From AntiVirus to AntiMalware Software and Beyond: Another Approach to the Protection of Customers from Dysfunctional System Behaviour

Dr. Klaus Brunnstein
Professor for Appplication of Informatics
Faculty for Informatics, University of Hamburg, Germany
Brunnstein@informatik.uni-hamburg.de

Paper submitted to 22nd National Information Systems Security Conference
Status: July 23, 1999

Abstract: As users tend to rely on systems of growing complexity without themselves being able to understand or control malevolent behaviour, threats contained in software must be well understood. The paper deals with different aspects of malicious software (malware), both self-replicating (aka viruses and worms) and "pure" payloads (aka Trojan Horses) which are understood as additional though unwished and unspecified features of systems of programs; such system or software features are regarded as "**dysfunctional**". As traditional definitions somewhat lack consistency which is prerequisite to describing complex dysfunctionalties, and as they are partially self-contradicting and incomplete concerning recent threats, a definition is developed which distinguishes "normal" dysfunctionalties (produced through weaknesses of contemporary Software Engineering) from "intentionally malevolent" ones. Complex real threats may be built from two atomic types, namely self-replicating and Trojanic elements, each of which may act under some trigger condition. Based on experiences collected from tests, AntiMalware methods need further developments, both concerning classification of newly experienced threats and concerning online detection in user systems.

1) Introduction: About dysfunctional software and user vulnerability:

With further growing velocity, size and functional complexity of digital artifacts (aka computers, network systems, digitally-controlled infrastructures etc), users become both growingly dependent upon proper work of those artifacts (from hardware and device drivers to operating systems and application software), and at the same time they become lesser and lesser able to understand and control whether some observed function or system behaviour is "what they need or should get". While the WYSIWYG principle ("What You See Is What You Get") postulates that any internal behaviour may be "observed" by its visual effects, this principle is not applicable to complex system functions (e.g. interoperation of tasks in a multi-tasking operating system), and it is even less applicable to observing functions and impact of "active content" travelling through networks and influencing local systems via hidden entries in network software (browsers etc). In some sense, Ralph Nader`s observation (when adressing missing safety features of automobiles, in the 1950`s) "unsafe at any speed" is even more applicable to contemporary Iinformation and Communication Technologies.

Nobody can therefore be surprised that users have difficulties to understand "unforeseen" effects. Based on some common understanding that present systems are not sufficiently secure and safe,

many users (including IT specialists whose expertise is in other areas than security and safety) tend to project any unforeseen effect onto the irrational diagnosis "I have been hit by some new virus", even if actual AV products dont support any such suspicion (e.g. based on their heuristic methods). Both this "viral assumption" and the usual attempt to escape "ill understood" situations by simply restarting the system (the key combination CTRL+ALT+DEL is what most users learn first) originate from the fact that users have (almost) no means or information at all (except accumulated experience) to understand the "proper" work of their digital artifacts, and they therefore cannot develop some understanding about whether a deviating behaviour is possibly hazardous.

One basic reason for the powerlessness of users comes from the fact that features of all these technologies related to digital artifacts are almost exclusively determined (that is: specified, designed, implemented, distributed, installed, maintained and updated) by the views of their manufacturers, and that the dominance of the supply side is not balanced to any meaningful degree by requirements of the users. Even worse: customers have just to "Take What You Get" (TWYG) which in turn makes "control" and "understanding" difficult if not impossible. In contemporary systems, users are just "using" technologies but are surely not "in control". With the advent of truely network-based artifacts (from applets to other forms of active content travelling though networks), "user control" becomes even more difficult to achieve.

As enterprises, agencies, institutions and individuals continue to build their economic existence upon such digital artifacts, vulnerability of such entities grows correspondingly. If the "normal" functions of these technologies are hardly masterable, malicious intent or ill-advised experimentation (e.g. of youngsters spreading viruses, Trojans or hacker toolkits) tends to further increase the vulnerability of enterprises, governments, individuals and societies. Consequently, there is strong need to find new ways to (somewhat) empower users to master failures of digital artifacts performing "essential" (if not "mission critical") work.

For some time, potentially hazardeous software has been growing both in numbers and in diversity of types. While old-fashioned system and file viruses (infecting either systems via boot processes or with the help of executable "host" programs) tend to be contained (although still growing in numbers and diversity, as the recent growth of 32-bit PE infectors demonstrates), the advent of powerful "script languages" such as Visual Basic for Applications (VBA) or Java has significantly increased threats of self-reproducing software both for local systems ("viruses") and for networks ("worms"). As such script languages become broadly used in standard and application software (e.g. from office applications to CAD/CAM systems, from distributed databases to electronic commerce), and as they closely interact with their operating software, active contents import growing risks into enterprises, offices and workstations of individuals.

Following traditional thinking, any risk of digital artifacts must be balanced with other "adequate" and "adapted" digital artifacts. In this respect, growingly complex "guardians" (from filters in firewalls to on-access scanners in servers and workstations) aim at protecting users - if properly maintained - from related (esp. known) threats. This approach feeds a whole branch of security expertise. But customer protection then depends upon "adequate reaction" and proper consciousness of related experts. Risks of this approach can well be illustrated with cases of "Non-Viral Malware" (NVM). While all manufacturers of AntiVirus Software are working very

hard to keep in pace with the developments of new self-reproducing software (both concerning viruses and worms), opinions are strongly divided whether and how to protect users from non-self-replicating malware such as Trojan Horses. While some AV experts are pragmatic enough to help protecting their customers from such threats even if antiviral methods don´t apply "well" to such "pure" payloads, others argue that mechanisms to handle self-reproducing software is not optimized to handle other malware (which is technically correct to some degree), and that one should consequently not care for Trojans as long as customers don´t broadly complain about such threats.

Indeed, one essential problem of the contemporary approach to defining "malware" starts from the assumption that a software program may contain some element (a virus or Trojan payload) that its manufacturer did not intend or implement. Moreover, finding such malware can usually not be based on any information from the manufacturer how the related original software works. To the contrary, AntiMalware experts must often apply Reverse-Engineering techniques to understand the intended ("normal") software behaviour; such analyses need high technical competence are are extremely time-consuming. The application of such techniques, however, may seriously contradict the interests (e.g. "intellectual property rights") of the original manufacturers.

Without putting contemporary Information and Communication technologies at disposal "in principle", this paper analyses whether at least avoidance of or, if some threat materializes, detection and cure of "malicious software" can be handled in a different way to support user abilities to understand and "control" what is going on.

2) Traditional classification of types of malware:

Within the IT/Network Security and Safety curriculum (4 semesters = 2 years) for advanced students at the Faculty for Informatics at Hamburg university (first full cycle started in winter semester 1988/89), computer viruses and other forms of malicious software have been analysed (as practical examples in learning reverse-engineering methods) in some detail since the advent of first viruses (on PCs, Brain/Pakistani boot virus in 1986 and Jerusalem file virus in 1987). Based on cooperation with several AntiVirus experts, antiVirus Test Center (VTC) maintains databases of viral and malicious software against which regular AV tests have been performed since 1991 [VTC Uni-Hamburg].

Originally, the term "computer virus" was introduced by F. Cohen in his doctoral thesis [Cohen 1986]. His theoretical approach (describing self-replication modelled upon a Turing machine), although more systematic than others, did not influence the practical development of viruses and countermeasures. Moreover, Cohen`s definition (nased on a Turing machine) is not directly applicable to other forms of malicious software including self-replication in networks (aka propagation). On the "practical" (that is also: less systematic) level, there are almost as many different definitions of computer viruses, and some books also describe other forms of "rogue" software [e.g. Brunnstein 1989; Ferbrache 1992; Highland 1990; Hoffman 1990; Slade 1994; Solomon 1991].

In his doctoral thesis, V. Bontchev [Bontchev 1998] gives a survey of the most relevant types of malware. Some of his most important definitions are:

Logic Bombs:	"The *logic bombs* are the simplest example of malicious code. They are rarely stand-alone programs. Most often, they are a piece of code embedded in a larger program. The embedding is usually done by the programmer (or one of the programmers) of the larger program."
Trojan Horses:	"A *Trojan Horse* is a program which performs (or claims to perform) something useful, while in the same time intentionally performs, unknowingly to the user, some kind of destructive function. This destructive function is usually called a *payload*."
Subtypes of Trojan Horses are:	Regular Trojan Horses (available from BBS), Trapdoors, Droppers, Injectors and Germs
Droppers:	"A *dropper* is a special kind of Trojan Horse, the payload of which is to install a virus on the system under attack. The installation is performed on one or several infectable objects on the targeted system."
Injectors:	"An *injector* is a program very similar to a dropper, except that it installs a virus not on a program but in memory."
Germs:	" A *germ* is a program produced by assembling or compiling the original source code (or a good disassembly) of a virus or of an infected program. The germ cannot be obtained via a natural infection process. Sometimes the germs are called *first generation viruses*."
Computer Virus:	" A *computer virus* is a computer program which is able to replicate itself by attaching itself in some way to other computer programs. ... (The) two main properties of the computer viruses (are) —merely that a virus is able to replicate itself and that it does it by always attaching itself in some way to another, innocent program. This process of virus replication and attaching to another program is called *infection*. The other program, i.e., the program that is infected by the virus is usually called a *host* or a *victim* program."
Worms:	"Programs which are able to replicate themselves (usually across computer networks) as stand-alone programs (or sets of programs) and which do not depend on the existence of a host program are called computer *worms*."
Subtypes of "Worms":	Chain Letters, Host Computer Worms (with a special form called Rabbits), and Network Worms (with its special form "Octopus" where the central segment manages the worm`s behaviour on the network).

Based on knowledge of its time, such definitions are rather "ad hoc" and they are neither sufficiently systematic nor applicable to forms of malware unexperienced at the time when related papers were published. This can well be studied - from today`s views - with Bontchev`s definitions:

> From the perspective of a **general form of "payload"** (which is also inherent in most "real" viruses), "logic bombs" are a special case of "Trojan Horses" (with trigger conditions of type "logic", and with the special case of "time bombs" where trigger condition is a logical condition including time/clock setting).

From the general case of **"self-reproducing software",** there exist two cases, namely **self-reproduction in single systems** (either "viruses" or "host worms"), and **propaga-tion in networks** (as originally described by Shoch-Hupp).

For viruses, **"infection"** needs some sort of **"host"** which may EITHER be a (compiled) program (as discussed by Bontchev, applicable under "traditional" operating systems such as DOS, UNIX or VMS) OR some form of "active content" (applicable to inter-preted systems such as Microsoft`s Visual Basic for Applications (VBA5/6), Visual Basic Script (VBS), or JavaScript and Java Applets). The latter case - not explicitly foreseen in Bontchev`s definition of viruses - applies to recently important cases as macro viruses and Java viruses (Strange Brew and BeanHive).

Interestingly, the well-known existence of **"virus toolkits"** such as "mutating engines" (which may be used to add polymorphic features to some viral code) and of **"malware construction toolkits"** (which allow laymen to construct their own viruses and Trojan Horses) are not addressed in these definitions, despite their broad availability (and although AV products have detected such forms for some time).

With rapid deployment of new forms of complex malicious software, and with features from several different categories - such as WNT/RemoteExplorer, which is a worm carrying and dropping a virus that has a special payload - there is some need for a systematic classification which also permits one to differentiate between intentionally constructed "malicious software" and, on the other hand, less intended but equally destructive instances in "normal" software.

Moreover, a new view of "hostile" software is needed as contemporary threats must address much more than traditional forms such as hacking and viruses. Under "holistic" perspectives [Brunnstein 1997] of security (traditionally addressing hacking and viruses) and safety (including also network attacks such as Denial-of-Service, spoofing etc), understanding of "improper functioning" or **dysfunctional operation** needs broader perspectives than "traditional" (less holistic) views.

3) Towards a systematic classication of "dysfunctional software"

Let us start with the **assumption** that **any software** that may become "essential" for some business or individual (in the sense that this institution becomes "dependant" upon its proper working, and that improper working and functions may lead to vulnerability), is the **product of a systematic, well documented and controlled engineering process**. Ideally, this "Software Engineering" (SE) process would start with **codification of "requirements"** that must be fulfilled when the software is used in the related application domains. At least, such requirements are needed where systems are **"mission critical"** for the proper work of systems concerning risks of life and health. Even for such critical applications, it is not always possible to anticipate in which environnments and under which conditions related software may work in the future. Formal codification of requirements can therefore not generally be postulated.

Remark: it must be admitted that the process described above does hardly apply to contemporary systems and software for business applications. Even somewhat "alien" functions are often regarded to be "not a bug but a system feature". On the other hand, software for critical applications is developed with such systematic SE processes. With

growing dependency of business upon computing and communication, it is both likely and desirable that "mature" SE methods are applied also in business software.

Let us at least **postulate** that we have, as achievement of a systematic SE process, some **"specifications"** (or assertions) that describe any function which may have an effect on the system in-use. Such a list of specifications (which may be functions permitted or such prohibited) may be used as a basis to conclude whether a software behaves "according to its specification"; with the existence of such a list, there would be no need for in-depth code analysis or reverse-engineering to understand deviating behaviour.

In the **ideal case** where software functions have been formally specified and their proper implementation has been formally proven, this may be sort of "proof-carrying code" [Lee-Necula 1997]. Such a formal approach is adequate essentially for areas where formal models are somewhat applicable (e.g. in avionics, process control software or mobile code for sensitive applications). In **most practical cases**, specification of functionality will be **rather informal**, describing present and potentially absent features. In order to "assure" the user of the presence of specific functions (and about a possible absence of others), any software may be characterized by a set of specifications of functions that the manufacturer assures its customer to be present or absent (e.g. "no function to format disk" or "does not format files"). Transmission of software could then contain some "contractual part" including such assertions of functionality.

> **Basic assumption:** for any software product, there is a given **"platform"** on which it works. Such platform must be uniquely defined e.g. by specifying the hardware, operating system or higher software levels such as office systems with script languages. Example: INTEL_PENTIUM/Windows-NT 4.0 (SP 5)/Office-2000. It is permissible to use abbreviations for single elements (e.g. W2k for Windows 2000) or classes of elements (e.g. W32 for Windows 95/98/NT/2000).

Under this assumption, we may "know" (to some degree) which functions a software product performs. Let us call the **set of functions specified** in this way **its "functionality"**:

> **Definition #1**: **A program´s or module´s or object`s "functionality"** is characterized by the set of all specifications, formal or informal, from which information about "proper work" of a program can be concluded, and from which certain undesired functions can be excluded.
>
> Remark: it is irrelevant whether the manufacturer`s specifications, formal or informal, are explicitly known to the user. Even if a manufacturer decides to hide functions (e.g. for objects with limited visibility and inheritance), such functions belong to the functionality of a software. If a manufacturer decides to include some hidden Trojanic payload, then this becomes part of the specification and therefore the functionality of that software. In cases where specifications are not explicitly known to the user, there must be some means (such as checksums) of assuring the user about the integrity of the software "as specified".

From here, we can start with stepwise development of a **classification of deviations from such functionality** which we call **"dysfunctionalities".**

Definition #2: **A software or module is called "dysfunctional"** when at least one function deviates from the specification.

"Dysfunctional behaviour" may result from **unintended effects** in the programming process, such as inadequately tested software or usage of inadequate tools (compilers etc.) which e.g. may produce dysfunctionalities. "Dysfunctional behaviour" may **originate** from the program manufacturing process, e.g. by faults in conception of the software architecture, or by errors in the programming process. Recognizing that the "state of the SE Art" is at present far from guaranteeing avoidance of any such fault, such dysfunctionalities must not be interpreted as "intentionally dysfunctional".

Definition #3: A software or module is called "intentionally dysfunctional", when some essential feature is not contained in the manufacturer`s specification, whether formal or informal.

"Intentional dysfunctionality" esp. addresses cases where significant deviations from the specifications can be observed. Such intentional deviations may be undocumented traps and backdoors installed for debugging but not removed, but there may also be Trojanic elements deliberately implanted e.g. to collect data. Such "unspecified" features may be implanted either by the original manufacturer or by third parties, e.g. during distribution or in the installation process. Both cases differ "only" by the assessement of who is responsible for the dysfunction. Moreover, proper sealing and protection of a module would also guarantee that the functionality is "as specified" by its manufacturer.

Moreover, dysfunctional behaviour may also **result from intentional action**, committed in the distribution process, during installation or during operation. In any case, **dysfunctional software behaviour must be avoided** or, if detected, handled in a way to protect users from unspecified software behaviour. That implies that **any detection of dysfunctional behaviour must immediately lead to some intervention** (e.g. stopping related processes).

Letting aside problems of dysfunctional behaviour originating from inadeqate software engineering at the manufacturer´s side, this paper specifically addresses problems of dysfunctionalities added by third parties with the deliberate intent to change the list of functionalities (e.g. adding self-replication or a triggered payload function).

Definition #4: A software or module is called "malicious" ("malware")
if it is **intentionally dysfunctional**, and if there is sufficient evidence (e.g. by observation of behaviour at execution time) that dysfunctions may adversely influence the usage or the behaviour of the original software.

Concerning malware, it is interesting to study how some formerly functional software is transformed into malware. Let us call this process "contamination".

Definition #5: A software or module with **given functionality** is transformed into "malware" by a process called **"contamination".**

Examples of "contamination" are: **"infection"** of some module with given functionalty by some self-reproducing software, either in single systems such as viruses, or in networks such as "worms"; in the network case, this process is also called **"propagation"**. Another case of contamination is the process of **"Trojanization"**, where some unspecified function is added. Evidently, it is this process which imports unwanted effects into systems that is otherwise well-behaved. As "infection" and "Trojanization" are different processes, real malware may consist of more than one level of contamination. A "real" virus often consist of an infection module, and it is further contaminated with one or more Trojanic functions; both the infection and Trojanic parts may depend upon specific conditions, called "triggers".

> **Definition #6: "Infection" is the process where some self-reproducing software is added (as first generation instance of an infection) or where it continues to add itself to (usually formerly non-infected) instances of software of a given type; infection may depend upon given (trigger) conditions.**

This definition generalizes infection processes both for single and distributed systems. It is applicable to all known types of viruses and worms that may be distinguished by the specific platforms on which the respective intentional dysfunctions operate.

> **Definition #6a: Any software that reproduces (or "self-replicates"), possibly depending on specific conditions, at least in 2 consecutive steps upon at least one module each (called "host") on a single system on a given platform, is called a "virus" (for that platform). A viral host may be compiled (e.g. boot and file virus) or interpreted (e.g. script virus).**
> Remark: if a virus replicates only once or twice, it is called "intended".

Numerous examples of viruses are known, both for diverse hardware platforms and related operating systems (prevalent on Intel-based systems, but also as demonstrations of virus possibility also on UNIX, VMS and other platforms) and for diverse software platforms such as script languages (prevalent for Microsoft`s Visual Basic for Applications, but also demonstrated for JavaScript and Java).

> **Definition #6b: Any software that reproduces (or "propagates"), possibly depending on specific conditions, in at least two parts (nodes) of a connected system on a given platform is called a (platform) "worm", if it has the ability to communicate with other instances of itself, and if it is able to transfer itself to other parts of the network using the same platform.**

While viruses and worms are in principle **observable** and rather easy to detect, namely during their contamination (=replication) process (where they affect a new previously unaffected host which is not protected from such actions by adequate means), **"trojanisation"** (=the addition of Trojanic functions) is usually not directly observable. Indeed, Trojanic functions are usually only detected when some Trojanic function is "fired", where an expert observer may in principle collect some "post mortem" information. On the other side, such Trojanic functions are contained

in many viruses and worms, and their functions (e.g. displaying texts, deleting files or formatting disks) are what users fear more than the infection process.

> **Definition #7: A "Trojan Horse" is a software or module that, in addition to its specified functions, has one or more additional hidden functions (called "Trojanic functions") that are added to a given module in a contamination process ("trojanization") usually unobservable for a user. These hidden functions may activate depending upon specific (trigger) conditions.**
>
> Remark: In this sense, "Trojanicity" is a feature of a software. This approach differs significantly from the assumption that a Trojan Horse is not a Trojan Horse to the person who knows its behaviour but for the one who does not. "Users" rarely "understand" what software does (they simply use it because of their assumption of desired effects), so they can hardly distinguish between normal and Trojanic functions.

This definition applies to many types that are at present understood as being "Trojanic", but there is one essential difference:

> Any software whose functions, even if potentially hazardous, works **as functionally specified**, with no sign that it may contain some hidden (Trojanic) functions, would **not be called "Trojan Horses"** under this definition. In this sense, hacker tools such as **NetBus, BackOrifice or BO2k** (as long as there is no evidence that they do more than what is specified) **are not Trojan Horses** according to these definitions.
>
> If a specific name for such software is needed, such types of "ill-advised" and potentially alien software may be called **"critters"** or alike, but they don´t belong to malware categories. On the other side, there may be some need to detect such critters, but this is simple as they are easily characterized with normal (static) means (signatures, checksums).
>
> Equally, **renaming some known critter** will not change its substance, and it will esp. not transform it into a "Trojan Horse" as there are no hidden functions. Example: FORMATC.EXE will upon start immediately format the hard disk; renaming it to HELP.EXE does neither change the specification nor the implemented function although a naive user may assume that a module of this name is "helpful" and start the program, with the related result.

"Real malware" may be constructed by repetitively combining different types or instances of self-reproducing software for one or several platforms with Trojanic functions. As an example, let us consider **WNT/RemoteExplorer** (which was detected in an IntraNet of some US enterprise on December 17, 1998):

WNT/RemoteExplorer	is a **worm** self-replicating on Windows-NT servers, where it installs itself using the "service" function. From a first infected server, the worm propagates copies of itself to other Windows-NT servers.

It has a payload to **install and propagate a virus** locally in systems infected by the worm.
Finally, the virus distributes itself within the local systems, and its payload compresses selected objects.

Consequently, WNT/RemoteExplorer is a virus-carrying worm with built-in Trojanic compression function.

In summarizing the classifications, we can describe the most general case:

> **Definition #8: Malware may be developed from a given (functional) software or module by intentionally contaminating it with unspecified (hidden) functions. Such malware may consist of combinations of self-replicating or propagating part, or both, which may be triggered by some built-in condition. Malware may include hidden Trojanic functions, which may also activate upon some built-in condition (trigger). The development of malware (in the contamination process, namely the Trojanization) may be observed in cases of self-reproducing software, but it is (at present) difficult to anticipate the malicious Trojanic behaviour before it materializes.**

Under these definitions, (at least) any currently known malware can be characterized completely by its combination of replicative and Trojanic parts, and it can be distinguished (at least "in principle") from an non-malicious software (suitably labeled "goodware").

4) Measures for protection against dysfunctional software

Customer protection concerning self-replicating code - computer viruses - is rather well developed, as several AntiVirus tests demonstrate. Concerning at least those viruses reported to be "In-The-Wild" [WildListOrganisation], several products reach favourable detection levels of 100%. And many products detect even the mass of "zoo" viruses (for PC platforms, presently about 40,000 viruses), which have only been reported in few instances (if at all), at levels of more than 90%. Customer protection against computer viruses (either knowns or closely related ones) is therefore **rather well developed** - at least by "good products".

Regrettably, protection from non-self-replicating malware is significantly less developed. Related VTC tests demonstrate that only very few products detect a significant portion of Trojan Horses, although VTCs Trojan zoo collection is deliberately limited to those reported and known to several AV producers. In the VTC test "1998-10" (published in October 1998), only 4 AV (out of 30) products detected at least 90% of the file and macro malware (essentially Trojan Horses and special variants such as droppers and "intended" viruses which don´t properly work). While many AV manufacturers at least try to protect their customers from such threats, some deliberately argue that they don´t see a need for protection, and one manufacturer even forbidded VTC for some time to test the related product for its degree of ability to detect known "malware".

> Remark: the final version of this paper will include a discussion of malware detection as tested presently in VTC test "1999-03" (results available March 1999).

Evidently, the consciousness of some AV experts that users must at least be protected from "easy-to-handle" malware forms must be further developed, either by instruction or by "market forces". Several reasons may contribute to a possible neglect of such threats:

- AV experts have no knowledge or methods to detect and counter non-viral threats.
- Or AV experts don´t have samples and are not convinced that such threats exist.
- Or AV experts are yet overloaded with AntiVirus work and have no time left.

While the second and third reason are not acceptable for those users who have suffered from a manifestation of one of several hundred trojans or droppers, the first argument must be taken serious:

> Viruses are comparatively easy to detect as they reproduce (at least under properly established and reproduced conditions), and as they can therefore be distinguished from the "host" without any need to know much about it. Contrarily, a "Trojanic" function can not be analysed without some knowledge (that is: reverse-engineering and analysis) of the host into which the Trojanic function has been implanted. In the lack of any knowledge about the "normal" code, it is indeed technically difficult to assess the degree of some suspected "Trojanicity".
>
> On the other side, both viruses and Trojanized software reach AntiMalware labs when customers have collected some suspicion about some potentially dysfunctional behaviour. In ancient times, viruses were also manually "dissected" (reverse-engineered), but for some time, suspicious code has been subjected to some - often automated - test method which analyses its viral or non-viral nature.
>
> Admittedly, inspection of Trojanic or other malicious code must currently be done "in some manual mode". But as automatic virus detection systems were developed over time, similar methods can be developed to observe suspicious dysfunctional behaviour of non-viral methods. (Following patterns of the "Virus Intrusion and Detection Expert System", VIDES [Swimmer 1997], a Masters thesis at Hamburg university develops a method to observe Trojanic behaviour in network environments [Engel-Lessig 1999]).

Indeed, significant work must be invested into developing adapted methods to detect non-viral malware. Here, any sort of specification the "normal" functions of some host (as described in part 3) would help AntiMalware experts to locate and analyse Trojanic dysfunctions faster, and to adapt their AntiMalware software to warn users when some new suspicious malware is found. At the very least, detection of analysed malware may start using similar methods as applied for the detection of traditional viruses:

> **Static means** include (combinations of) signatures of typical code or (constant) data. Moreover, checksums over (parts of) typical Trojanic code can be used for detection. This also applies to non-malicious code, such as standard compiler libraries.

In order to develop some **dynamic detection** (or at least some form of "suspicion generation"), comparable to contemporary heuristic virus detection methods, significant methological work is needed. On the other aside, users will adequately assess the "value" of such programs that can also protect them against non-viral malware.

5) Outlook

Apart from supporting better understanding of non-viral malware, one general aspect of the classification presented here may be more valuable for users: if software comes with some form of assertion about its inherent functions, users have some means not only to observe whether some function conforms with given specifications, but they may also decide whether such software supports to some minimum degree a list of their own "requirements". This may be especially valuable for software offered and transmitted through a network:

> **General prerequisite**: "Never install any software transmitted without any assurance of specification (and without any authentication of the sender)"
>
> **Any specification**, carried with software transmitted via network, may be **compared to a (locally held) list of requirements**. Provided that some threshhold of "coincidence" of both lists is reached, the software may be accepted (provided that the sender is properly authenticated).
>
> On the next level, during installation of such software with assured functionality (which may include signatures and checksums of essential parts), a **malware auditing system** may inspect the new software for (static) signs of known malware (e.g. based on signatures and checksums), and it would inhibit any execution of suspicious or detected dysfunctional software.

Ideally, additional **surveillance** of benevolent behaviour would become possible when **dynamic dysfunctional behaviour** could be detected in a heuristic process where indications of suspicious dysfunctional behaviour are **collected and accumulated** to trigger interceptive actions when a threshhold of malicity is reached. This ideal will only be reached through serious theoretical and practical developments.

6) Acknowledgement:

Within the community of AntiVirus experts, definitions of different forms of malicious software are vidivly and often controversially discussed, both in academic and business constraints. The author has learned much from disputes with his students and various friends and experts. He wishes to especially thank Bruce P. Burrell, Marko Helenius, Padgett Peterson and Sarah Gordon for their support in discussing issues related to this paper.

7) Literature:

[Brunnstein 1989] Klaus Brunnstein: "Computer Viren Report" (in German), WRS Verlag, Planegg (1st edition: 1989; 2nd edition: 1991)

[Brunnstein 1997] Klaus Brunnstein: "Towards a Holistic View Of Security and Safety of Enterprise Information and Communication Technologies: Adapting to a Changing Paradigm",
Invited lecture, IFIP SEC 97, Copenhagen, May 1997

[Cohen 1986] Frederick B. Cohen: "Computer Viruses", PhD Dissertation, University of Southern California, 1986

[Bontchev 1998] Vesselin V. Bontchev: "Methodology of Computer Anti-Virus Research", Doctor Thesis, Faculty for Informatics, University of Hamburg, 1998

[Engel-Lessig 1999] Stefan Engel, Stefan Lessig: "Internet-based attacks and selected countermeasures", Diplom thesis, University of Hamburg (to be published: summer 1999)

[Ferbrache 1992] David Ferbrache: "A Pathology of Computer Viruses", Springer Verlag London, 1992

[Highland 1990] Harold Highland: "Computer Virus Handbook", Elsevier Publ, 1990

[Hoffman 1990] Lance Hoffman: "Rogue Programs - Viruses, Worms, and Trojan Horses", Van Nostrand Reinhold, 1990

[Lee-Necula 1997] Peter Lee, George Necula: "Research on Proof-Carrying Code for Mobile-Code Security", DARPA Workshop on Foundations for Secure Mobile Code, Monterey, March 1997

[Slade 1994] Robert Slade`s "Guide to Computer Viruses: How to avoid them, how to get rid of them, and how to get help", Springer Verlag, New York, 1994

[Solomon 1991] Alan Solomon: "PC Viruses: Detection, Analysis and Cure" Springer Verlag, London 1991

[Swimmer 1997] Morton Swimmer: "Virus Intrusion Detcdtion Expert System (VIDES)", Diplom thesis, University of Hamburg, 1997

[VTC Uni-Hamburg] antiVirus Test Center, Hamburg University: test reports of AntiVirus and AntiMalware software (published regularly in summer and winter) are available from:
http://agn-www.informatik.uni-hamburg.de/vtc

[WildListOrganisation] The "List of Known Viruses" is maintained on regular basis

by "The WildList Organisation International". For details see:
http://www.wildlist.org