ELSEVIER

Computers & Security Vol. 17, No. 1, pp. 69-89, 1998

0167-4048/98 $19.00

Macro virus identification problems

Vesselin Bontchev

FRISK Software International, Postholf 7180, 127 Reykjavik, Iceland, E-mail: bontchev@complex.is

Computer viruses written in the macro programming language of the popular office applications like Microsoft Word have become extremely widespread. Unlike the MS-DOS viruses which are single entities, the macro viruses often consist of entire sets of several independent macros. This poses some interesting theoretical problems to the virus-specific anti-virus software that attempts to identify exactly the viruses it detects. Two viral sets of macros can have common subsets — or one of the sets could be a subset of the other. The paper deals with the problems caused by this, some of which are extremely difficult, if not impossible to solve. Emphasis is put on how the difficulties could be exploited by the virus writers and how the anti-virus products should be improved in order to be made resistant to such attacks and to avoid damaging the user's documents when misidentifying the virus in it and attempting to remove the wrong virus variant.

1. The Need for Exact Virus Identification

Before we begin tackling the macro virus identification problems, it is worthwhile mentioning why exact virus identification in general and exact macro virus identification in particular are important. After all, historically, most scanners have always worked by picking some small part of the virus and using it as a 'scan string' to detect all other instances of that virus. However, such an approach has several drawbacks.

Firstly, it carries the very real danger of misidentification — i.e., confusing one virus (e.g., a destructive one) with another (which, for instance, is not intentionally destructive). In the past year we saw one anti-virus producer making a fool of itself in public by publishing a press release which warned about the imminent activation of the destructive virus `Tedious` which was, according to the press release, widespread — and, of course, urged everybody to get that anti-virus producer's anti-virus program. That press release initially caused significant puzzlement among the competent anti-virus researchers. Even if we left aside the ethical problems caused by using scare tactics to persuade the public to buy one's product, it was relatively well known that `Tedious` is not widespread at all and, most important, does not have an activation date or any payload whatsoever. That is, it is not intentionally destructive. As it turned out, the scanner of the anti-virus producer in question used a scan string which was unable to distinguish between `Tedious` and `Bandung` — and the latter virus is both widespread and destructive — thus leading to the confusion in the press release and, undoubtedly, to negative publicity for the people who have published it. That incident was relatively benign to the users —

but one could easily imagine the opposite mistake caused by misidentification. That is, a destructive virus could be reported by the anti–virus product as a non–destructive one, thus failing to warn the user.

Secondly, precise identification of the virus found is particularly important when virus removal (i.e., disinfection) is involved. Here, misidentification could lead to an attempt to remove the wrong virus variant — with fatal consequences for the infected object, which could be damaged beyond repair. This is already important enough in the world of DOS viruses. However, it is even more important in the world of macro viruses. For, it could be argued that the proper way of removing DOS viruses is by destroying the infected objects and by replacing them with virus–free backup copies. In such cases it does not matter much whether the virus in them has been identified exactly or not — the infected object is destroyed anyway. Macro viruses, however, usually reside in documents — which are bound to change often, and virus–free backup copies of which are usually not available. Therefore, disinfection of macro viruses by their removal from the infected documents is a must — and it is of ultimate importance that it is accomplished correctly, without damaging the document or any user macros present in it. This goal simply cannot be achieved reliably enough without the means of exact virus identification.

Thirdly, exact virus identification is often necessary for the purposes of technical support. The users often ask us "what does this or that virus do?" — because the virus in question has been found on their machine and they want to know what it could have done to their data. Answering this question correctly is often impossible if the scanner which has found the virus is unable to identify the virus exactly. Often the difference between a virus which is extremely destructive (for example, a data diddling virus like `WM/Wazzu.A`) and a variant of the same virus which does nothing is only a single byte — or even a single bit. Such a level of identification is practically impossible to achieve with scan strings alone. The only way to achieve it is to compute some kind of checksum of every single bit of the non–modifiable parts of the virus body.

Fourthly, exact virus identification is necessary for the purposes of proper reporting and tracking the spread of computer viruses. One authoritative source of such information is the so–called WildList, maintained by the anti–virus researcher Joe Wells. Many testers use it as a source of information what viruses to include in their in–the–wild test sets. Recently, the fact that this list does not identify exactly some of the viruses listed on it caused our scanner to score unfavorably in a comparative review. The list had the `Plagiarist` virus listed on it. As it turns out, there are several different viruses, all of them members of the `Plagiarist` family. Our scanner could detect one of them — the one which is really in–the–wild and which was reported originally to Joe Wells. It could not detect another of the variants, however — a variant which is not in–the–wild. However, since the WildList mentions just "Plagiarist" and does not identify the particular variant, one tester used that other variant which our scanner could not detect and wrote in the review that our product does not have 100% detection of the viruses known to be in–the–wild. One could easily imagine a similar mishap involving a macro virus.

Fifthly, exact virus identification is a very powerful protection against false positives. It is well–known that scanners which use only simple scan strings to detect viruses often cause false positives — i.e., report a virus in some innocent file which just happens to contain the same sequence of bytes picked by the anti–virus producer as a means for detecting the virus. At the same time, exact virus identification can never lead to such mishaps — because, if it identifies exactly a virus in a file, it simply means that the virus *is* there; no doubts about it, since every single bit of it has been identified and found to be present. And indeed, the products which use exact virus identification cause false positives significantly less often (almost never — unless they also try to detect new variants of the virus, because then exact identification cannot be used) than those which rely on scan strings.

Finally, exact virus identification is essentially the only way to handle VBA macro viruses (that is, macro viruses written in the programming language of Excel and the Office 97 suite). This is because, due to their

design, VBA programs contain lots of variable areas (which contain pointers to a common pool of identifiers — common to all VBA modules in the document). As a result of this the average length of the possible scan strings is only two bytes — clearly unsuitable for any practical use. The problem can be partially circumvented by using very long wildcard scan strings — i.e., scan strings which contain 'don't care' bytes in the positions of the variable pointers to the common pool of identifiers. Unfortunately, this is not a good solution either, because several different pieces of VBA code can compile to exactly the same image — with the differences becoming apparent only after the pointers to the identifiers are resolved. Clearly, this will increase the danger of false positives even further.

Due to all these reasons mentioned above, we maintain that it is very important for the virus-specific anti-virus products (e.g., scanners and disinfectors) to attempt to identify exactly the viruses they claim to detect. Exact identification is even more important in the case of macro viruses, because significant changes in the behavior of the macro virus can be the result of only a minuscule change in only one of its macros. Consequently, our own macro virus scanner rigorously attempts to identify exactly every single macro virus it claims to detect — and we urge all other anti-virus producers to take the same approach when implementing their scanners. Fortunately, many of them have already understood the benefits of exact macro virus identification — we are seeing many anti-virus products begin using it. In many cases, even if the scanner does not apply exact identification to the DOS viruses it detects, it at least applies it to the macro viruses it detects.

2. Definitions

Every well-designed anti-virus solution should be based on a careful definition of the problem it purports to solve. Therefore, a program designed to detect, recognize, identify and remove macro viruses should be based on a good definition of what a macro virus is.

In the world of DOS viruses it was sufficient to define a virus as a program that replicates. In the macro virus world the situation is not so simple, however. A macro virus does not necessarily consist of a single program (i.e., a single macro). It can consist of many macros (for instance, the `WM/Xenixos.A:De` virus consists of a couple of dozens of macros), some of which are more or less independent.

What does 'independent' mean? Well, there are three factors involved. Firstly, since WordBasic is an interpreted language, it is rather corruption-tolerant. That is, some parts of a macro can become corrupted — but, if they never receive control (or receive control only rarely — e.g., if they are in the payload of the virus), the error will never (or only rarely) become apparent. (In fact, this can be used to attack some kinds of anti-virus programs; see [Bontchev96].) Besides, the commands responsible for the replication (e.g., `MacroCopy`) usually occupy only a few lines (as many lines as there are macros to be copied). That is, the probability that a corruption will occur only in the part of the virus responsible for the replication is relatively small.

Secondly, WordBasic has powerful error trapping functions — and the virus writers often make extensive use of them. For instance, if a virus contains the operator

```
On Error Resume Next
```

at the beginning of its macros, even if control is transferred to some corrupted lines, they will simply be ignored and execution will continue from the first line after them which makes some kind of sense from the point of view of the WordBasic interpreter.

Thirdly, most macro viruses are written with significant redundancy. We are inclined to think that this is most likely the result of laziness than of foresight from the part of the virus writers. If they want to make a virus which intercepts several system macros and, therefore, replicates on several likely user actions (e.g., `FileSaveAs`, `FileSave`, `FileNew`, etc.), it is easier to them to simply copy the virus replication code in all these macros — instead of having it in a single macro which is called by all these macros. As a result of this, even if one of the macros of the virus

becomes so corrupted that it no longer works (or is even completely missing), the virus will still be capable of replicating — although its exact behavior is likely to change a bit.

All this (and some other factors, described in the next section) has forced us to define macro viruses not as single programs — but as sets of macros. In particular, we use the following:

Definition: *A macro virus is a set of one or more macros which set is capable of replicating itself recursively.*

Some parts of the above definition require further explanation. In particular, by 'recursive replication' we mean that an infected document can spread the virus to another document, this other document can spread the virus further, and so on. If the set of macros is capable only of copying itself elsewhere *once*, then it is not considered to be a virus. (We call such things 'Intended' — because the virus writers often produce programs which are obviously intended as viruses but are so buggy that they are unable to replicate themselves more than once — bugs which the virus writer has never discovered, because, as most virus writers, he has been afraid to run his own creation on his computer and test it.)

Furthermore, in the fashion of the first macro virus, `WM/Concept.A`, many macro viruses consist of slightly different sets of macros in the global template and in the infected documents. Therefore, for practical reasons it usually makes sense to consider the macro viruses as pairs of macro sets capable of replicating — with one member of the pair representing the viral set of macros in the infected documents and the other member of the pair — the viral set of macros in the infected global templates.

The main consequence of the above definition is that different sets of macros constitute different viruses — even if one of the sets is a subset of the other. Furthermore, even if a set consists of some of the elements of two other known sets, it is nevertheless a new, third virus. This should always be kept in mind when discussing 'different' and 'new' viruses in the next sections.

It should be emphasized that many of the macro virus identification problems described in this paper are tightly related to the definition of the term 'macro virus' given previously. Even those of them which seem unsolvable can be avoided completely by other, alternative definitions of the term.

One particularly attractive alternative definition relies on the idea of considering the macro viruses as completely independent macros which are not related in any way. For instance, according to this approach, a document does not "contain the `WM/Concept.A` virus" — instead, it is considered as containing "the macros `WM/Concept.A#AAAZAO` (two copies of this macro; the second one under the name `AutoOpen`), `WM/Concept.A#AAAZFS` and `WM/Concept.A#Payload`".

Unfortunately, such a definition has other, much more inconvenient practical problems. For instance, it seems silly to report "this document contains the `WM/Concept.A#AAAZFS` virus" — because this macro, all by itself and unsupported by the other members of the set, is unable to replicate and, therefore, is not a virus. Furthermore, the macros of two known viruses, when they meet each other on the same machine, can result in virus 'mating' and the creation of a new self–replicating set of macros. (For a more detailed description of this process, see [Bontchev96].) This new self–replicating macro set usually has properties and behaviour very different from the properties and the behavior of any of the original two sets — and, therefore, it makes much more sense to consider it a new virus.

For these reasons it seems that the definition we gave earlier in this section is the most practical and convenient one. We do not insist that it is perfect — but it is definitely the best one we could come up with so far. Therefore, our anti–virus product (and several other anti–virus products we are aware of) are based on it. In the rest of this paper we shall consider some of the interesting macro virus identification problems which arise as a result of it.

3. Easy Macro Virus Identification Problems

In this section we shall discuss some of the relatively easy to solve macro virus identification problems. They are presented in the order of increasing difficulty.

3.1. Devolving Viruses or the Rapi Virus Problem

One of the first problems of macro virus identification became apparent when the so–called **WM/Rapi.A** virus appeared. This virus consists of the following macros:

Document:	Global template:
AutoOpen	RpAO
RpAE	RpAE
RpFO	RpFO
RpFS	RpFS
RpFSA	RpFSA
RpTC	ToolsCustomize
RpTM	ToolsMacro
	AutoExec
	FileOpen
	FileSave
	FileSaveAs

However, due to a bug in one of the macros of the virus, when the virus is replicated via File/Save (as opposed to, say, File/Save As), some of its macros are not copied. Essentially, this results in a new macro set — which is still viral and is, therefore, a new macro virus. This new, 'devolved' virus, looks like this:

Document:	Global template:
AutoOpen	RpAO
RpAE	RpAE
RpFS	RpFS
RpFSA	RpFSA
RpTC	
RpTM	
	AutoExec
	FileSave
	FileSaveAs

This new set of macros, however, is not stable. It almost immediately devolves further to a new, even more reduced macro set. This third set is still capable of replicating itself, yet it is different from the previous two sets — therefore, it is a new macro virus. Furthermore, it is stable — in the sense that it does not devolve further. It looks like this:

Document:	Global template:
AutoOpen	RpAO
RpAE	RpAE
RpFS	RpFS
	AutoExec
	FileSave

The main consequence of this 'devolution' is that every particular 'main' variant of the **WM/Rapi** family can potentially generate two new variants. For instance, the existence of **WM/Rapi.A** also leads to the existence of **WM/Rapi.A1** (the second phase of the devolution) and **WM/Rapi.A2** (the third devolution phase). Furthermore, such new variants are created relatively often. Since the **AutoExec/RpAE** macro of the virus is relatively big and is not responsible for the replication process in any way (it contains the payload of the virus), it can easily become corrupted, or 'snatched' — i.e., replaced by another macro with the same name taken from another virus or from a legitimate macro package. (See [Bontchev96] for more information about macro snatching.) Furthermore, since this macro is preserved during all three phases of the devolution, it means that a single corruption of it in the main variant leads to the appearance of three new viruses. Modifications in some other macros (e.g., in **ToolsMacro/RpTM**) create fewer than three new virus variants, because these macros are 'lost' at some phases of the devolution and the resulting devolved variants can be equivalent to the variants produced by the devolution of some other main variant. For instance, the **WM/Rapi.C** virus devolves only to **WM/Rapi.C1**; on the third step a previously known variant, **WM/Rapi.A2**, is produced.

This problem was first discovered by David Chess ([Chess96]). It is relatively easy to solve — the scan-

ner's database simply has to contain up to three new entries for almost every such minor modification of the WM/Rapi virus. However, this leads to an annoyingly fast increase of the size of this database. Furthermore, it can sometimes lead to confusions — some users have trouble understanding how could a document containing one particular virus (e.g., WM/Rapi.A) infect their files with two different viruses, which, on the top of everything, are different from the original — WM/Rapi.A1 and WM/Rapi.A2.

3.2. Missing Macros or the Dzt Virus Problem

The second problem occurred soon after the WM/Dzt.A virus was discovered. This virus consists of the following macros:

Document:	Global template:
AutoOpen	FileSave
FileSaveAs	FileSaveAs

So far — nothing unusual. However, soon we discovered a new WM/Dzt variant, WM/Dzt.B, which looked like this:

Document:	Global template:
AutoOpen	FileSave

That is, the second macro of the virus was missing — yet the new virus was perfectly capable of replicating on its own! What is much worse, we soon found out that many of the scanners on the market would create WM/Dzt.B if they do not know WM/Dzt.A and are presented with the following situation:

1. A system's global template is infected with the WM/Concept.A virus. Then it contains the macros WM/Concept.A#AAAZAO, WM/Concept.A#AAAZFS, WM/Concept.A#FileSaveAs and WM/Concept.A#Payload.

2. The same global template then becomes infected with the WM/Dzt.A virus. As a result of this, its contents now becomes WM/Concept.A#AAAZAO, WM/Concept.A#AAAZFS, WM/Dzt.A#FileSave, WM/Dzt.A#FileSaveAs and WM/Concept.A#Payload. Note that the FileSaveAs macro of the WM/Dzt.A virus overwrites the macro with the same name which belongs to the WM/Concept.A virus.

3. A disinfector which does not identify macro viruses exactly decides that the system is infected with the WM/Concept.A virus — because it does not know about the WM/Dzt.A virus and the scan string used by the disinfector to detect the WM/Concept.A virus is found in the file. So, the disinfector tries to remove the macros of what it believes to be the WM/Concept.A virus, identifying them by their *names* (as opposed to by their contents).

As a result, the global template is left containing the WM/Dzt.A#FileSave macro — that is, precisely the WM/Dzt.B virus! In other words, the new virus variant was created by the disinfector!

This problem can be solved easily by forcing the disinfector to always identify the viral set of macros exactly and completely. That is, make sure that every single macro of it is indeed present by identifying the non–modifiable parts of the virus body, instead of relying on the macro names. Then it would easily notice in the above situation that one of the macros of what it believes to be the WM/Concept.A virus has different contents. Therefore, it is probably a new variant and should not be disinfected — instead, a sample should be requested from the user.

3.3. Variable Macro Sets or the CAP Virus Problem

The problem described in this section is significantly more difficult than the previous ones — and most anti–virus products have been for many months incapable of handling properly the virus which presented it for the first time to our attention. This was the WM/CAP.A virus.

This virus, written by a 14–year boy from Venezuela, spread like wildfire all over the world in a matter of

weeks — and is still one of the most often reported viruses. There were several reasons for this success. Firstly, the virus is language version independent — it works on any language version of Word. Secondly, documents infected with it do not show the typical symptom of many other WordMacro viruses — Word does not insist on saving them in the Template directory. Thirdly, as mentioned above, most anti-virus products had problems removing this virus properly, so it had a rather long time period in which to spread before the said products were updated to become capable of handling it.

By design, the virus is supposed to consist of either 10 or 15 macros. When infecting an English Word system, it is supposed to consist of the macros **AutoClose**, **AutoExec**, **AutoOpen**, **CAP**, **FileClose**, **FileOpen**, **FileSave**, **FileSaveAs**, **FileTemplates** and **ToolsMacro**. The macros **FileTemplates** and **ToolsMacro** are empty; they are present only in order to disable the system macros with the same names and, therefore, provide some limited form of 'stealth'. The remaining macros are essentially just calls to the **CAP** macro which contains most of the virus and is responsible for its replication.

When infecting a non-English version of Word, besides the above 10 macros, five additional macros are created. They are essentially copies of the macros **FileClose**, **FileSave**, **FileSaveAs**, **FileTemplates** and **ToolsMacro** and their names can be different, depending on the particular language version of Word.

In order to achieve Word language version independency, the virus does not address its macros by name. Instead, it examines the structure of the menus and uses as names of these additional five macros the names of the system commands which handle the menu items from the places where the File/Close, File/Save, File/Save As, File/Templates and Tools/Macro menu items are supposed to be. This indeed makes the virus Word language version independent. However, of course, it makes it dependent on the menu structure of the system it infects.

Furthermore, in order to ensure that its macros are copied, yet not address them by name (which would have made it Word language version dependent), the virus uses the percent sign ("%") in the descriptions of its macros and, during replication, copies to the infected document all macros, the description of which contains this sign. Also, in order to be completely sure that it will not 'snatch' any foreign macros, the virus first removes all macros from the document it intends to infect.

Unfortunately, due to a quirk in WordBasic unforeseen by the virus writer, this trick fails to achieve its purpose. As it turns out, a macro **Foo** with *no* description is copied over a macro **Bar** *with* description, this results in the body of the macro **Foo** replacing the body of the macro **Bar** — however, the description is not touched and it remains that of the macro **Bar**. (If both macros have descriptions, the description of the macro **Foo** replaces that of the macro **Bar**.) As a consequence of this, contrary to the virus author's expectation, the **WM/CAP.A** virus can snatch macros from other viruses and macro packages — if these viruses infect a system which is already infected with **WM/CAP.A** (or if the macro package is installed on such an infected system) and if they contain macros with the same names as the macros of the **WM/CAP.A** virus but with no descriptions.

Finally, since all the replication code of the virus concentrated in the **CAP** macro and all other macros are just calls to it, it is possible to corrupt, replace, and even remove any of the other macros — and the resulting set will still be capable of replicating itself — i.e., will still be a virus.

As a consequence to all these peculiarities, depending on the menu structure of the user's system and on what viruses that user's computer becomes infected with in addition to **WM/CAP.A**, many different self-replicating macro sets can result. The only constant thing seems to be the **CAP** macro. The other macros can be under bizarre names, can be corrupted, snatched or even missing (just the **CAP** macro and one of the other seven non-empty macros is enough to ensure that the set of macros is viral), or multiple copies of some of them can be present (under differ-

ent names, of course). It would be plain silly to consider all these different sets of macros as different virus variants.

Therefore, we were forced to revise our definition of what a macro virus is — in order to allow for such monstrosities as the **WM/CAP.A** virus. In particular, we define it now as a set consisting of the **CAP** macro, plus zero or more instances of the other eight different macros (the macros **FileTemplates** and **ToolsMacro** are both empty, so they are not different), while at least one instance of one of the non–empty macros.

If the above 'revised' definition looks a bit complicated, it is because it *is* complicated. Unfortunately, it is also the best one we have found so far that can handle things like the **WM/CAP.A** virus. Implementing it in an anti–virus product is not easy, either. As a temporary, stop–gap solution, some anti–virus producers have implemented the idea of identifying just the **CAP** macro and, if it is found in a document, remove all macros from that document. This is not as bad as it sounds, because the virus would have removed all user macros from that document when infecting it anyway.

Of course, this whole approach (both the revised definition and the stop–gap partial implementation of it) suffers from one very unpleasant drawback. During its replication, the virus could manage to snatch a macro which would modify its behaviour drastically. For instance, an auto macro snatched from another virus could make **WM/CAP.A** intentionally destructive on some dates — something which the original virus is not. The user will then rightfully wonder how come that a virus which was 'identified exactly' by his anti–virus product as **WM/CAP.A**, a virus which is known not to be intentionally destructive, has suddenly began displaying messages and deleting files on Friday the 13th. Unfortunately, this seems to be a price that has to be paid in order to solve the classification and identification mess which arises if we decide to consider all possible different macro sets of this virus as different variants.

3.4. Mass–Replicators or the Cebu Virus Problem

Some time ago a customer of ours sent us a document which he supposed was infected with a new virus. That, it indeed was. The virus, apparently, consisted of four macros — **AutoClose**, **AutoExec**, **AutoOpen**, and **MsRun**. (The reason for using such imprecise terms as 'apparently' will become clear in a moment.) Of those, only the **AutoOpen** macro was responsible for the replication of the virus. The **AutoClose** macro attempted to replicate the virus too, but always failed, due to a stupid bug. The **AutoExec** macro contained the payload and the **MsRun** macro was used as a "this document is infected" marker.

Problem was, the author of the virus, obviously an inexperienced programmer, had made some unwarranted assumptions. In particular, he seemed to have decided that the macros of his virus can be the only macros present in the document. After all, probably all virus–free documents he had seen did not contain any macros. So, he had decided, why bother with such complicated things like copying the macros of the virus one by one and trying to figure out whether it is infecting a document or a global template? There is a much simpler method — just write a loop which copies all macros from the current document to the document being infected (this is just three lines of WordBasic code) and that's it!

So, what happens if the infected document already contains some macros on its own? Right, these macros will become part of the virus and will be distributed to all other infected documents. In fact, the set of macros of the virus can only increase and acquire new members — never decrease!

The second problem was, when our customer began suspecting a macro virus which his anti–virus program failed to detect, he went and downloaded **ScanProt** — Microsoft's very own anti–virus tool against macro viruses. Unfortunately, besides the fact that **ScanProt** is mostly useless for detecting anything but the **Concept** macro virus, it is written in WordBasic and consists of several macros itself. So,

when our customer contacted us, he had `ScanProt`'s macros all over his documents and was definitely not amused.

Later we received other samples of the same virus. It had snatched macros from other macro packages, the `AutoExec` and the `AutoOpen` macros were overwritten by some anti–virus macros, and the `MsRun` macro was missing completely. Yet the macro set was still capable of self–replication — i.e., it was a virus.

Nowadays we call such macro viruses 'mass–replicators'. `WM/Cebu.A` (the virus described above) is not the only such virus; there are a couple of others. It seems that the only practical way of handling them is to identify only the bare minimum of macros responsible for the replication and, once they are identified, remove all macros from the infected documents. After all, they have already become part of the virus. This approach has a similar drawback to the one mentioned in the previous section — some of the snatched macros could result in a drastic change of the behaviour of the virus. However, this again seems to be a price worth paying for avoiding the horrible classification mess which would occur if any other method is used.

4. Difficult Macro Virus Identification Problems

This section presents some significantly more difficult macro virus identification problems. The first one is solved only in very few anti–virus products. No satisfactory solution of the second problem is known, as of yet.

4.1. Richard's Problem

The following interesting problem was brought to our attention by Richard Ford — then an anti–virus researcher at Command Software Systems, so we have named the problem after him ([Ford 96]).

Let us suppose that a known macro virus, `Foo`, consists of the set of macros {`A1, B1, C1`}. Then somebody takes this virus, modifies two of its macros, and produces a second virus, `Bar`, consisting of the set of macros {`A2, B1, C2`} where `A1` is different from `A2` and `C2` is different from `C1`. This new virus is not known to a virus–specific anti–virus product which is capable of exact macro identification and disinfection of macro virus remnants.

The anti–virus product will report a document containing the virus `Bar` as containing remnants of the virus `Foo` (namely, the macro `B1` which the anti–virus product can identify). So far, so good. However, if the anti–virus product attempts to disinfect the document, it will remove only the macro `B1` — since this is the only macro it can identify.

Now, let us suppose that the resulting set of macros {`A2, C2`} is capable of replicating itself (i.e., is a virus). Of course, during the replication, an error will occur — because an attempt will be made to copy the macro `B1` too — a macro which is not present because it was removed by the anti–virus product. However, if the viral macros use some form of error trapping (e.g., `On Error Resume Next`), the replication process *will* be able to complete successfully.

The bottom line will be that the anti–virus product would have created a new, third macro virus which is different from the existing ones (`Foo` and `Bar`). Most anti–virus producers wouldn't be pleased to learn that their product is creating new viruses.

There are several possible approaches towards a solution of this problem. Firstly, it is possible to combine the scanner with some kind of heuristic analyzer and remove all suspicious macros from the documents which are found to be infected with a known virus. Unfortunately, as demonstrated in [Bontchev96], it is possible to employ a wide range of anti–heuristic attacks and trick the heuristic analyser into a 'false negative' — i.e. make the virus undetactable by heuristics.

Secondly, a disinfector could remove *all* macros present in the documents found to be infected with a known virus (or to contain remnants of a known virus). Unfortunately, many users are relying on their

own macros and find such a 'solution' highly unsatisfactory. Judging from our own experience, this solution is not commercially viable.

Thirdly, a possible approach is to *never* disinfect a document in which only remnants of a known virus have been found. This approach has the significant drawback that disinfection will not be performed in many cases when it is perfectly possible and safe. Examples include misdisinfected viruses (when some but not all of the viral macros have been deleted by the user or by some inferior anti–virus product), viruses, some of the macros of which are so corrupted by Word that they have become unable to replicate, and so on.

Fourthly, it is possible to make the action "remove remnants" optional (and turned off by default) and have the user turn it on whenever necessary. Unfortunately, experience shows that most users lack the necessary anti–virus knowledge and expertise to take correct decisions in dangerous situations and we can be certain that many users will use this option in an inappropriate way or will turn it on "just in case" — regardless of whatever warnings the developer of the product includes that tell them not to do so unless they know what they are doing. Indeed, then it could be argued that the users themselves and not the anti–virus product are responsible for the creation of the new virus — and that they could have done so even with Word's Organizer command — by deleting some of the viral macros. Nevertheless, it would be better if the anti–virus product does not delegate such a responsibility to the inexperienced user.

A fifth approach, suggested by Richard Ford, is to determine for each particular virus the minimal subset of its macros that must be present, in order to be safe to remove the remnant (called the "minimal safe subset" of the virus). For instance, if all macros with the exception of the `PayLoad` macro of the `WM/Concept.A` virus have been found in a document, then it is obvious that it is safe to remove them. That is, by removing them one would not create a new virus, regardless of what other macros are present in the document — because the `PayLoad` macro is never executed by `Concept.A` and cannot have any viral capabilities by itself. (Note that this is just an example; we do not claim that the other three macros of `WM/Concept.A` form the "minimal safe subset" for this virus; it is perfectly possible that the size of the set can be reduced further by removing some additional macros from it — macros other than `PayLoad`.)

This method has the drawback that it can sometimes destroy (and therefore 'lose' for the anti–virus researchers) a new virus by mistaking it for a remnant of an old one. For instance, consider a variant of `WM/Concept.A` which differs from the original virus only by the contents of its `PayLoad` macro. A disinfector which behaves according to the algorithm described in the above paragraph would remove the other macros of the virus from the infected documents by mistaking them for remnants of the original virus. While this action will be 'safe' (in the sense that it cannot result in the creation of a new virus), the documents still will not be repaired properly (they will still contain one unwanted macro — `PayLoad` — and this will prevent the disinfector from turning off their template bit) and a new virus variant will be 'lost' for the anti–virus researchers.

A slight improvement of this method is to also remove the macros in the document which have the same *names* as the macros of the virus which are not included in its 'minimal safe subset'. However, first, we consider the identification of a macro by its name to be extremely unsafe and unreliable (a checksum of the macro body, combined with its length should be used instead) and, second, this improvement still does not solve the problem of 'losing' new virus variants.

We have come up with a better solution to Richard's problem, based on the following observations. If the only macros found in the document are the ones which form the remnant, then it is obviously safe to remove them. If there are any additional macros present, the following cases are possible:

1. The additional macros belong to the virus which is a new, unknown variant of a known virus. It doesn't really matter whether by removing them we would create a new virus, whether the new variant is produced intentionally, or whether it is a result of the ran-

dom corruption of one of the macros of a known virus. In all of these cases we have a new virus variant (although some corruptions might result in a non–working virus). Therefore, the correct approach is to refuse to disinfect anything (not even the remnant — i.e. the macros we have been able to identify as belonging to a known virus variant) and to request a sample from the user.

2. The additional macros are legitimate and belong to the user. This case will occur extremely rarely — only when a document which already contains legitimate macros (a rare enough situation by itself) is infected with a known virus and an improper attempt has been made to remove the virus — an attempt which has resulted in the removal of only some of the viral macros. Due to the relative rarity of this case, we think that it is affordable to refuse to perform any kind of disinfection on the document and to request a sample from the user.

3. The document has been infected by two different viruses — one known and one completely unknown, and an inappropriate attempt has been made to remove the known virus — an attempt which has resulted in the removal of only some of its macros. In this case the document contains one new virus. The user should be asked for a sample of it — so that the new virus can be analysed. Therefore, it is again affordable to refuse any kind of disinfection of the document — most users understand that virus–specific anti–virus programs like scanners are unable to remove unknown viruses.

The above can be summarized in the following simple rule:

If, after removing from the document the macros belonging to macro viruses which can be identified exactly, the remnants of a known virus are the only macros found in the infected document — remove them; otherwise do not remove any macros from the document and request a sample from the user.

4.2. Igor's Problem

When discussing Richard's problem, its possible solutions, and the need for exact virus identification in general, Igor Muttik, an anti–virus researcher at Dr Solomon's Software turned our attention to a natural extension of the problem with much more dangerous implications ([Muttik 96]). We have, therefore, named this new problem after him.

Let us consider a known virus, `Foo`, consisting of the set of macros {`AutoOpen, Payload`}. When an infected document is opened in a clean Word environment, the macro `AutoOpen` is executed. It determines that it is running from a document (as opposed to running from the global template) and copies the macros `AutoOpen` and `Payload` from the infected document to the global template and executes the macro `Payload`. When a clean document is opened on an infected system, the macro `AutoOpen` again receives control. It determines that it is running from the global template (as opposed to running from a document) and copies the macros `AutoOpen` and `Payload` from the global template to the document and executes the macro `Payload`.

Now, let us suppose that a virus writer takes this virus `Foo` and constructs another virus, `Bar`, which consists of the set of macros {`AutoOpen`, `Payload`, `AutoClose`, `NewPayload`}. The macros `AutoOpen` and `Payload` are identical to the macros with these same names in the virus `Foo` — and they work in exactly the same way. Again, when an infected document is opened in a clean Word environment, the macro `AutoOpen` is executed. It determines that it is running from a document (as opposed to running from the global template) and copies the macros `AutoOpen` and `Payload` from the infected document to the global template and executes the macro `Payload`. When that document is closed, the macro `AutoClose` receives control. It checks that the macros `AutoOpen` and `Payload` exist and have been copied to the global template. Then it copies the macros `AutoClose` and `NewPayload` to the global template too and executes the macro `NewPayload`. Similarly, when a clean document is opened on an infected system, the macro `AutoOpen` is executed. It determines that it is running from the global template (as opposed to running from a document) and copies the macros `AutoOpen` and `Payload` from the global template to the document

and executes the macro **Payload**. When that document is closed, the macro **AutoClose** receives control. It determines that it is running from the global template, checks whether the macros **AutoOpen** and **Payload** are present and have been copied to the document which is being infected, then copies the macros **AutoClose** and **NewPayload** to this document as well and executes the macro **NewPayload**.

If a scanner which knows the virus **Foo** but doesn't know the virus **Bar** scans a document infected by the virus **Bar**, it will seem to it as if the document is infected with the virus **Foo** — even if the scanner identifies exactly every single bit of the every macro of the viruses it knows about. (It should be noted that all virus–specific anti–virus programs which handle macro viruses are vulnerable to this attack — regardless of whether they do exact virus identification or rely on simple scan strings.) If run in disinfection mode, it will then proceed to remove what it believes to be the virus **Foo** — i.e. the set of macros {**AutoOpen**, **Payload**}.

What will remain in the document will be the set of macros {**AutoClose**, **NewPayload**}. However, it is perfectly possible to construct these macros in such a way, that this set forms a third, different virus — let us call it **Snafu**. In fact, this third virus will be created by the anti–virus program when it attempts to remove what it believes to be the virus **Foo** from a document infected with the virus **Bar**.

One could argue that the virus **Bar** actually consists of two viruses — **Foo** and **Snafu**. However, this separation of the set of macros **Bar** into the two sets **Foo** and **Snafu** can be made far from obvious — if the macros from the set **Snafu** refer to the macros in the set **Foo** in many convoluted ways (by checking their presence, copying them, executing them, etc.). Furthermore, the virus **Bar** could consist of many more macros and their division into separate viral subsets could be even more difficult, non–obvious, and even possible in several different ways.

The attack described above can be implemented in an even more devious way. The subset of macros **Snafu** could not be a virus. Instead, its macros (e.g., **NewPayload**) would just check for the presence of the macros of the virus **Foo**. If they are found to be absent, a destructive payload would be triggered. From the point of view of the user, this situation looks like this: the user runs a scanner. The scanner reports a particular known virus (maybe even claims to have identified it exactly) and proceeds to remove it. The user loads one of the 'disinfected' documents with Word and the information on the hard disk gets trashed. Naturally, the user is likely to accuse the scanner for not doing its job properly. In this case it is even more difficult to argue that the document is infected with two macro viruses — **Foo** and **Snafu** — because **Snafu** is not a virus by itself. It is not even a Trojan horse — because its macros are present in many infected documents. It is simply a part of a new virus — **Bar**. However, there is no reliable way for a scanner which does not know the Bar virus to figure out that the document is infected not by the **Foo** virus but by the **Bar** virus. The attack is very easy to implement by taking one of the 'popular' (i.e. widespread) macro viruses like **WM/Concept.A** or **WM/Wazzu.A** and 'booby–trapping' it in the way described above.

Only partial solutions of this problem seem possible. The scanners should have an option to remove all macros present in the infected documents — and the users should be educated that this is the only secure way to disinfect macro viruses. It would be helpful if the virus–specific scanner is combined with some kind of heuristic analyzer and, whenever a document containing a known virus is found, all 'suspicious' (i.e., able to copy themselves) macros should be removed from it. Unfortunately, as explained in [Bontchev96], there are many ways to attack the heuristics and force them to cause false negatives — i.e. to miss viral macros.

4.3. The Importance of Identifying the Macro Names

At first glance, it seems that it would be sufficient for a macro virus scanner to identify only the bodies of the macros belonging to a virus — without paying any particular attention to their names. True, renaming

a macro can convert a macro virus into a non–virus. For instance, the **WM/Wazzu.A** virus consists of a single macro named **autoOpen**. If it is renamed to something else, e.g., **ButoOpen**, it will become unable to replicate (even if executed manually — because the macro body addresses itself by the name "autoOpen") — that is, it will stop being a virus.

However, such a distinction seems rather superficial. Firstly, renaming the macro back is an operation which could easily be done by the user — so, it is worthwhile warning him/her that the document contains macro(s) which can too easily be converted into a replicating macro virus. Secondly, the macro names seemingly belong to the environment of the virus — not to the virus itself. As an analogy, an extension–priority companion DOS virus will stop replicating if the extension of the file it resides in is changed from COM to something else.

Unfortunately, the things are not as simple as they look at a first glance. Consider the following example ([Chess97]):

A macro virus, **Foo**, consists of two macros — **AutoOpen** and **Bar**. The macro **AutoOpen** copies itself to the document (or global template) being infected. Then it checks for the presence of the macro **Bar**. If the macro is present, it is copied as well; otherwise a destructive payload is triggered. So far — nothing unusual. The tricky part is that the contents of the macro **Bar** is identical to that of the **autoOpen** macro of the **WM/Wazzu.A** virus.

Now, if a scanner which does not know the virus **Foo** and does not pay attention to the names of the viral macros encounters a document infected with this virus, it is presented with Igor's Problem. It will 'identify' the virus **WM/Wazzu.A** and will remove the macro **Bar** — therefore leaving the macro **AutoOpen** which is perfectly capable of replicating on its own. So, the scanner has created a new virus. On the other hand, for a scanner which insists on identifying the macro names as well as the bodies of the viral macros will not have such a problem in the situation described above. It will report the document as infected (since the body of a known viral macro has been found in it) but will claim that it contains a new virus — because the macro name does not match and there are additional macros in the document.

5. VBA5 Identification Problems

The advent of Office 97 and a new macro programming language common for its applications, VBA5, brought us a whole lot of new macro virus identification problems. While most of them are not as complex as the difficult macro virus identification problems described in the previous sections, the anti–virus producers should be aware of them nevertheless. Most of these problems are caused by the fact that existing WordMacro and ExcelMacro viruses are automatically rewritten in the VBA5 language when a document containing them is opened with the respective Office 97 application. Unfortunately, as we shall see, this conversion is far from being straightforward, logical, bug–free and unambiguous.

5.1. Empty Lines

As it turns out, the converters to VBA5 — both from WordBasic and from VBA3 — add one empty line at the beginning of the program when converting it. By itself, this is not so bad. Unfortunately, Excel97 contains a converter in both directions. That is, when it opens an Excel95 workbook containing VBA3 modules, it converts them into VBA5 modules. However, it also allows Excel97 workbooks to be saved in Excel95 format — and, unlike Word97, then it 'downconverts' the VBA5 modules in the workbook to VBA3 modules. (The converter of Word97 simply ignores the VBA5 modules when saving a Word97 document in Word 6.x/7.x format.)

In practice, it means that you can take a VBA3 module, convert it into a VBA5 module — this adds one blank line at its very beginning. Then you can convert this VBA5 module back to a VBA3 module — the empty line is preserved. If you now convert the resulting VBA3 module to a VBA5 module again, another blank line will be added at its beginning. In the case of a macro virus (e.g., **XM/Laroux**) and an organization which is just switching to Office 97 and still has lots of Office 95 machines, such 'upconversion/downcon-

version' loops can be performed many times (because the users would want to save the documents in the old format — in order to keep them compatible with the machines which have not been upgraded yet) — therefore, adding many blank lines at the beginning of the virus. Yet this is still the same virus. Therefore, anti–virus programs should ignore the empty lines when identifying VBA (3 or 5) viruses.

Or, at least, it seems that they should ignore them in the case of Excel viruses and if they are at the beginning of the virus. Unfortunately, the situation is a bit more complicated than that.

The first sign that something else was amiss was brought by the `W97M/Gambler.A` virus — a native (i.e., not the result of 'upconversion' of an existing `WM` virus) virus for Word97. This virus contains several user forms, designed to spoof the Tools/Macro dialog box and provide some elementary form of stealth. When we replicated it, we noticed that the checksum of the code in the streams containing the user forms was different in the different replicants. As it turned out, each time the virus replicated, one blank line was inserted in the code of one of the user forms. Consequent replications resulted in additional blank lines being inserted. Worse, the lines were inserted not at the beginning of the code but somewhere in the middle — between the definition of the form and the code implementing the procedures designed to handle the different events (e.g., mouse clicks) associated with the different parts of the form. Why this happens is beyond our understanding. Ask Microsoft.

Anyway, it seems that it would be wise if the empty lines are ignored when identifying a VBA macro virus — regardless of whether it is written in VBA3 or in VBA5, regardless of whether it is an Excel or a Word97 virus, and regardless of where the empty lines are in the code of the virus.

5.2. White Space

Another problem of the identification of the `WM` viruses 'upconverted' to `W97M` viruses is caused by the way the converter treats white space in general and tabulation characters in particular. The first report that something is fishy came from Dmitry Gryaznov — an anti–virus researcher working for Dr Solomon's ([Gryaznov97]). According to him, the first generation of the `W97M/Appder.A` virus (i.e., produced immediately by the converter; before the new virus has had the chance to replicate) was somehow different from its replicants.

Careful examination revealed that the difference was caused by an operator which contained a tabulation character and an apostrophe–style comment at its end. This prompted us to research how WordBasic macros containing tabulation characters in different places are upconverted to VBA5 form. The results were quite interesting.

It should be noted that the tabulation characters do not exist in VBA5. If you press the Tab key while editing a VBA5 program, a number of spaces is inserted instead. The precise number of spaces inserted depends on the current position of the cursor and on the contents of the Tools/Options/Editor/Tab width setting of the VBA5 Editor. However, tabulation characters can be freely used in WordBasic — and often are used to indent lines.

Obviously, when converting the WordBasic programs to VBA5, the converter has to do something with these tabulation characters. What it does is quite logical — or at least it seems so at first glance. The converter takes the current setting of the 'tab width' field described above and uses it to convert the tabulation characters in the corresponding number of spaces — so that the look of the program (i.e., its indentation) is at least approximately preserved.

Unfortunately, when `WM` viruses containing such tabulation characters are upconverted, this means trouble. In particular, it means that machines with different setting of the tab width field will produce different `W97M` viruses from one and the same `WM` virus — if this virus contains any tabulation characters used as line indents! Furthermore, the user might even not know the contents of this setting — or even that the setting exists at all. Clearly, it will be too much of an inconvenience if all these `W97M` viruses are to be considered different. Therefore, they have to be considered

one and the same virus. That is, the indentation of the lines has to be ignored when identifying a viral VBA5 module — because we often do not know whether its originating **WM** virus had contained any tabulation characters used as line indents.

But this is not all. As it turns out, tabulation characters can be used not only as line indents. The only good news is that there are not that many other places where they can be used. In most cases, if the user inserts extra white space in the middle of an operator, both WordBasic and VBA5 will throw it out automatically. The WordBasic editor throws it out when the macro editing window is closed (thus, it becomes apparent that the extra white space has been removed the next time the macro is opened for editing), while the VBA5 Editor throws it out when the cursor leaves the editing line (thus, the change becomes apparent immediately). For example, the line

```
x     =     2     *     2
```

is automatically converted (both by WordBasic and VBA5) to

```
x = 2 * 2
```

But not always. There are a few exceptions, described below.

Firstly, the white space is preserved in the front of the apostrophe–style comments. That is, the lines

```
x = 2 * 2 ' This is a comment
```

and

```
x = 2 * 2  ' This is a comment
```

generate different code. In VBA5, the position of the beginning of the apostrophe–style comment is contained in the first operand of the "apostrophe–style comment" p–code instruction.

Secondly, the white space is preserved after the ":" operator. That is, the lines

```
x = 2 : y = 4
```

and

```
x = 2 :    y = 4
```

generate different code. In VBA5, the position of the beginning of the second operator is contained in the argument of the ":" p–code instruction.

Thirdly, the white space is preserved in the **Dim** statements before the **As** keyword. That is, the lines

```
Dim x As Integer
```

and

```
Dim x      As Integer
```

generate different code. This case is a bit more complicated than the previous two. It seems that VBA5 can use two different **Dim** p–code instructions — one for "**Dim** without spaces before the **As**" and another for "**Dim** with spaces before the **As**". The second p–code instruction contains one additional operand, containing the position of the **As** keyword on the line.

Fourthly, white space can be used when indenting the different parts of a VBA5 line which is split into multiple lines (with the "_" character at the end of the line indicating the split point). This has a direct equivalent in WordBasic (where "\" is used as a line continuation character) and the subparts of the split line can be indented with tabulation characters. Therefore, the corresponding parts of the VBA5 p–code instruction for line continuation should be skipped when computing the checksum of the VBA5 modules.

Since all the cases described above have direct equivalents in WordBasic, and since their WordBasic equivalents can contain tabulation characters as part of the white space, this means that such lines upconvert differently depending on the tab width settings of the VBA5 Editor. Therefore, the white–space–related operands of the p–code instructions mentioned above should be ignored when identifying VBA macro viruses.

Another white–space–related problem is presented by the trailing blanks in the VBA comments. To put it simply, VBA3 allows trailing spaces in comments, while VBA5 does not (and removes them). The following example illustrates this.

1) Start Excel 95, create a new, blank workbook, and insert a module sheet in it.

2) In the module sheet, enter the following short program:

```
' The following comment has trailing spaces:
'
Sub Test()
   MsgBox "This is a test.", vbOKOnly +
   vbInformation, "Test"
End Sub
```

On the second line, enter several spaces after the initial apostroph. After the cursor leaves the line with the trailing spaces in the comment, if you return to that line and press the End key, the cursor will be positioned immediately after the apostroph — as if there are no trailing spaces. Fear not — instead, save the workbook in a file and examine the generated p–code. You'll see that the trailing spaces (exactly as many as you entered) *are* there.

3) Exit Excel 95. Start Excel 97 and open the document you have created on the previous step. The VBA3 macro in it will be upconverted to a VBA5 macro. Use Excel 97 to save the upconverted document into another file — but save it in Excel 95 format (*not* in Excel 97 format), thus 'downconverting' the macro to VBA3 again.

4) Examine the p–code of the new file. You'll see that not only one empty line has been inserted at the very beginning, but also the trailing spaces from the comment are gone.

Therefore, the trailing spaces in comments have to be ignored when identifying a VBA virus.

In fact, there is another case in which white space is used — however, it turns out that is does not cause any macro virus identification problems. In particular, tabulation characters can be used inside a string literal. However, the converter handles these cases quite smartly — it finds all such characters in the literal strings and replaces them with `Chr$(9)`. For example, the WordBasic line

```
x$ = "This is a Tab ->    <-"
```

is upconverted to the following VBA5 line:

```
x$ = "This is a Tab ->" + Chr$(9) + "<-"
```

5.3. Ambiguous Upconversion

Since the above two problems force us to ignore the blank lines and the white space (used as indentation or not) when identifying a VBA virus, it also creates the very real possibility that two (or more) different `WM` viruses would upconvert to one and the same `W97M` virus. For instance, our research indicates that *three* different `WM/Wazzu` variants — `Q`, `W`, and `AD` — would upconvert to one and the same `W97M/Wazzu` virus. (If an early beta version of Word97 is used, of course — the release version recognizes these `WM/Wazzu` viruses and refuses to upconvert them.) Similar ambiguous upconversions can certainly happen with some other `WM` viruses as well.

In cases like this, it is not clear how exactly to name the upconverted virus. Normally, the upconvert of a particular `WM` virus is assigned the same name/variant combination as the originating virus and the `W97M` platform prefix. For instance, the virus created by upconverting the `WM/Wazzu.A` virus into a VBA5 virus is named `W97M/Wazzu.A`. However, the fact that the upconversion can be ambiguous breaks this simple naming scheme.

In order to 'fix' it, we propose that in those cases when the upconversion is ambiguous, the virus created by it is assigned the lowest of the variant names which could have created it. That is, in the example given above, the resulting `W97M/Wazzu` virus is named `W97M/Wazzu.K` — and not, say, `W97M/Wazzu.Q` or `W97M/Wazzu.AC`.

5.4. Letter Case in the Identifiers

As mentioned elsewhere in this paper, all VBA (both VBA3 and VBA5) modules in a file share a common pool of identifiers (variable names, procedure names, etc.). But that is not all. In addition, when a new module is addressed, a module which uses the same identifier as one of the existing modules, no new identifiers are added to the common pool of identifiers. Problem is, when deciding whether the new identifier is 'the same' as one of the existing ones, the letter case of the said identifier is ignored.

This can have some rather puzzling effects. For instance, if one creates a VBA module containing the line

```
fOO = Bar
```

then creates another module, containing the line

```
bAR = Foo
```

and opens the first module for editing again, its contents will look like this:

```
Foo = bAR
```

which is not quite exactly what the user originally entered. In fact, these two lines can be typed in one and the same module — and, as soon as the second one is entered, the letter case in the first one will change.

How is this relevant to the subject of macro virus identification? Well, it means that when the checksum of the module is computed and the pointers to the common pool of identifiers are resolved (and they have to be resolved, because some quite different programs can compile to one and the same p–code with just the identifiers pointed by it being different — so, if these pointers are not resolved, the identification of the virus will not be exact and could even cause false positives), the letter case of the identifiers has to be ignored.

Unfortunately, in the case of VBA5, this introduces another problem. Ignoring the letter case of the identifiers is usually achieved by unconditionally converting them to either upper or lower case before comparing them. However, in VBA5, the identifiers can contain foreign (i.e., non–ASCII) characters. For instance, `Français` is a perfectly valid identifier there.

Unfortunately, there is no reliable and easy way for converting these foreign characters to upper or lower case. The Windows APIs contain some functions for this purposes (e.g., AnsiToUpper). However, they are available only to Windows applications (e.g., a DOS virus scanner cannot use them). Furthermore, they work correctly only under the right language version of Windows. For instance, the function AnsiToUpper will convert correctly Czech characters to their uppercase equivalent only under the Czech version of Windows (or if the Windows locale is set to Czech).

The implications of this problem are that a scanner which relies on such functions when identifying VBA5 viruses could miss a virus containing identifiers with foreign characters if the document containing it is scanned under the wrong version of Windows — a clearly inacceptable situation, especially having in mind that the virus itself will very probably have no problems replicating in that environment.

The best solution of this problem that we could think of is to convert all non–ASCII characters in the identifiers to some common ASCII character (e.g., "_") before converting them to upper case and checksumming them. This solution can result in an identification algorithm which is unable to distinguish between too very slightly different viruses — but this is much better than missing a virus completely.

5.5. Other VBA5 Identification Problems

Unlike WordBasic, VBA has not one but two levels of granularity. At the first level are the modules — roughly equivalent to the macros in WordBasic. On the second level are the functions and subroutines of each module. Those of them which are declared as `Public` (the default declaration) will be visible in the Tools/Macro dialog box.

Two levels of granularity of the functional components of the VBA viruses means that both Richard's and Igor's Problems can occur on two levels. And they *will* occur — as soon as VBA viruses are written which can 'snatch' not just VBA modules from other VBA packages but also separate functions or subroutines from these modules.

The solutions of the two Problems will look similar (in the sense that the solution of Igor's Problem will be similarly non–existent); they will simply have to be applied on two levels too. In particular, it means that it might make sense to change the definition of a macro virus and consider it as a set of modules, where every module consists of a set of functions and/or subroutines — and identify the modules as sets; not as a whole. Unfortunately, this might require some significant re–design of some anti–virus products and is unlikely to happen before the reality forces it — that is, before subroutine–snatching VBA viruses are created.

6. Artificially Created Macro Virus Identification Problems

The macro virus identification problems discussed in the previous sections can be called 'natural'. That is, they arise either because of some inherent property of all macro viruses (e.g., the fact that they are not stand–alone programs but consist of sets of macros) or because of some design flaw of the macro programming language and its environment. In this last section we shall consider some problems which can be created artificially by the virus writers — in order to make the identification of their viruses more difficult. Most of these problems are related to the different forms of 'polymorphism' — i.e. the capability some macro viruses have to modify themselves during replication.

We shall keep the description of the different methods for implementing polymorphism intentionally brief — because we want to limit the usefulness this paper could have for the virus writers. Similarly, we shall only outline the basic ideas for handling these problems. Unfortunately, a detailed public description of our solutions found so far would make them easily attackable. Therefore, at least until we can come up with more robust solutions, we will have to rely on 'security through obscurity' — regardless of how unreliable this protection is.

6.1. Insertion of Do–Nothing Lines

The simplest form of polymorphism in the macro virus world is implemented by inserting, at random place of the virus code, 'do–nothing lines'. They are WordBasic (or VBA) operators which do not have any impact on the working of the virus and can be inserted anywhere in its code without its operations being harmed (or even modified) in any way. Most often those do–nothing lines consist of randomly generated comments — but they can be almost anything, like variable assignments, function calls to empty functions, and so on.

The easiest way of handling this form of polymorphism is to have the anti–virus recognize the do–nothing instructions used in the different polymorphic viruses and skip them when identifying the macros of the virus. Unfortunately, this approach has two serious drawbacks.

Firstly, there is an almost endless choice of do–nothing operators. As a consequence, different polymorphic viruses can (and usually do) pick different operators to use as a 'filler'. If a scanner recognizes a set of do–nothing operators (used in the polymorphic viruses known to that scanner) and a new polymorphic macro virus appears, one which uses a different do–nothing operator, it will mean that the scanner will have to be updated in order to make it capable of handling the new operator. We would like to emphasize that it is the *program* of the scanner that will have to be updated — merely updating its database of virus detection information will usually prove insufficient. This requirement will hamper the attempts to make the scanner entirely driven by its database. Worse, in most cases it will mean that, since the identification algorithm has been changed to skip some additional operator(s), all previous entries in the database which rely on this algorithm will have to be revised and possibly updated. This process is rather time–consuming and prone to mistakes.

Secondly, the virus writers could target this approach directly. For instance, let us suppose that a given polymorphic macro virus uses the following block as a do–nothing filler:

```
If <condition> then
      <operator1>
      <operator2>
      :
      <operatorN>
EndIf
```

where **<condition>** is never true. If a macro virus scanner decides to handle this kind of polymorphism by skipping the whole **If...EndIf** block, another virus can be written which consists entirely of only the operators **<operator1>**, **<operator2>**,... **<operatorN>** — and such a virus will be completely skipped by the scanner's algorithm.

In order to handle these problems, the scanners should have a more flexible, virus–specific algorithm for ignoring the do–nothing operators, and this algorithm should be entirely controllable by the database of the scanner.

6.2. Variable and String Modification

Another trick which is often used in the polymorphic macro viruses is to have the virus change randomly the names of the identifiers (usually — used as names of variables) it uses. Of course, this random modification is done consistently — in the sense that one and the same identifier is always replaced with one and the same randomly generated name. Similarly, the virus could also change in a random way the contents of some literal strings used in it.

These two tricks can be countered relatively easily. In order to handle the random literal string modification, the literal strings used in polymorphic viruses can be simply ignored when identifying the macros (or modules) of the virus.

Handling the random identifiers trick is only a little bit more complicated. The scanner should build a table of the identifiers used in the macro or module and then replace each identifier with some arbitrary number (e.g., 001, 002, 003, etc.), so that one and the same identifier is always replaced with one and the same number and different identifiers are always replaced by different numbers. Once the virus body has been 'canonized' this way, a checksum of it can be computed — and it will be always one and the same, regardless of how the virus changes its identifiers during replication.

6.3. Line Swapping

Another possible kind of polymorphism can be implemented by swapping around lines of code, the exact sequence of which is not important for the correct working of the virus. To the best of our knowledge, currently no known macro virus uses this trick — but it is very easy to implement. Fortunately, it is just as easy to counter.

In order to handle it, a macro virus scanner which uses exact virus identification should compute the checksum of the virus macro (or module) on a line–by–line basis. Then, the partial checksums of each line should be XOR–ed together, to form the final checksum of the macro (or module). When computed this way, the checksum does not depend on any line swappings.

6.4. Commenting and Uncommenting Lines

The next trick which can be used in polymorphic viruses consists of adding and removing comments in the front of some lines of the virus body. This achieves the effect of changing completely the internal representation of these lines — and, therefore, of the virus body as a whole. Of the known polymorphic macro viruses, the **WM/Dakota** viruses use this kind of polymorphism to a great extent. They keep the whole contents of one of its macros in commented form (which is additionally interspersed with a random character, it order to make it look different in each replicant). At runtime, the virus removes the comments, so that the macro can execute and perform its tasks.

The simplest way of handling this method of polymorphism is simply to ignore the comment lines. Unfortunately, this has the drawback of reducing the exactness of the identification.

6.5. Encryption

The encryption of the macro bodies used by Word when a macro is copied as 'execute–only' is trivial (XOR with a byte–long key; and the key itself is stored in the document) and most macro virus scanners on the market are capable of penetrating it easily. Furthermore, the author of the virus has no control on the particular encryption key used and cannot change it. Finally, this kind of encryption is not available in Word97 — there the protected projects cannot copy their modules around. Therefore, this kind of encryption is not suitable for implementing polymorphism.

However, it is possible for the virus writer to implement some additional kind of encryption of the virus body. This is done by treating the lines of the virus body as text strings and applying some kind of scrambling string manipulation on them. This is a relatively slow process, but the virus writers are rarely interested in producing effective code. The viruses `WM/Slow` and `WM/UglyKid` are examples of polymorphic macro viruses which use such custom encryption.

There are several possible ways of handling this method of polymorphism. Unfortunately, most of them are dependent on the particular implementation of the polymorphic mechanism in the virus — that is, they are virus–specific. The easiest way is simply to treat the encrypted part of the virus as a variable literal string and ignore it. This method works nicely with `WM/Slow`. Unfortunately, it has the drawback of reducing the exactness of the identification.

A more sophisticated method consists of implementing at least a partial WordBasic (or VBA) emulator and emulating the part of the virus which performs the decryption at runtime — until the encrypted part is decrypted, after which it can be identified. This method is rather difficult to implement. However, we expect that in the near future more and more polymorphic viruses will appear, the proper handling of which will require it.

6.6. Parasitic Infection

The currently existing macro viruses use infection techniques roughly analogical to those of the overwriting and companion viruses in the DOS virus world. However, as explained in [Bontchev96], there is no practical reason why truly parasitic infection cannot be implemented as well. Such parasitic viruses will not be self–contained; instead, they will search for other macros in the documents they try to infect and will modify those macros to include either the virus macros completely, or at least calls to them. User macros are encountered relatively rarely, so a virus which uses this infection technique alone is unlikely to be able to spread successfully. However, this technique could be combined with more conventional ones which the virus could resort to in the absence of infectable user macros. Such a combined infection strategy would both ensure the spread of the virus and make its identification (and, occasionally, removal) rather difficult.

None of the known macro viruses uses parasitic infection intentionally. However, some of them (e.g., `WM/Goggles`) can modify some macros with special names (e.g., `FileSaveAs`), if they already exist in the document it attempts to infect. Fortunately, since this is not done intentionally (obviously, the virus author has simply failed to foresee this side effect of the infection strategy he has chosen for his virus), when it happens the existing macro is simply damaged and the virus fails to work correctly. However, a proper implementation of the parasitic infection technique is rather easy to imagine.

In order to identify such viruses correctly, a macro virus scanner has to resort to similar methods as those used to identify parasitic viruses in the DOS virus world. Instead of using a single checksum for the whole body of the viral macros, the scanner will have to carry in its database some kind of a map of the virus, consisting of ranges of bytes to checksum. Furthermore, the scanner will have to have some kind

of mechanism designed to follow the instructions which transfer control to the virus body inside the user macro — similar to the mechanism used by DOS scanners to follow the initial JMPs and locate the entry point of a virus in a DOS program.

7. References

[Bontchev96] Vesselin Bontchev, "Possible Macro Virus Attacks and How to Prevent Them," *Computers & Security*, 1996, Vol. 15, No. 7, pp. 595–626.

[Chess96] David Chess, personal communication.

[Chess97] David Chess, personal communication.

[Ford96] Richard Ford, personal communication.

[Gryaznov97] Dmitry Gryaznov, personal communication.

[Muttik96] Igor Muttik, personal communication.