AV Testing Exposed

(revision 1.1)

Peter Košinár, Juraj Malcho, Richard Marko, David Harley

ESET, Aupark Tower, 16th floor, Einsteinova 24, 85101 Bratislava, Slovakia

Tel. +421 (2) 322 44 111; Fax +421 (2) 322 44 109;

{kosinar, malcho, marko}@eset.sk, david.harley@eset.com

Abstract

As the number of security suites increases, so does the need for accurate tests to assess detection capability and footprint, but accuracy and appropriate methodology gets harder. Good tests help consumers to make better-informed choices, and vendors to improve their software. But who really benefits when vendors tune products to look good in tests instead of maximizing their efficiency on the desktop?

Conducting detection testing may seem as simple as grabbing a set of (presumed) malware and scanning it. But simplicity isn't always easy. Aspirant detection testers typically have limited testing experience, technical skills and resources. Constantly recurring errors and mistaken assumptions weaken the validity of test results, especially when inappropriate conclusions are drawn, as when likely error margins in the order of whole percents are ignored, translating into exaggerated or even reversed ranking.

We examine (in much more detail than previous analyses) typical problems like inadequate, unrepresentative sizing of sample sets, limited diversity of samples, the inclusion of garbage and non-malicious files (false positives), set into the context of 2010’s malware scene.

Performance and resource consumption metrics (e.g. memory usage, CPU overhead) can also be dramatically skewed by incorrect methodology such as separating kernel and user data, and poor choice of "common" file access.

We show how numerous methodological errors and inaccuracies can be amplified by misinterpretation of the results. We analyse historical data from different testing sources to determine their statistical relevance and significance, and demonstrate how easily results can drastically favour one tested product over the others.

Introduction

This paper aims to answer the basic question – does AV testing provide an incentive for security vendors to improve the products themselves, thus enhancing computer security globally? Or is it more likely to be the exact opposite – spending too much time focused on achieving good performance in tests, responding to the expectations of users and marketing departments (who over time have become accustomed to these testing nuances and themselves prefer clearly differentiated rankings expressed by percentages and charts), rather than introducing meaningful improvements to the product for actual use in the battlefield?

We’ll first focus on the most common area of problems in testing – the tests of detection of malicious software – and see how useful their results are (or are not). We describe a general approach which can be used for evaluating detection and which we would expect to provide more useful results than the ones generally employed by current tests. Afterwards, we dive into the area of statistics and see what can be said about collections used in current tests. The second section looks at certain procedural issues, related to new methods employed by antivirus products, and the effect they have on the results. Following that, we move on from detection testing to look at the other commonly tested factors - performance and resource usage.

Detection testing - the theory

There is little doubt that the detection of malicious programs is the most important property of software describing itself as anti-virus or anti-malware. It is therefore one of the most commonly tested and mis-tested features of such software – especially since it seems to many that such testing is very easy, even trivial, to conduct. After all, what can be difficult about:

- Collecting a large group of (presumed) malicious files
- Running the tested product(s) against those samples and seeing how many of them they detect
- Putting the results into a nice spreadsheet?

Even a properly trained monkey could do that... or could it? Let's take a deeper look at how messy things can get.

Collecting a large set of malicious files

How large should the sample set be? These days, we're seeing tens of millions of unique malicious files each year. Intuitively, in order to achieve good coverage of the “real” world, the number of tested samples should not be significantly lower (as in, three orders of magnitude or more). Of course, there are specialized types of tests in which the nature of the test implies that the size of the collection has to be considerably smaller – like, the one for obtaining VB100 award – but those tests concentrate on different type of problem, where the total population is much smaller than the totality of malicious objects in the whole world.

Most of the time, it's very easy to collect quite a few malicious files, simply by visiting known malicious websites and allowing the computer to get infected; or by plugging it into the Internet without applying the patches that a cautious user applies in order to counter known security vulnerabilities. Respectable antivirus-testing organizations also usually get actual samples from mainstream AV companies as part of the sample-sharing initiatives that enable the industry to offer its customers better protection. Yet, from time to time, some “creative” testers decide that files provided by a tested vendor could be engineered to bias the results in favour of that particular competitor. Those who do not learn from mistakes of others are bound to repeat them – either by reinventing something along the lines of the infamous Rosenthal virus simulator, or even by producing their own malicious programs. The latter has always been a matter of heated discussion [1], while the former is just an outright demonstration of not understanding how antiviruses actually work these days – it was an attempt to simulate known viruses by abstracting a “signature” on the absurd assumption that every antivirus package uses the same signature.

Collecting a set of malicious files always generates a question as to how to winnow the chaff from the grain. Most sources of “malicious” files are not pure – there is always some proportion of files which are corrupted, non-functional (non-viable), belong to the grey zone between malicious and clean, or are completely benign (innocent) files. In an ideal world, such files would be identified by the tester at the beginning of the process, before executing even one of the antiviruses in the test environment. Unfortunately, that would require the tester to be more skilled in determining what is or isn’t malicious than all the tested products – in which case the tester might be better employed producing a better security product in his own right.

Thus, the usual, rational mode of operation is reversed – first, the set of files is collected, including such a proportion of inappropriate samples. These files are run past the tested products and the burden of removing the chaff is transferred to the antivirus vendors who are forced to “defend” themselves by pointing out which files they don't detect should not have been a part of the set in the first place. In other words, it's a model based on presumption of guilt – unless you can prove you really should not be detecting the files, you're guilty of not detecting them.

Is that a good approach? Well, it certainly is very convenient from the tester's point of view. It is also very efficient in hampering the antivirus vendors' real work – especially since the percentage of unsuitable files is generally quite high – often reaching percentages in double figures. No, that doesn't mean that the rest of the samples are grain (i.e. valid) – only that they haven’t been proven to be invalid. The number of man-hours vendors have to spend on this is getting higher and higher, and the law of diminishing returns is very applicable in this case. A proportion of the problematic files are usually very easy to identify. Others are more difficult to identify, but still manageable by the use of advanced technology. Yet other files might really require the intervention of an experienced human malware analyst to determine whether they actually are malicious or not – after all, if the problematic files were that easy to identify, they would have been detected in the first place.

Having the proper collection of infected files is, of course, only a part of the test. Unless one also takes into account the quantity of false alarms produced by the product when run against a collection of clean files, the test is going to be essentially useless, regardless of the quality and size of the malicious collection. After all, even a simplified version of the Perfect Antivirus by Dr. Solomon (Echo %1 is infected by a virus!!!) would win such a test, regardless of its contents. Yes, an antivirus which declares everything to be malicious is going to get full marks in a test that doesn’t care about false positives.

The issues we’re discussing can be characteristic of experienced and renowned testers – so this is more of an example of how good things can be, than how bad. In general, the more tests are there, the more they look like a Denial-of-Service attack on the antivirus vendors.

For now, though, let's assume that the tester was able to collect a set which is reasonably free of inappropriate or innocent files and that he has unleashed the antivirus products under test on that collection.

What do the test results really say?

The obvious answer is – they tell us how many files from the collection a particular product detected under the conditions of that particular test. Everything else is a matter of extrapolation and interpretation. If one doesn't look at the complete picture (which means having access to all the malicious files that existed at the time of the test), *any* interpretation of the results along the lines of “This product detects that X percent of all the malicious programs that existed” is going to have a smaller or larger error margin, and ascertaining that error margin is largely guesswork. Predicting the future success rate of a product is even less of an exact science.

Igor Muttik wrote about this problem almost nine years ago [2], yet the results still seem to be largely unconsidered or misunderstood by quite a few testers. We'll look at this problem through 2010 eyes.
In most cases, the final test results are presented in a form which is easy to comprehend for a layman – the detection rates of products are written as percentages and put into a spreadsheet next

to the names of tested products. Then, one click of the “Sort!” button reveals who gets the first prize. This is sometimes followed by further “simplification” of the results –for example, by removing the actual detection rates and only retaining the final order of the products.

Obviously, the less information the result sheet provides, the more opportunity is left for the reader or publisher to (mis-)interpret the conclusions. In general, it seems that the less information is given, the more room for doubt there is about the tester’s qualifications and – a report stating that “Vendor X is the best” without any supporting evidence is more likely to indicate an incompetent tester than a detailed report describing the methodology, source of information and detailed results. AMTSO (the Anti-Malware Testing Standards Organization) has published several documents on these issues [3,4], ranging from best practices and guidelines for various types of testing to discussion of some of the ethical questions related to such tests. We won't be going deeper into the problem of presentation of the results here, but rather concentrate on the most important question:

What is the best way to measure detection rates?

The basic question we need to ask is – what do we actually mean by “detection rate”? The answer is more complicated than it might seem at the first glance. The usual mechanism of testing detection by taking a “snapshot” of product detection, by scanning a static collection of files was applicable enough in earlier times, when the state of the threat landscape changed very slowly compared to today’s threatscape. It's been suggested that one necessary ingredient is the temporal information [5] (how results change over time), especially when we consider the rapid updates provided by cloud-based technologies (but more on that later). Another important aspect concerns the users (or rather their computers) – some might be more prone to certain types of attacks than others. For example, servers are less likely to be compromised via browser vulnerabilities, while computers running Mac OS are themselves immune from purely Win32-based viruses (unless, of course, they also host some form of Windows emulation, and ignoring (for our present purposes) the issue of heterogeneous malware transmission). .

In order to avoid ambiguity, we'll present the proposed method in mathematical notation. This is a synthesis of commonly used approaches that already exist, and within appropriate limits, these approaches can be represented simply in this mathematical form. It is, however, capable of demonstrating a wider range of scenarios.

We'll be considering three (non-empty) sets – a set U of users for whom this particular test should be relevant, a set of tested products P and a set A of malicious attacks. For our purposes, the *attack* is an attempt to subvert user's computer in undesirable way (for example, stealing his information, performing denial of service, or one of many other possibilities). We only consider attacks which would succeed, if the user didn't use any product from the set P – thus, visiting a website exploiting a vulnerability specific to Internet Explorer is not considered here as an attack when the visitor is using a text-based browser like Lynx. Also, protecting the computer from threats that have no relevance is not something that should be rewarded – it's more important to protect from real dangers. Of course, it’s justifiable to declare the attack to be successful if the malicious program would have “visited” the computer – thereby reducing the concept of “attack” to the more common description.

There will always be one-to-one correspondence between attacks and files (or samples) which caused them: if the scenario consists of a chain of events caused by different files, they will be considered separate attacks, as each can (and most likely should) be detected and prevented. For example, visiting a website where an injected IFRAME is followed by access to Javascript which in turn loads a malicious SWF file will be considered as three separate attacks. This is because at each stage, one component could be replaced by another, undetected by the current version of a product under test.

Finally, without loss of generality, we'll treat the time t as discrete and every event as happening inside the time-span of length *T*.

Definition: Indicator function *Incident(u, a, t)* is equal to 1 if user *u* was subject to attack *a* at time *t* and zero otherwise. Indicator function *Detect(p, a, t)* is equal to 1 if product *p* would detect attack *a* at time *t*.

The first function captures the notion of "*when* was *who* attacked by *what*". We'll assume that each user was attacked at least once. We're also assuming that the products work consistently with respect to the users – if the attack is detected by the product for one user at some point in time, it will also be detected for other users at the same time. Note that we're not implying consistency over time – a product might stop detecting some type of attack or detect it only "by chance" every now and then. [6] Having established these two concepts, we're ready to define the functions we're interested in:

Definition:

$$Prevalence(a,t) = \frac{1}{|U|}\sum_{u} Incident(u,a,t)$$

Prevalence tells us how large a proportion of users was affected by a particular attack at some point in time – the higher the proportion, the more often the attack occurred. Naturally, prevalence of a particular attack varies with time, in most cases starting with quick acceleration followed by slower decline and a long tail descending to a level almost indistinguishable from zero.

Definition:

$$Success(p,u) = \frac{\sum_{a,t} Incident(u,a,t).Detect(p,a,t)}{\sum_{a,t} Incident(u,a,t)}$$

$$AvgSuccess(p) = \frac{1}{|U|}\sum_{u} Sucess(p,u)$$

The first function describes the success rate of product *p* when stopping attacks for user *u*. The second function represents the average success rate among all users – that is, the expected value of the answer to the question "How successful has product *p* been for user *u*" when choosing user *u* randomly. Note that this approach treats all users equally – if one user gets attacked eight times and the product stops six of the attacks (success rate 75%) and the other is attacked just twice, but the product fails to stop either of the attacks (success rate 0%), the average success rate is 37.5% rather than the alternative "stopped 6 attacks out of 10" = 60% score. The reasoning behind our choice is that it's the user who is interested in the results of tests – so it's better to assume a uniform choice of user, rather than a uniform choice of attack. As a bonus, if the attacks actually are distributed uniformly, our value AvgSuccess will accord with the other approach.

In simple terms, the value of *AvgSuccess(p)* estimates how well product *p* would help Joe User avoid becoming a victim of a successful attack, and thus might be more useful for him than the raw detection count. There is one important disadvantage, though – it's almost-a Catch-22 situation when obtaining the values of the *Incident* function. In order to know that there was an attack, there has to be some means of *detecting* the attack. But if one of the products to be tested is used to find out whether there really was an attack in the first place, it will score a guaranteed point in the final conclusions for its *Detect* function too – resulting in a 100% detection score for this product in the end, while ignoring the possibility that it has generated false positives! [7] For now, we'll postpone this discussion until the next section, where we deal with the practical hurdles.

It is usual to perform various simplifications in order to reduce the amount of data that needs to be collected or processed. First, there is the approach which assumed uniform distribution of the attacks among the users. This is equivalent to assuming that there is just one user who gets hit all the time. Since there might have been multiple users being subjected to the same attack at the same time in the original scenario, a common trick is to incorporate *Incident(u, a, t)* by *Prevalence(a, t)* into the Success formula (AvgSuccess coincides with Success in this case, as there is only one user):

$$AvgSuccess(p) = \frac{\sum_{a,t} Prevalence(a,t).Detect(p,a,t)}{\sum_{a,t} Prevalence(a,t)}$$

Still, the values of *Prevalence* are hard to come by: almost as hard as was the case with the *Incident* function. Thus, in many tests, this value is assumed to be equal to 1 – if the file appeared at least once, it becomes part of the test-set. The problem with this approach becomes apparent immediately – one simply needs to look a few years back into the past, when worms like Sasser were rampaging through the Internet. If a particular instance of Sasser was just one out of ten thousand files, not detecting it would amount to a negligible loss of 0.01% in the overall detection score. Yet, most people would be likely to consider a product as missing such a threat almost completely. In other words, the prevalence information is unequivocally important for producing useful results, if not critically necessary.

Even if we overlook the lack of prevalence data in most tests, the problem outlined in [2] remains – using just a smaller subset of all the malicious files for the purpose of the test produces an error margin, which might be too large for the test results to bear any actual meaning. Let's look at this in more detail!

Assume that there are *N* files in total in the test-set (in the terminology introduced above, |*A*|=*N*), there is only one user and the prevalence of all files is equal to 1. In other words, we're looking at the simplest possible case – just the count of detected files divided by the total number of files. Now, take a product which detects *D* percent of the files and run it against a collection consisting of *M* randomly chosen files from this collection. What can we expect to find?

The table below summarizes the results of a simple simulation for various selections of the parameters, showing the average number of detected files, minimum and maximum detections encountered during the simulations, and the standard deviation of the results:

N	D	M	Average	Minimum	Maximum	Std. dev
1 million	80.00%	1 000	80.03%	75.70%	83.70%	1.25%
1 million	80.00%	10 000	79.97%	78.81%	81.24%	0.39%
1 million	80.00%	100 000	80.00%	79.65%	80.37%	0.12%
1 million	80.00%	500 000	80.00%	79.88%	80.14%	0.04%
1 million	97.00%	1 000	97.00%	95.30%	98.60%	0.54%
1 million	97.00%	10 000	97.00%	96.39%	97.47%	0.16%
1 million	97.00%	100 000	97.00%	96.83%	97.16%	0.05%
1 million	97.00%	500 000	97.00%	96.94%	97.06%	0.02%
10 million	97.00%	10 000	97.01%	96.47%	97.52%	0.17%
10 million	97.00%	100 000	97.00%	96.85%	97.20%	0.05%
10 million	97.00%	1 000 000	97.00%	96.94%	97.06%	0.02%
10 million	97.00%	5 000 000	97.00%	96.98%	97.02%	0.01%

Table 1

Instead of using a simulation, it is also possible to calculate the values explicitly – although letting someone see the random number generator produce the results is often more convincing than requiring them to process the calculations. Since there are *D.N* files which are detected by that particular product and *(1-D).N* of them which are missed, the probability of the product detecting exactly *K* files from the subset of size *M* is equal to:

$$C_{N,D,M}(K) = \frac{\binom{DN}{K}\binom{(1-D)N}{M-K}}{\binom{N}{M}}$$

It's not very surprising that the expected value ("average") of this distribution is equal to (*M.D*), as can be obtained by proper manipulation of the calculation:

$$Average_{N,D,M} = \sum_{K} C_{N,D,M}(K).K = \sum_{K} \frac{\binom{DN}{K}\binom{(1-D)N}{M-K}}{\binom{N}{M}} K = M.D$$

In a similar fashion, we can calculate the standard deviation:

$$StdDev_{N,D,M} = \sqrt{D.M}.\sqrt{\frac{(N-M)(1-D)}{(N-1)}} \sim \sqrt{D.M}.\sqrt{\left(1-\frac{M}{N}\right)(1-D)}$$

From statistics we know that the interval of two statistical deviations from the average to both sides covers about 95% of the cases, assuming that the distribution isn't too irregular (which in this case it isn't). In other words, if we have one million files and test an antivirus which detects 97% of them, but we restrict our attention to a randomly chosen hundred thousand of them, the result is quite likely to fall into the interval [96.9, 97.1], but it's surely not bound to be equal to exactly 97%. It's also noteworthy that, technically, we could have ended up with a detection rate as low as 70%, if we were unlucky enough to choose all thirty thousand of the files this product does not detect. Fortunately, the probability of such a choice is astronomically small if files are chosen randomly…Deliberate selection of the "bad" choices is another issue.

The approximation of standard deviation can be used to estimate the required size of the test set. If *E* denotes the standard deviation we'd like attain, the resulting formula is

$$M = \frac{1}{\frac{E^2}{D(1-D)} + \frac{1}{N}}$$

For example, if the detection rates are specified as percentages with two decimal places, we can set *E=0.0001* divided by 2, so that the „two standard deviations" rule allows us to rest happy, with 95% confidence that the result will differ from the actual value only to the first non-significant place.
A few examples follow:

N	D	E	Calculated M
1 million	80%	0.001/2	~390 thousands
10 millions	80%	0.001/2	~600 thousands
100 millions	80%	0.001/2	~640 thousands
1 million	95%	0.001/2	~160 thousands
10 millions	95%	0.001/2	~190 thousands
100 millions	95%	0.001/2	~190 thousands
1 million	97%	0.001/2	~100 thousands
10 millions	97%	0.001/2	~120 thousands
100 millions	97%	0.001/2	~120 thousands
10 millions	97%	0.0001/2	~5.4 millions

Table 2

Interestingly, the value of M which is necessary to achieve an accuracy of one decimal place converges quite quickly. This is related to the fact that for a very large value of N, the denominator in the formula above becomes dominated by its first term, rather than the second one, as was the case for smaller N. Thus, the limiting size of the useful test-set can be expressed as

$$UM = \frac{D(1-D)}{E^2}$$

Using a number of files above this limit is not very likely to improve the accuracy of the result for a particular tested product. Naturally, this assumes that the files were chosen from the whole full set of N files without the introduction of any statistical bias; otherwise the results might be skewed in either direction.

So far, we've been looking at one antivirus. However, if there is more than one being tested, the problems can reappear, even with test-sets of such a size as described above. Using the same simulation as before, let's see what happens if two competitive products of nearly equal detection rate are tested on the same subset. The set will consist of one million of files and we'll choose a random subset of a particular size, and test both products on this subset. If the results of this test do not agree with the "true" detection rates (the ones corresponding to the whole set of files), we'll count this as an *inverse* case. Results of this simulation are summarized in the following table (each row representing an average of one thousand simulations):

Detection rate 1	Detection rate 2	Subset size	Inverse cases
97.00%	96.90%	1 000	44.90%
97.00%	96.90%	10 000	36.30%
97.00%	96.90%	100 000	8.20%
97.00%	96.90%	500 000	0.00%
97.00%	96.80%	1 000	40.70%
97.00%	96.80%	10 000	22.90%
97.00%	96.80%	100 000	0.60%
97.00%	96.80%	500 000	0.00%

Table 3

As we can see, even with 1/10 of the whole set, the results can quite often be reversed: – possibly too often for the test to be meaningful. Unfortunately, the number of inverse cases depends on the difference between the actual detection rates of the products – so if they're very close to each other, it will require a very large test-set to tell them apart and put in correct order of ranking. In particular, we might want to fit four standard deviations in between them, in order to prevent the intervals from overlapping (up to the 95% confidence of "two standard deviations", as usual). For example, an estimated 320,000 files or so is necessary to tell the 97% and 96.9% detections apart, while approximately 120,000 should suffice for 97% and 96.8%.

Last, but surely not least, there is the question of content in the set that is not-actually-malicious. The calculations described above were based on the assumption that the files in the set are actually appropriate for true positive detection. In most cases, this is not really the case – the proportion of non-functional, clean or damaged files in the set varies quite a lot between various types of tests, ranging from single figure percentages to tens of percents. Some products tend to play it safe here, working with a “presumption of innocence” and not detecting files that are not unequivocally malicious, whereas others prefer the “guilty unless proven innocent” approach and declare them malicious.

Naturally, this can pose a problem if such products compete with each other, since this results in a test of design philosophy rather than accurate detection. In order to model this and see how it affects the results, each product will be assigned a real number *J(p)* between 0 and 1 (inclusive), describing the percentage of the “junk” files product *p* detects. In our experience, this seems to be pretty much constant for each product and independent of the actual test-set. In this scenario, the original detection rate *R(p)* will need to be adjusted to

$$R'(p) = R(p).(1 - B) + J(p).B$$

Here, *B* denotes the percentage of the ballast files in the particular test.

As an example, we can look at what happens if two products with actual detection rates of 99% and 95% and *J(p)* equal to 0.1 and 0.8 respectively compete in a test whose *B* is as little as *8%*. The adjusted detection rate of the first product will be 91.88% and for the second it’ll be 93.80%. Yes, the difference is astounding – not only has the second product surpassed the first, it has done so by quite a large margin. Even to break even (pun not intended), the ballast ratio would have to be lower – about 6% at most, as can be calculated from the following general formula:

$$B = \frac{1}{1 - \frac{J(p_1) - J(p_2)}{R(p_1) - R(p_2)}}$$

We can also look at the problem from the other side. Imagine that a test includes three products – X, Y and Z. Their results from the test are summarized in table below, along with their junk-detection ratios (vendor X is from the “innocent unless proven guilty” camp, vendor Z from the other one; Y stands somewhere in between). What are the true results if we know that the test set was of somewhat lower quality, consisting of 20% chaff and 80% grain? A simple formula gives us the answer:

$$R(p) = \frac{R'(p) - J(p).B}{1 - B}$$

Vendor	Test result	Junk detection	True detection
X	77%	5%	95.00%
Y	81%	40%	91.25%
Z	70%	75%	68.75%

Now it should be obvious that the large fraction of ballast helps those who don’t care about detecting junk or actually seek it out actively (sometimes at the expense of actual malware). As a little bonus, product Y managed to detect more malicious files than were present in the test!

All in all, detection of junk files can be used to alter the detection rate rather easily, unless the tester is extremely careful about excluding such files from the test-set.

Detection testing - practice

In the previous section, we saw why and how the size of the test set affects the results. Unfortunately, even if the test-set is well selected, validated and of appropriate size, the outcome might not be correct due to an incorrect methodological approach being taken during the test. New technologies have constantly appeared in the area of virus detection and some of them are quite incompatible with previous testing methodologies.

The first of such technologies that we'll consider is cloud-based detection of files. First of all, due to the nature of the cloud, the answer given at one time might not be the same as the answer given at another time – thus, the results are no longer repeatable [6, 8] (which is against the basic principles of the scientific method). Naturally, this can happen with regular (non-cloudy) products as well, although non-deterministic behaviour is usually considered a bug in such case, rather than being inherent to the method itself.

The standard method of testing a product by setting it loose on a large collection of files can also be problematic. In most cases, the strength of the cloud lies in its ability to shorten the response time when new, as of then undetected, threats appear. Querying "the cloud" about lots of old, already known samples does not exercise this important feature at all – it just asks for a static, one-time "dump" of the information a long time after its "best before" date.

Another popular phrase is "behavioural detection". Quite often, this approach presented as a method of catching possibly harmful actions performed by a program on execution, even if the program itself wasn't declared malicious by the scanning engine. Since very many malicious programs are already detected by other methods before they get to execute, testing the quality of behavioural detection itself can be difficult. Thus, one of the more common approaches used by the testers is to disable (or try to disable) these other methods and see whether the attacks would still be blocked by some form of dynamic analysis and detection. This has very little to do with real-world use of the products – after all, the very idea of multiple levels of defence is to have different layers co-exist in synergy. In itself, testing each layer separately is not a bad idea, but without knowing and considering the correlation between the results attributable to various layers, there is not much that can be said about their interrelation and integration.

For example, if a product misses detects 95% of malicious programs using "standard" detection (thus missing 5% of them) and 80% via behaviour-based blocking (20% of misses), one cannot conclude that together, these two methods miss just 20% of 5% = 1% of all the threats, resulting in 99% detection rate. It could well be the case that those 80% of behavioural detections are fully covered by the 95% of standard "static" detections. Naturally, the same reasoning applies to any number of protective layers, not just two of them – without knowing the correlations between their detection rates, one cannot conclude anything about their final, real-world effectiveness.

This can be seen when testing Internet content blacklisting methods. Blocking URLs or IP addresses known to be hosting malicious content has proven to be very efficient in certain cases. Since one website can be hosting various pieces of malicious code which are frequently updated, blocking the whole website can solve several problems at once. There are even publicly available sources (for example, Google's Safe Browsing [9]) which allow *anyone* to place queries about potentially unsafe links. On the other hand, blocking a particular piece of malware blocks it regardless of its origin – be it a single site or a whole lot of them. Thus, it is highly likely that there is a significant overlap between these approaches and, just as in the example above, attempt to test them independently of each other prevents anyone from drawing valid conclusions about their real-life success rate using a combination of features.

As with the cloud-based detections, this approach brought new problems for testing. Unlike a static collection of malicious files, which do not change by themselves, the content of malicious websites varies over time, so they need to be tested repeatedly – but without being noticed. Otherwise, the website might stop serving the malicious content completely or even replace it with something benign in order to fool over-curious analysts (and testers).

Performance testing

While the detection rate of a product is important, it will not be very useful in terms of protecting its users if it takes too much time to identify a program as clean or malicious, or if it slows the system down to the speed of a snail, since it will most likely be either disabled or completely removed from their machines. Thus, testing the performance and resource usage is an important component of any comprehensive “full product” test.

One very common approach is to test the slowdown introduced by the product’s resident protection when it monitors activities in the system and tries to warn the user when something malicious is trying to enter the computer (as when copied from another system or downloaded from the Internet, for example). Since testing should be done in automated fashion, this particular aspect is usually tested by simulating certain activities which are likely to be intercepted by the antivirus – usually by copying and/or creating a large number of files and measuring the overhead introduced by the measured product.

While there is nothing wrong with measuring this type of overhead, the basic question is usually left unanswered – how closely does this scenario simulate what real user is going to experience? Do users really copy large numbers of files here and there? [10] If so, what kind of files are they? Such questions need to be answered before one can determine how relevant the results of such test really are.

It is usually expected that executables (and perhaps archives) introduce the largest overhead – the first because they’re most often holders of malicious code and thus need to be analysed very thoroughly, and the second because they tend to be very large. But how often do these files get moved around the disk? Perhaps during the installation of a new application or an update of an existing application (unless the user is a developer who compiles new applications more often than Joe User, but even then the new executables are probably not generated that often). Isn’t it more likely that photos are downloaded from digital cameras, or songs downloaded from the Internet, and that these are the objects that users tend to copy the most? However, these tend to introduce very little extra delay – simply because they’re only rarely usable for nefarious purposes, and searching for the few potential dangers that could be hiding in them can be done very quickly.

Thus, this kind of testing presents an awkward dilemma – either test something that is easily measurable, but happens only rarely, or look at something insignificant which happens often. Unfortunately, it seems that the first choice tends to be the one found more often in existing tests, because consumers have got used to the fact that it’s easy to demonstrate huge and measurable differences among various AV products, even though those differences have little or no significance in real-life scenarios.

Clearly, the best approach to this type of measurement is to start by building profiles of typical users (or even better, class of users – since programmers tend to behave differently from gamers, who aren’t very much like office workers, etc.). [11] This approach allows the same type of activity to be either replayed or simulated repeatedly so that the relevant measurements can be performed. Otherwise, the results are going to say very little about the real world.

Conclusion

We’ve presented a few examples of how easy it is to test an antivirus product and produce results… which mean nothing at all. The list is not comprehensive by any means – but the problems presented are the ones we tend to encounter most often. Some of the methods can be corrected by taking the advice described here, but some of them might need to be abandoned and replaced by a completely different approach. Time is a precious resource and poor tests waste too much of it for vendors, who could have used it for real improvement of their products, fine-tuning for protection in the real world rather than for optimized performance on the testers workbench. After all, good real-world testing helps both the consumers/users and the producers/vendors – so the sooner the flagrant flaws are corrected, the better

[1] AMTSO, Issues involved in the "creation" of samples for testing; URL: http://amtso.org/amtso---download---issues-involved-in-the-creation-of-samples-for-testing.html

[2] Muttik, I.: Comparing the comparatives; Virus Bulletin Conference, September 2001; page 45-56; URL: http://www.mcafee.com/common/media/vil/pdf/imuttik_VB_conf_2001.pdf

[3] URL: http://www.amtso.org/documents.html; http://amtso.org/related-resources.html

[4] URL: http://amtso.org/documents.html

[5] Muttik, I.: A Brief History of Time; CARO 2010 presentation;
URL: http://www.f-secure.com/weblog/archives/Igor_Muttik_brief_history.pdf

[6] Harley D. & Lee A.: Call of the WildList: Last Orders for WildCore-Based Testing?, VB 2010 Conference Proceedings.

[7] Harley D., Untangling the Wheat from the Chaff in Comparative AV Reviews, Small Blue-Green World 2007. http://www.eset.com/resources/white-papers/AV_comparative_guide.pdf

[8] Garrad M., Jones P., Myers L. and Parsons M.: Paradigm Shift - From Static To Realtime, A Progress Report; EICAR 2010 Conference Proceedings. URL: http://./uploads/eicar2010-meyers-paradigmshift.pdf

[9] Google Safe Browsing API
http://code.google.com/apis/safebrowsing/

[10] AMTSO, Fundamental Principles of Testing: URL: http://amtso.org/amtso---download---amtso-fundamental-principles-of-testing.html

[11] Vrabec J & Harley D., “Real Performance?”, EICAR 2010 Conference Proceedings. URL: http://amtso.org/uploads/eicar2010-harley-realperformance.pdf