Decompilation [18; 52] tries to reconstruct high level language source code from assembly. During that, all hardware related features that have been introduced during compilation have to be removed and all high level control and data structures have to be recovered. In general, this reversal process of compilation is not possible, but often usable and helpful approximations can be achieved. Problems arise through the fact that compilation is a lossy process. Not only comments and symbol names are stripped, but also optimization techniques are applied that make it impossible to reconstruct the original high level structures. If a program can be decompiled correctly, the result normally does not match the original source code. Instead, a semantically equivalent program is retrieved. *Hex-Rays* [68] is a powerful general purpose decompiler which creates C-like pseudocode as output. There also exist language-specific tools like *DeDe* [27] for *Delphi* and *VBDecompiler* [73] for *Visual Basic* programs.

One other significant problem in static analysis is the unavailability of concrete runtime memory or register values [49]. Accordingly, program analysis has to over-approximate them, which may be either unsolvable or does not reveal any useful information at all. One popular method to counter this drawback is *symbolic execution* [20], a technique originally used in program testing for generating test cases with a high code coverage. Program execution is simulated while all input variable values are substituted by symbolic ones and for each conditional branch a different execution is forked. Accordingly, all possible execution paths are traversed and for each one a symbolic representation of the path condition is created. *Klee* [13] is a popular tool for symbolic execution that is frequently used in security auditing.

3.2 Dynamic Analysis

Dynamic analysis involves the actual execution of an observed binary. For several reasons this often is not performed on a *real* machine, but with the help of an *emulator* or a *Virtual Machine* (VM). For instance in case

of malware analysis, the analyzing host may get infected with malicious content through the analysis, which is not desirable in most cases. Furthermore, the utilization of an emulated/virtualized machine offers enhanced controlling and monitoring capabilities.

The most powerful actual tool for dynamic analysis is the *debugger*. State of the art debuggers are *OllyDbg* [89], *Immunity Debugger* [37] (OllyDbg plus a powerful Python interface), *WinDbg* [48] and *gdb* [57]. In the past one very popular tool has been *SoftIce* [23], but it is no longer supported and maintained for newer operating systems and has been unofficially replaced by *Syser* [76] nowadays. A debugger places an additional interaction layer between the underlying operating system and the debugged application, the so called *debugee*. By that, it becomes possible to pause and resume the monitored process, to single step through its code, or to view and modify the CPU state and the memory content. For that purpose a debugger application either utilizes special debugging APIs offered by the operating system or it uses custom methods to control execution of the debugee and access its runtime context appropriately. Depending on how deep the debugger is integrated into the operating system, it either allows to debug usermode processes only or it also enables to debug kernel-mode applications and the operating system itself. In the latter case normally a *remote debugging session* is used, in which two different hosts are required: the debugged one, which requires special OS extensions, and the debugging one that executes the debugger.

Normally debugging is an interactive process and, hence, debuggers offer an integrated disassembler to display the instructions at the current instruction pointer or particular memory addresses. The user can place *BreakPoints* (BP) at particular instructions to make the debugger suspend execution when those are reached. In general there exist two different types: *software BPs* and *hardware BPs*. A software BP is realized by overwriting the operation to break at with a specific *software interrupt instruction*, e. g., `int 0x03`. When this is executed, the running process will pause and control is delegated to the attached debugger. In order to resume execution, the debugger has to remove the interrupt instruction and replace it with the original one. *Hardware BPs* are implemented in a different way. The instructions of the debugee are not modified, but the address to halt at is placed into some special debug register of the CPU. On the x86/64 architecture at most four different hardware breakpoints are available at any time. One remarkable advantage of hardware BPs is the ability to further specify the trigger conditions. For each one a combination of *read, write and execute* triggers can be specified. Accordingly, they can be used as data breakpoints as well. One different way to implement data breakpoints is by using *guard pages*. Every memory access to such a page triggers an exception that can be catched by the debugger. Obviously, this works on a much coarser granularity, since any single byte on the corresponding page will trigger the exception, while hardware BPs can use byte, word and double word (dword) granularity for the specified address.

Another important debugging technique is *tracing*, which can be realized on different granularity levels, e. g., on instruction, on branch or on function call. The first case is called *single stepping* and all processors offer native support for that. Since full instruction traces – even of simple programs – are very large, often a coarser granularity is used. In case of *branch tracing* one is only interested in the particular execution path, i. e., in the edges of the CFG and CG and not in the internals of each basic block. Branch tracing can be realized with the help of debugger extensions or by hardware support [79]. An even coarser tracing perspective is the *function call level*, i. e., only the transitions of the CG are monitored. This kind of tracing can be used to quickly get an overview of the called system services and delivers a high-level view of an unknown application's behavior.

Especially in malware analysis behavior monitoring is very helpful. Therefore, several behavior analysis techniques and tools have evolved in the last years [30]. Depending on if the analysis is performed on a *real* [87] machine, a *virtualized* [6] or an *emulated* [1] one, the required data about function calls is gathered in different ways.

On a real machine, either *hooking*, OS-specific *callback* routines or *binary instrumentation* is used. *Hooking* is the redirection of control (or data) flow to a user-supplied target. During function call monitoring it is used to redirect execution to a custom *hook function* each time the *hooked* function is called. Inside the hook function different actions may be taken: the call parameters are observed, the hooked function is called or its effect are simulated, and afterwards the result value may be observed or modified. This all happens transparently, such that the caller does not notice that a hook was in place. There exist different techniques to actually implement hooking. One of them is to overwrite a function pointer in the *import* or *export table*, in the *vtable* of objects, or in the *system service table*. Another method is *inline hooking* [82], in which the first instructions of the target function are overwritten with an unconditional jump. Since the original operations are destroyed by that, they first have to be saved somewhere. To that purpose a *trampoline function* is created that consists of the saved original instructions followed by a jump to the remaining untouched code of the hooked function. By that, the trampoline realizes the same semantics as the originally hooked function. There exist many further alternative ways to redirect function calls, e. g., by using *proxy libraries* or by placing *breakpoints* at the target function prologues.

Another contemporary related technique is *binary instrumentation*. Here, additional code is inserted into an application in order to observe or modify its behavior.

Static instrumentation approaches exist for a long time now [19]. With those a binary is modified before it is executed. Newer *dynamic methods* [11; 46; 54; 80] perform the transformation during runtime when a code block is executed for the first time and, thus, are able to overcome the drawbacks of static methods like difficulties with indirect branches.

When using a *full system emulator* or a *virtual machine* for analysis, the technique of *virtual machine introspection* (VMI) [35] can be applied. It enables to monitor and control a VM from the outside, normally without being noticed from the inside. Today many virtualizers/emulators offer VMI support, e. g., *QEMU* [7; 74], *Xen* [5] and *Ether* [28].

One serious drawback of dynamic analysis is its restriction to only one execution path. Newer methods try to address this problem with *multi path execution* [50] or *dynamic symbolic execution* [74]. The idea is to execute *all* possible paths by forking parallel executions at branch points. The exponential size of the resulting execution trees lead to a *path explosion effect*. Therefore, *selective symbolic execution* [17] tries to confine on only a few *significant* branches to reduce the necessary space and runtime complexity and regain computability.

While most of the presented methods in this section aim at observing the control flow of a monitored application, *data tainting* is a current technique used to monitor the data flow instead. It enables the analyst to track the usage and influence of specific variables. To that end, labels can be assigned to specific variables or memory locations and each time those are used the label is propagated to the affected destination variable. The tainting can be done with static analysis or based on memory accesses during runtime [53]. Especially when tracking the flow of sensitive information, this technique empowers one to easily detect malicious code on an infected system [29].

4 Reverse Engineering Countermeasures

Anti-analysis protections originally have their roots in copy-protection mechanisms used against software piracy, but nowadays are also heavily used for malware and software that is concerned about security or theft of intellectual property. Since all protections can be broken [4], the aim often is not to render the analysis impossible, but at least to make it as hard as possible and to hide essential data within the irrelevant. Especially for malware the winning of time is crucial to reach maximum infection before an *AntiVirus* (AV) signature is available.

The different methods to protect a binary can be divided into *passive* and *active* measures. The passive ones try to disturb or complicate the static analysis approach, while the active measurements aim at the dynamic analysis process.

4.1 Passive Protection Measures

As already explained in Sect. 1 code optimizations [51] pose a real problem for RE and, hence, are heavily used for obfuscation as well. The problem with code transformations in general is that the originally used high-level-structures get harder to recover. This ranges from reduced readability of particular instruction sequences, e. g., when branchless coding, frame pointer omissions or function chunking is applied, to total decoupling from the initial instruction order, e. g., for utilizing CPU pipelining, optimizing branch prediction and speculative reads.

While code optimizations normally have a constructive aim, i. e., to improve the performance of a given application, *code obfuscation* techniques solely are used for protection and often even decrease computing speed. They try to prohibit program analysis by disguising the code semantics and data structures.

One main concept behind control flow concealment is the inability to predict the effective targets of *indirect branches*. If these cannot be determined during static analysis, the control and data flow remains (partially) unknown without actually executing the application. But even then, only those branch targets for the specific run will be known. In such cases a *def-use-analysis* can be performed to resolve the targets of the indirect branches. For each *used* variable target, the effective *definition* of that very value has to be determined. This imposes a strong dependency between control and data flow, i. e., they become co-dependent [84]. In order to further complicate the identification of branch targets *aliases* are used, which are different symbols for the same memory location. This is an effective method, since precise alias detection in presence of pointers and recursive data structures is undecidable [84].

Another important concept are *opaque predicates*, i. e., boolean expressions that are non-trivial to compute and always evaluate to a-priori known values, which are independent from the inputs. An extension to this are *opaque constants* [49], which resolve to arbitrary values and not only to *true* or *false*. Obviously, the reasoning in both cases is to harden the analysis. With the help of these constructs conditional branches can be placed into the binary that are never taken during runtime. The instructions at those branch targets contain junk code or *unaligned/overlapping* instructions [21] in order to confuse disassembly. Since most CISC architectures have variable size instruction sets and instructions do not have to be memory aligned, it is possible that two (or more) instructions share the same bytes and – depending on the address at which execution starts – different operations are performed. As an effect all following instructions are disassembled differently, depending on the used disassembly start address.

Most contemporary disassemblers use recursive traversal and one weakness of this method is that it relies on static determination of all branch targets. Another weak point is expecting each branch target to contain valid code and each call operation to eventually return. These assumptions can be exploited to disturb the disassembling. One method is to remove all branches and delegate them

through one or more dedicated *branch functions* [45]. For further complicating the analysis, the branch targets are dynamically calculated, most effectively by using aliasing pointers and opaque predicates.

Another popular way to obscure the control flow is the usage of *exception handlers*, since those allow implicit control transfer and impose the problem to determine the currently active handler in case of a triggered exception. For example, static analysis will only yield an inconspicuous `div` instruction, but by forcing a division by zero one can ensure that on execution, an exception is thrown and control is transferred to the handler currently in place.

Also multi-threading or multiple processes with *interprocess communication* can be used to further complicate understanding the code flow, since it is very hard to comprehend timing and synchronization issues and to infer the execution order of particular code sequences in the presence of multiple threads.

One essential requirement to understand the behavior of an application is to identify which *system services* are called. In order to hide the revealing names of those system functions, the import table of an application can be obfuscated. To that end, the table is either destroyed or reduced to contain only a few essential entries. Then, during runtime, the addresses of necessary system functions are obtained without the help of the OS. For further obfuscation those functions are not referenced by their name, but located by searching the memory for precalculated hash values.

Runtime packers [56] compress and/or encrypt the content of executable files, add a small decompressing stub and modify the *Entry Point* (EP) such that execution will first invoke the stub to reconstruct the original binary in memory. While those packers were invented in times when memory was expensive and bandwidth limited, nowadays they are mostly applied solely to protect a binary from being reversed. Therefore, the compression is being more and more neglected, and only encryption and obfuscation is applied. Nevertheless, they still are called *packers* and the inverse process of reconstructing the original file version is called *unpacking* [10; 36; 40; 47; 58; 65; 75].

One effect of packing a binary is the changing of its complete content, e. g., every single byte is modified. This feature was heavily utilized by virus authors in the past, since AV scanners are no longer able to detect malicious files by simple static signature matching. As a consequence, the AV companies came up with signatures that detected the constant decompression stubs instead. Thereupon, the malware scene responded with *Polymorphic code*. With that the appearance of the stub is changed with each infection or propagation, mostly by reordering the instructions, swapping used registers and inserting junk code at various places. *Metamorphic code* [83] goes one step further by transforming not only the decoder stub, but the complete file instead. As a result, there never is any unique form of the original file reconstructed in memory. Instead, with every single replication a new (semantically equal) binary is created.

One of the most sophisticated protection scheme is the application of customized *virtual machines*. Each time a binary is protected in this way, a custom instruction set and a corresponding virtual CPU is created. The binary is then translated into that instruction set and combined with code that implements that interpreting CPU. Enhanced versions of this protection create a new interpreter and instruction set on each replication. Several commercial protectors use this technology, e. g., *VMProtect*, *Code Virtualizer* and *Themida*, and it is very hard and time-consuming to reverse such protected applications [64; 70].

4.2 Active Protection Measures

Active protection methods try to disturb and misguide the dynamic analysis. Accordingly, they become active during runtime. If they detect that the protected file is under observation, they react in manifold ways. Some cause a different behavior of the analyzed file in order to mislead the analyst and conceal its real semantics. Others try to attack or crash the analyzing host or even to break out from a controlled analysis environment to do actual harm to the underlying system. There exist a broad range of protection techniques from simple to highly sophisticated ones and one commonly used umbrella term for all of them is *anti-unpacking* [14; 31; 33]. In order to categorize them in some meaningful way, we differentiate between *anti-debug*, *anti-emulation*, *anti-virtualization* and *anti-dump* methods. Many of the following described methods cannot be definitely assigned to one single category. In those cases we have tried to choose the category that fits best.

Anti Debugging

The most intensively used tool for dynamic analysis is the *debugger*. Hence, detecting and dealing with them is an essential task for applications that require protection. There are many different ways to disclose the presence of an attached debugger, from which some are generic and others react on debugger-specific side effects. Since debugging support is an essential system feature, it is normally integrated into the operating system. Hence, there exist system services that can be used to query for an active debugger, e. g., the Windows API functions `IsDebuggerPresent()`, `CheckRemoteDebuggerPresent()` or `NtQuerySystemInformation()`. Since these system calls easily can be defeated to hide a running debugger, a more effective way is to inspect the underlying data structures directly, e. g., the `Process Environment Block` (PEB) which contains the field `BeingDebugged` or the `DebugPort` field of the kernel structure `EPROCESS`.

When running inside a debugger, the runtime environment differs in several places. For instance, normally the parent process of a manually started application is

different to one started in a debugger. There are ways to identify specific debuggers by checking the environment for their known side-effects, e. g., the name of related windows, files or virtual devices. Moreover, a lot of detection methods are based on exception handling. In the normal case, an exception handler is called instantly when an exception is raised. In a debugging session the debugger first receives a *first chance notification* and the possibility to intervene *before* the exception handler is invoked. Time measuring can be used to detect this detour through the debugger. In general, *timing attacks* are a generic protection method to cope with all different kinds of analysis tools. Nearly all active or passive observation methods somehow interact with the control flow of the analyzed application or the utilization of the systems facilities, e. g., the caching mechanisms. Therefore, always a performance overhead arises that can be measured and compared against a base line of native machines.

When using a debugger for program analysis, breakpoints are used to stop at certain functions or instructions. Consequently, many protection schemes try to either detect or disable those breakpoints. Software BPs can be detected by scanning for their related instruction opcodes. Hardware BPs are identified by examining the corresponding *debug registers*. Additionally, *checksums* can be used to detect code modifications like software BPs, hooks in place and other code patches. For that purpose the checksum of a code region is precalculated at compile time and then compared against the current memory contents during runtime. A way to *circumvent* installed hooks or patches is based on the fact that those modifications normally are placed only on the *first* few instructions of a function. The idea now is to store a copy of those original first instructions inside the binary itself and execute this copy before calling the function not from its entry point, but with an offset that points to the location directly behind the copied instructions. By that, possibly modified operations at the function beginning are skipped.

As an alternative to only *detect* a debugger, some times efforts are made to completely *prevent* the execution of a binary within such a tool. One way to do so exploits the fact that there at most can be *one* debugger per process: with *self debugging* a second process is spawned that attaches as a debugger to the first one. A more enhanced version is *circular self debugging* in which at least three different processes exist and each one is debugging and being debugged at the same time.

Another way to prohibit the execution of a binary on an analysts' host is to bind the file to one particular machine. In case of the *Rustock.c* rootkit this was realized by encrypting each copy of the sample with some machine specific hardware value, such that the resulting file becomes executable only on this specific system [15].

Malformed files can be used to crash particular analysis tools, mostly while they try to parse the file structure. There have been several flaws that were exploited in the PE loading process of debuggers, e. g., the usage of specially crafted table sizes could lead to memory corruption in *OllyDbg*.

Other examples of debugger-specific techniques that exploit implementation flaws are the `int 0x2d` operation that can be used to crash *SoftIce* or a specially crafted call to `OutputDebugString()` that does the same to *OllyDbg*.

Furthermore, there are ways to break out from the debugging session and execute malicious code without the analyst noticing. One method exploits the *Thread Local Storage* (TLS) facility of PE files. The TLS offers several callback functions that can be used to execute code each time a new thread is created or destroyed and one of these callbacks is invoked even before the EP. Another method uses *code injection* to execute malicious operations from within other running processes.

Anti Emulation

Emulation is an important technique that is used for analysis within many AV products. In order to remove potential existing packer stages, execution of the binary is emulated for some amount of time until a heuristic states that the file has completely unpacked. Thereupon the resulting memory content is searched for known malware signatures or otherwise related traces. Anti-emulation techniques utilize the incompleteness of such emulators in several areas: the instruction set, the system services and the system environment, e. g., rarely used CPU instructions are executed or less utilized APIs are called. Furthermore, due to performance reasons AV products always use an upper bound for the amount of emulated instructions. Therefore, some binaries start execution with large or nested loops in order to exceed this limit to conceal their real semantics. Finally, the generic all-purpose detection technique *Timing* can be used to detect emulators as well.

Anti Virtualization

The term *virtualization* aggregates multiple techniques that operate on different abstraction levels, depending on how much hardware support is used and how much code is executed on the native CPU. It ranges from *pure software emulation* (everything is emulated), over *reduced privilege set* (most of the code is executed natively, but critical instructions have to be simulated) over to *hardware-assisted* virtualization (the CPU offers specific virtualization support and, hence, all code is executed natively). Depending on the used technique, different detection methods can be applied. Those for software emulation *(Hydra, Bochs, QEMU in emulation mode)* have already been discussed in the previous paragraph.

Reduced privilege set virtualizers (*VMWare*, *Xen*, *VirtualPC*, *Parallels* in traditional mode) have to virtually duplicate certain low-level CPU facilities that only exist *once* on the system. This is realized by intercepting and simulating each access by the VM. Unfortunately, many

traditional *Instruction Set Architecture* (ISA) are not fully virtualizable, which means there are sensitive instructions that are not privileged [63]. As a consequence, the hosting machine is not informed if such facilities are accessed by the guest and this fact can be exploited to detect the virtual environment. For instance, one popular VM detection method is to read out the value of the *Interrupt Descriptor Table Register* (IDTR) [67] and compare it against the known values of native machines. Another example is to utilize some rarely used *Model Specific Register* (MSR) [59], since those mostly are not simulated.

Detecting hardware-assisted virtual machines (newer version of *Xen, Parallels, Xen, Virtual Box* on *Intel VT* or *AMD-SVM* architectures) is a bit more complicated, but possible as well. All known methods utilize the different low-level behavior even when running on hardware with VM support, e. g., timing issues or reduced TLB performance.

Besides those generic detection methods, there also exist numerous ways to identify each particular virtualizing software. One way is to fingerprint the environment by looking for particular device names, hardware addresses or running system services. Other methods exploit backdoor facilities of the various products that normally are used for guest-to-host communication, e. g., the port `0x5658` in *VMWare* or the special `0x0f` opcodes in *VirtualPC* [34].

An issue much more critical than detection is *breaking out* from a virtual machine. There was a flaw in VMWare's shared folders implementation [81] that allows an attacker to read and write to arbitrary locations on the host system from within the virtual machine. By exploiting this vulnerability it is possible to compromise the underlying host and control it.

Anti Dumping

Many protected binaries are heavily armored by multiple stages of encryption, compression and/or code obfuscation. To enable analysis, they first have to be transformed back into their original unprotected form. One effective solution for this is to actually execute the binary and identify the point in time in which the *Original Entry Point* (OEP) is called. At that point a memory dump is saved to disk, since one can assume that all unpacking stages have been completed. Subsequently, the *Imports* have to be reconstructed with the help of the still running unpacked process to get a valid executable, since those have been stripped while packing the binary.

In order to prevent memory dumping, several protection techniques can be applied. One simple method is to modify or destroy the PE layout once the binary is fully loaded. Normally, this information is only needed for mapping a file appropriately into memory, but dumping an already loaded module back to disk requires it as well.

The method of *stolen bytes* [72] or *stolen functions* destroys essential parts of the binary after storing an encrypted version of them within the unpacker stub. When the binary later on is unpacked, these instructions are reconstructed in dynamically allocated memory and all branches to their original location are redirected to that location. If the binary is dumped, the dynamic memory regions obviously are ignored, rendering the resulting dump file invalid.

A related technique removes all branch instructions from a binary and replaces them by so called *Nanomites* [60], e. g., the `int 0x03` instruction. When executing the program, a second process is started that attaches to the original one as a debugger. Each time a Nanomite is encountered, the debugger is invoked and transfers control to the original branch target, depending on the current context information. The mapping between context and target destination is stored in the *Nanomite table*. To further complicate reconstruction of the original file, a bunch of fake Nanomites are inserted into the table and the binary at unreachable code locations.

Another anti-dumping approach employs *guard pages* for realizing *page by page en-/decryption* [60]. In that case, the binary is not unpacked in a whole, but each memory page is decrypted on the fly on its first access. This complicates the dumping process, since there is no point in time in which there is a complete unpacked version of the binary in memory, with still having its initial state. One way to break this method is by intentionally *touching* each memory page and enforcing their decryption by that.

5 Outlook

In this work, we have surveyed the current state of the art in the area of reverse code engineering, and discussed how software tries to protect itself from such analysis. In contrary to the public meaning, RE can be applied for many constructive purposes like vulnerability research and malware analysis. Recently, there have been many improvements in control and data flow analysis from which the analysis process benefits a lot. For example, techniques like symbolic execution become more and more scalable and even complex systems can nowadays be analyzed using such techniques. In contrast, some very sophisticated protection techniques can be used as countermeasures for reverse code engineering. While already being known and used for several years now, one of the most powerful method utilizes *virtual machines* to protect software and avoid analysis. Like in many IT security areas, there is an ongoing cat-and-mouse game between the analysts and those who want to misguide and inhibit their efforts. Accordingly, we expect to see various new protection methods evolving in the future and, as a consequence, appropriate new analysis methods as well.

References

[1] Norman ASA. Sandbox analyzer. http://www.norman.com/security_center/security_tools/.

[2] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In: *Proc. of 13th Int'l Conf. on Compiler Construction*, CC, pages 5–23. LNCS 2985, Springer-Verlag, 2004.

[3] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. In: *ACM Trans. on Programming Languages and Systems*, 32:23:1–23:84. ACM, 2010.

[4] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In: *Proc. of 21st Annual Int'l Cryptology Conf.*, CRYPTO 2001, pages 1–18. LNCS 2139, Springer-Verlag, 2001.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In: *Proc. of the 19th ACM Sympos. on Operating Systems Principles*, SOSP '03, pages 164–177. ACM, 2003.

[6] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. In: *Journal in Computer Virology*, 2(1):67–77, 2006.

[7] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In: *Proc. of the Annual Conf. on USENIX Annual Technical Conf.*, ATEC '05, page 41. USENIX Association, 2005.

[8] J. Bergeron, Mourad Debbabi, M. M. Erhioui, and Béchir Ktari. Static analysis of binary code to isolate malicious behaviors. In: *Proc. of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises*, WETICE '99, pages 184–189. IEEE Computer Society, 1999.

[9] The GNU binutils. objdump. http://sourceware.org/binutils/docs/binutils/objdump.html.

[10] Lutz Boehne. Pandora's Bochs: Automated Unpacking of Malware. Diploma Thesis, University of Mannheim, January 2008.

[11] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In: *Proc. of the Int'l Symp. on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, pages 265–275. IEEE Computer Society, 2003.

[12] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol format using dynamic binary analysis. In: *Proc. of the 14th ACM Conf. on Computer and Communications Security*, CCS 2007, pages 317–329. ACM, 2007.

[13] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: *Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation*,OSDI'08, pages 209–224. USENIX Association, 2008.

[14] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In: *Proc. of the 38th Annual IEEE Int'l Conf. on Dependable Systems and Networks*, DSN'08, pages 177–186, 2008.

[15] Ken Chiang and Levi Lloyd. A case study of the rustock rootkit and spam bot. In: *Proc. of the First Workshop on Hot Topics in Understanding Botnets*, page 10. USENIX Association, 2007.

[16] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. In: *IEEE Software*, 7(1):13–17, 1990.

[17] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In: *ACM SIGPLAN Notices – APSPLOS'11*, 46(3):265–278, 2011.

[18] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to high-level language translation. In: *Proc. of the Int'l Conf. on Software Maintenance*, ICSM '98, pages 228–237. IEEE Computer Society, 1998.

[19] Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey, and Brian Lewis. Experience in the design, implementation and use of a retargetable static binary translation framework. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2002.

[20] Lori A. Clarke. A program testing system. In: *Proc. of the 1976 Annual Conf.*, ACM '76, pages 488–491. ACM, 1976.

[21] Frederick B. Cohen. Operating system protection through program evolution. In: *Computers and Security*, 12(6):565–584, 1993.

[22] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Krügel, and Engin Kirda. Prospex: Protocol specification extraction. In: *IEEE Symp. on Security and Privacy*, pages 110–125, 2009.

[23] NuMega / Compuware. Softice.

[24] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In: *Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation*, OSDI'08, pages 255–266, 2008.

[25] Weidong Cui. *Discoverer : Automatic Protocol Reverse Engineering from Network Traces*, In: *Proc. of the 16th USENIX Security Symp.*, Security'07. USENIX Association, 2007.

[26] Weidong Cui, Helen J. Wang, Marcus Peinado, Luiz Irun-briz, and Karl Chen. Tupni: Automatic reverse engineering of input formats. In: *Proc. of the 15th ACM Conf. on Computer and Communications Security*, CCS'08, pages 391–402. ACM, 2008.

[27] DaFixer. Dede delphi decompiler. http://www.softpedia.com/get/Programming/Debuggers-Decompilers-Dissasemblers/DeDe.shtml.

[28] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In: *Proc. of the 15th ACM Conf. on Computer and Communications Security*, CCS'08, pages 51–62. ACM, 2008.

[29] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In: *Proc. of the USENIX Annual Technical Conf. 2007*, pages 18:1–18:14. USENIX Association, 2007.

[30] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware analysis techniques and tools. http://www.iseclab.org/papers/malware_survey.pdf.

[31] Nicolas Falliere. Windows anti-debug reference. http://www.symantec.com/connect/es/articles/windows-anti-debug-reference, 2007.

[32] Nicolas Falliere, Liam O. Murchu, and Eric Chien. W32.Stuxnet Dossier. Technical report, Symantic Security Response, October 2010.

[33] Peter Ferrie. Anti-unpacker tricks. http://pferrie.tripod.com.

[34] Peter Ferrie. Attacks on virtual machine emulators. http://pferrie.tripod.com/papers/attacks.pdf, 2007.

[35] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In: *Proc. of Network and Distributed Systems Security Symp.*, NDSS 2003, pages 191–206. The Internet Society, 2003.

[36] Fanglu Guo, Peter Ferrie, and Tzi cker Chiueh. A study of the packer problem and its solutions. In: *Proc. of the 11th Int'l Symp. on Recent Advances in Intrusion Detection*, RAID'08, pages 98–115. LNCS 5230, Springer Verlag, 2008.

[37] Inc. Immunity. Immunity debugger. http://debugger.immunityinc.com/.

[38] Daniel Jackson and Eugene J. Rollins. A new model of program dependences for reverse engineering. In: *Proc. of the 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering*, SIGSOFT'94, pages 2–10. ACM 1994.

[39] Noah M. Johnson, Juan Caballero, Kevin Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential slicing: Identifying causal execution differences for security applications. In *Proc. of the 32nd IEEE Symp. on Security and Privacy*, SP'11, pages 347–362. IEEE Computer Society, 2011.

[40] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: a hidden code extractor for packed executables. In: *Proc. of the 2007 ACM Workshop on Recurring Malcode*, WORM '07, pages 46–53. ACM, 2007.

[41] Akos Kiss, Judit Jasz, Gabor Lehotai, and Tibor Gyimothy. Interprocedural static slicing of binary executables. In: *Proc. of the 3rd IEEE Int'l Workshop on Source Code Analysis and Manipulation*, pages 118–127. IEEE, 2003.

[42] Bogdan Korel and Janusz W. Laski. Dynamic program slicing. In: *Information Processing Letters*, 29(3):155–163, 1988.

[43] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In: *Proc. of the Network and Distributed System Security Symp.*, NDSS 2010. The Internet Society, 2010.

[44] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In: *Proc. of the Network and Distributed System Security Symp.*, NDSS 2008. The Internet Society, 2008.

[45] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In: *Proc. of the 10th ACM Conf. on Computer and Communications Security*, CCS '03, pages 290–299. ACM, 2003.

[46] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In: *ACM SIGPLAN Notices – Proc. of the 2005 ACM SIGPLAN Conf. on Programming language Design and Implementation*, 40(6):190–200. ACM, 2005.

[47] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In: *Proc. of the Annual Computer Security Applications Conference*, ACSAC 2007, pages 431–441, 2007.

[48] Microsoft. Windbg. http://msdn.microsoft.com/en-us/windows/hardware/gg463009.aspx.

[49] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In: *Proc. of the Annual Computer Security Applications Conference*, ACSAC 2007, pages 421–430, 2007.

[50] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In: *Proc. of the 2007 IEEE Symp. on Security and Privacy*, SP'07, pages 231–245. IEEE Computer Scociety, 2007.

[51] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[52] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In: *Proc. of the 8th European Symp. on Programming Languages and Systems*, ESOP'99, pages 208–223. LNCS 1576, Springer Verlag, 1999.

[53] Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In: *Proc. of the Network and Distributed System Security Symp.*, NDSS 2007. The Internet Society, 2007.

[54] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: *ACM SIGPLAN Notices – Proc. of the 2007 ACM SIGPLAN Conf. on Programming language Design and Implementation*, PLDI'07, 42:89–100. ACM, 2007.

[55] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[56] Markus F. X. J. Oberhumer, Laszlo Molnar, and John F. Reiser. Upx – the ultimate packer for executables. http://upx.sourceforge.net.

[57] The GNU project Debugger. gdb. http://www.gnu.org/software/gdb/.

[58] Danny Quist and Valsmith. Covert Debugging Circumventing Software Armoring Techniques. In: *Blackhat Briefings*, 2007.

[59] Thomas Raffetseder, Christopher Krügel, and Engin Kirda. Detecting system emulators. In: *Proc. of the 10th Int'l Conf. on Information Security*, ISC 2007, pages 1–18. LNCS 4779, Spirnger Verlag, 2007.

[60] Silicon Realms. Armadillo. http://www.siliconrealms.com/.

[61] Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. In: *ACM SIGSOFT Software Engineering Notes*, 20(4):41–52. ACM, 1995.

[62] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. In: *Trans. of the American Mathematical Society*, 74(2):358–366, 1953.

[63] John Scott Robin and Cynthia E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In: *Proc. of the 9th Conference on USENIX Security Symp. – Volume 9*, page 10. USENIX Association, 2000.

[64] Rolf Rolles. Unpacking virtualization obfuscators. In: *Proc. of the 3rd USENIX conference on Offensive Technologies*, WOOT'09, page 1. USENIX Association, 2009.

[65] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In: *Proc of the 22nd Annual Computer Security Applications Conf.*, pages 289–300. IEEE Computer Society, 2006.

[66] Mark Russinovich. Sony, rootkits and digital rights management gone too far. http://blogs.technet.com/b/markrussinovich/archive/2005/10/31/sony-rootkits-and-digital-rights-management-gone-too-far.aspx, 2005.

[67] Joanna Rutkowska. Red pill... or how to detect vmm using (almost) one cpu instruction. http://invisiblethings.org/papers/redpill.html, 2004.

[68] Hex Rays SA. Hex rays decompiler. http://www.hex-rays.com/decompiler.shtml.

[69] Hex Rays SA. Ida pro disassembler and debugger. http://www.hex-rays.com/idapro/.

[70] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In: *Proc of the 30th IEEE Symp. on Security and Privacy*, pages 94–109, 2009. IEEE Computer Society, 2009.

[71] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: a dynamic excavator for reverse engineering data structures. In: *Proc. of the Network and Distributed System Security Symp.*, NDSS 2011. The Internet Society, 2011.

[72] ASPack Software. Asprotect. http://www.aspack.com/asprotect.html.

[73] DotFix Software. Vb decompiler. http://www.vb-decompiler.org/.

[74] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, Newso James, Pongsin Poosankam, and Prateek Saxena. c. In: *Proc. of the 4th Int'l Conf. on Information Systems Security*, ICISS'08, pages 1–25. LNCS 5352, Springer-Verlag, 2008.

[75] Joe Stewart. OllyBone: Semi-automatic unpacking on IA-32. In: *Defcon 14*, 2006.

[76] Sysersoft. Syser. http://www.sysersoft.com.

[77] Andrew Tridgell. How samba was written. http://www.samba.org/ftp/tridge/misc/french_cafe.txt, 2003.

[78] A. M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. In: *Proc. of the London Mathematical Society*, 2(42):230–265, 1936.

[79] Amit Vasudevan, Ning Qu, and Adrian Perrig. XTRec: Secure real-time execution trace recording on commodity platforms. In: *Proc. of the 44th Hawaii International Conference on System Sciences*, HICSS, pages 1–10, 2011.

[80] Amit Vasudevan and Ramesh Yerraballi. SPiKE: Engineering malware analysis tools using unobtrusive binary-instrumentation. In: *Proc. of the 29th Australasian Computer Science Conf. – Vol. 48*, ACSC'06, pages 311–320. Australian Computer Society, 2006.

[81] Common Vulnerabilities and Exposures. CVE-2008-0923. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0923.

[82] Redmond Wa, Galen Hunt, Galen Hunt, Doug Brubacher, and Doug Brubacher. Detours: Binary interception of Win32 functions. In: *Proc. of the 3rd USENIX Windows NT Symp. – Vol. 3*, pages 135–143. USENIX Association, 1998.

[83] Andrew Walenstein, Rachit Mathur, Mohamed R. Chouchane, and Arun Lakhotia. Normalizing metamorphic malware using term rewriting. In: *Proc. of the 6th IEEE Int'l Workshop on Source*

Code Analysis and Manipulation, pages 75–84. IEEE Computer Society, 2006.

[84] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, University of Virginia, Charlottesville, VA, USA, 2000.

[85] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. ReFormat: Automatic reverse engineering of encrypted messages. In: *Proc. of the 14th European Conf. on Research in Computer Security*, ESORICS'09, pages 200–215. LNCS 6879, Springer-Verlag, 2009.

[86] Mark Weiser. Program slicing. In: *Proc. of the 5th Int'l Conf. on Software engineering*, ICSE'81, pages 439–449. IEEE Press, 1981.

[87] Carsten Willems, Thorsten Holz, and Felix C. Freiling. Toward automated dynamic malware analysis using cwsandbox. In: *IEEE Security and Privacy*, 5:32–39, 2007.

[88] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, and Engin Kirda. Automatic network protocol analysis. In: *Proc. of the 15th Annual Network and Distributed System Security Symp.*, NDSS 2008. The Internet Society, 2008.

[89] Oleh Yuschuk. Ollydbg. http://www.ollydbg.de/.

Received: September 11, 2011, accepted: November 27, 2011

Dipl.-Inform. Carsten Willems is a Ph. D. student at the Ruhr-University Bochum, where he is doing research on software security and malware. After he gained his diploma in computer science at the University RWTH in Aachen, he founded his security company CWSE GmbH in 2007. This company has developed the malware analysis suite CWSandbox, which is nowadays used by many AV vendors, banks and governments.

Address: Ruhr-University Bochum, Chair for Systems Security, Universitätsstraße 150, D-44780 Bochum, Germany, e-mail: cwillems@cwse.de

Prof. Dr.-Ing. Felix C. Freiling is a full professor of computer science at Friedrich-Alexander University Erlangen-Nuremberg. He is interested in all aspects of dependable and secure computing, theory and practice.

Address: Friedrich-Alexander University Erlangen-Nuremberg, Department Informatik, Martenstraße 3, D-91058 Erlangen, Germany, e-mail: felix.freiling@cs.fau.de