DARWIN INSIDE THE MACHINES: MALWARE EVOLUTION AND THE CONSEQUENCES FOR COMPUTER SECURITY

Dimitris Iliopoulos, Christoph Adami
Keck Graduate Institute of Applied Life Sciences, Claremont, CA 91711, USA

Péter Ször
Symantec Corporation, 20330 Stevens Creek Blvd, Cupertino, CA 95014, USA

Email pszor@symantec.com

ABSTRACT

Recent advances in anti-malware technologies have steered the security industry away from maintaining vast signature databases and into newer defence technologies such as behaviour blocking, application whitelisting and others. Most would agree that the reasoning behind this is to keep up with the arms race established between malware writers and the security community almost three decades ago. Still, malware writers have not as yet created new paradigms. Indeed, malicious code development is still largely limited to code pattern changes utilizing polymorphic and metamorphic engines, as well as executable packer and wrapper technologies. Each new malware instance retains the exact same core functionality as its ancestor and only alters the way it looks. What if, instead, malware were able to change its function or behaviour autonomously? What if, in the absence of human intervention, computer viruses resembled biological viruses in their ability to adapt to new defence technologies as soon as they came into effect? In this paper, we will provide the theoretical proof behind malware implementation that closely models Darwinian evolution.

Biological viruses are under constant attack by immune systems and artificial drugs. Yet they systematically manage to evolve new functionalities that circumvent such countermeasures, leading to recurrent epidemics. According to the biological analogy, evolvable malware will be able to alter its functionality by autonomously incorporating behaviours freely available to it by the numerous discoverable APIs. The new behaviour profiles would constantly be screened by security software in the same way natural selection acts on biological organisms. In the end, the malware instances that are better equipped to survive countermeasures will be able to proliferate more efficiently. Such malware poses a real threat to the current methods of detection due to the vast numbers of functions it can adopt, and that cannot possibly be screened for. Furthermore, it is likely that clean-program functionality will be favoured amongst such behaviours since it shields malware that is mimicking clean programs from behaviour blocking. As a consequence, we predict that behaviour-based virus detection would quickly become ineffective if malware can evolve based on the Darwinian paradigm.

INTRODUCTION

During 2007 alone, attackers created over 700,000 new malicious programs [1, 2]. While this number appears daunting, modern anti-virus (AV) systems have dealt with this crop successfully every year, by using sophisticated signature and behaviour detection techniques. All of the new viruses are, as far as we know, the product of human engineering. Here we discuss the possibility of an alternative form of design: the emergence of novel function in an autonomously evolving piece of malware.

The possibility of autonomously evolving computer programs is not new. Artificial Life ('Alife') researchers have studied self-replicators in computer environments since 1979 [3], and the first autonomously mutating self-replicating computer programs were introduced by Ray in 1992 [4]. More recent systems such as the *Avida* platform have established that Darwinian evolution of computer programs can be used to study evolutionary biology and genetics [5, 6]. As opposed to computer malware, Alife systems are squarely aimed at the research environment. In these systems, the code is implemented with small, well-defined instruction sets that are highly evolvable; that is, the probability that a mutation leads to another functional program is high (of the order of several to tens of per cents). Most standard computer languages are not robust in this manner. For example, the x86 instruction set only tolerates very few mutations (of the order of fractions of a per cent of the code [7]). Alife systems also differ in that they implement simulated environments where specific program behaviours are rewarded. While this setting is very different from the standard computer malware paradigm, computer virus researchers already hinted at the idea of autonomously evolving malware in the early 1990s [8]. In particular, Spafford performed an exhaustive review of computer viruses under the Artificial Life perspective confirming the absence of functional evolution in such programs [9]. Although the possibility of an autonomously evolving virus – as the one discussed here – was mentioned, it was quickly dismissed as a task too daunting and requiring a very large implementation, possibly larger than the OS itself. More recently, the idea has resurfaced, with a higher emphasis this time on the outcome [10] and the possible mechanics of its emergence [11]. We argue here that apart from a hint and some failed implementation attempts (e.g., W32/Zellome, see [12]), self-evolving malware has yet to appear. The concept itself, however, is relatively simple, and the consequences of the release of evolving malware should be studied.

DARWINIAN EVOLUTION VERSUS MALWARE EVOLUTION

Evolution as a process requires the presence of three simple elements: replication, variation and differential fitness. Replication allows for inheritance, and in particular the inheritance of variations, introduced by different mechanisms of mutation. These variations are then selected for or against via the competition of individuals over limited resources. The competition leads to differential fitness, i.e. differences in reproductive success between individuals within a population. Mutations that provide organisms with a reproductive advantage over others (beneficial mutations) will tend to propagate to future

generations. It is important to note that the fraction of beneficial mutations (among all those that can occur) is usually very small, with most mutations being deleterious and the rest neutral. Nonetheless, given the long timescale available to biological evolution along with large population sizes, such gradual stepwise beneficial changes will lead to the emergence of complexity [13].

Some have argued that neither the geological timescales nor the large populations that are germane to biological evolution will be available to evolving computer malware. However, generation times for computer viruses are many orders of magnitude shorter than even bacterial generation times, and furthermore selective pressures are expected to be much stronger and mutation rates higher, speeding up evolution [14]. We should note here that geological times, in contrast to popular belief, are not a prerequisite for the appearance of evolutionary change, even in biological evolution. The emergence of new functional complexity on much smaller timescales than geological ones has recently been observed in very different experiments. A 36-year experiment with lizards, for example, has shown that such a comparatively short time frame is sufficient for major adaptive morphological changes to arise, such as novel morphological structures in the form of caecal valves [15]. Also, in a long-term experiment with *E. coli* bacteria, one particular strain evolved the ability to metabolize a new carbon source (citrate) after about 31,500 generations, a complex adaptation that required a multitude of mutations [16]. Given the drastically shorter generation time of computer viruses, generations of this order of magnitude can be achieved within weeks at most.

During the three decades of their existence, computer viruses have moved from simple replicators to advanced polymorphic and metamorphic implementations [17]. The underlying goal of this progression has been to increase the variability of the virus's signature to the point that tracking different instances of the same virus becomes too daunting a task. Nonetheless, even for the most sophisticated metamorphic viruses [18], the specific functionality and overall behaviour of the virus remain intact. Signature obfuscation, or as we will refer to it here 'cryptic variation', will not allow for the discovery of new functionality. What true Darwinian evolution can accomplish is vastly different, because it is the process responsible for the *de novo* generation of all of the complexity of life.

The variation observed in surviving lineages of biological viruses (as compared to their ancestors) is a direct result of information 'exchange' between the virus and its environment. Simply put, biological viruses are constantly testing new ways of exploiting environmental resources via the process of mutation. In contrast, computer viruses do not exhibit such traits, relying instead on changing their appearance to avoid detection. Functional (as opposed to cryptic) variation, such as the discovery of a new exploit or the mimicry of non-malicious behaviour masking malicious actions, is not part of the arsenal of current malware. While there are examples of functional variation that have occurred by chance (reviewed below in support of our hypothesis) there are no examples of computer malware that exhibits intentional functional change between generations. In the absence of functional variation, differential fitness will never be realized in computer viruses since the reproductive success of offspring remains unchanged. In the event that a behavioural signature is developed for the virus, the entire population, including the cryptic variants, is affected equally. In this case, the functional/behavioural uniformity of the virus population would force all of it to extinction by the countermeasure. Functional variants, on the other hand, because of the variation in behaviour, can escape behaviour-based detection methods. Below, we investigate the theoretical possibility of functional variation in computer viruses and their consequences.

MODEL IMPLEMENTATION

Suppose, malware M is comprised of an arbitrary number of malicious functions:

$M = \{M_1, M_2, M_3, \ldots, M_N\}$.

An evolutionary function $EF \in M$, is introduced into the existing set:

$M = \{M_1, M_2, M_3, \ldots, M_N, EF\}$.

EF has the ability to generate new functional code N after each generation of M, with a given probability. This can be achieved by different methods, including via the insertion of code extracted from randomly selected APIs present in the malware's native environment (for example, extraction of function calls that are part of the *Windows* API). Alternatively, in the case of a script threat, extraction of script code from other scripts might also serve the same purpose. In any case, the resulting malware M now consists of the set

$M(EF) = \{M_1, M_2, M_3, \ldots M_N, EF, N_1..N_m\}$, where N=N(EF).

The newly generated function N need not necessarily be malicious. Also, the M_i might trigger anti-virus responses individually or as a combination of each other. The malware M(EF) is now an evolvable threat.

Researchers that envisioned how autonomously evolving malware could be coded have largely focused on binary code manipulation [8,11]. The high percentage of lethal mutations that would be experienced by such malware (due to the brittleness of the code) would forbid evolution from occurring. Consequently, the need for an 'evolvable' language or meta-language of implementation becomes apparent. While some work in this direction has been undertaken [7], the prospects for such a design are daunting. By relying instead on functional code that is already present in the malware's environment, the problem is reduced to finding a way of adopting that functionality in the malware code, instead of creating a language that can code for that functionality autonomously.

Another major consideration with such an implementation is the actual coding of EF itself, as it might appear to represent a static component for which a signature could be produced. However, the EF could use any number of advanced polymorphic or metamorphic engines to hide itself from detection. The ongoing effort by the anti-virus community to avoid maintaining the large signature databases of the past, combined with the advanced metamorphic engines used by viruses such as

W95/Zmist [19] make a cryptic EF implementation feasible. In fact, it is highly probable that EF's presence can be fully disguised by M. By simulating non-malicious behaviour for example, malware would at best be classified as a false positive even if EF is detectable. Given the anti-virus industry's aversion to false positives, it is conceivable that malware would exploit this loophole and autonomously and forcefully mimic clean application behaviour.

FUNCTIONAL EVOLUTION IN CURRENT SECURITY ENVIRONMENTS

Adapting to security environments can be illustrated via a number of examples. In this section, we list several ways in which random changes can lead to evasion of detection from behaviour-and signature-based blocking as well as application whitelisting.

Clean-application mimicry

AV systems are usually designed with a bias to avoid false positives. This bias creates a loophole that can be exploited by functional adaptation, because as the number of detections by an AV system increases so do the false positives of each product. As a consequence, the AV industry attempts to reduce the footprint of signatures, but this can significantly amplify the risks of exploitation by programs that have learned to mimic clean applications either by behaviour or by signature.

Imagine that a program imports API function 'Foo', which makes it similar to programs that create false positives for the AV system. As a result, the presence of function Foo excludes that program from the attention of behavioural blocking. This problem exists even if the system uses weighting, as long as the negative weights to identify possible clean applications can be simulated by the newly evolved functions in malware M. There are analogies for this type of evolutionary adaptation in biology. Some viruses have evolved a set of proteins that mimic precisely those proteins that are involved in combating the virus (the so-called complement regulator proteins [20]), or else by mimicking a protein that the immune system uses to recognize viruses (so-called Fc proteins [21]). In both cases, the immune system is fooled into treating the virus as 'clean' because it looks like part of the immune system itself.

If malware M appears to have an application user interface, it will most likely be classified as a clean application because typical malware does not have one. In another example, if malware M is not packaged or wrapped, it might appear much less suspicious. Malware writers use packers extensively to hide signatures from AV scanners. One of the cheapest ways for attackers to do this is to use run-time packers or wrappers on top of the existing malware. As a result, malware released in packed form might look much more suspect than a non-packed version. Evolving malware capable of presenting itself in both packed and unpacked forms would create new challenges for heuristics. There is a parallel to this evasion technique in biological viruses. In fact, immune systems often trigger on the envelope (or capsid) of a virus. As a consequence, some viruses have adapted to an infection mode without an envelope and capsid (for example, the viroids [22]). The reverse strategy also exists, where viruses such as the herpes viruses, in an attempt to masquerade themselves as host cells, create elaborate envelopes from host cell membranes that contain a series of host-unique markers on them. Such envelopes allow the herpes virus to evade host detection and suppress immune responses [22].

Whitelisting deception

The practice of application whitelisting offers another exploit for evolvable malware. Because the number of known applications is already very high, typically application whitelisting focuses only on executable applications (such as Portable Executable files on *Windows* systems). As a consequence, attacks originating from other types of objects (such as documents) are not controllable by these systems. Malware that is not presented in an executable form would thus be excluded from the attention of whitelisting systems completely. Self-evolving executable malware that could convert itself to new forms, for example via executable-to-script-conversion, would pose a challenge to any whitelisting system. Instead of presenting itself as an executable, the code might rewrite itself as a macro in a document, or as a command-line script.

Another problem appears if there are no file objects involved, as in the case of the W32/CodeRed family that targets a whitelisted application in memory over the network. Not surprisingly, when such threats emerge for the first time, they often cause an epidemic. While in-memory threats are not common, their presence is expected to rise if there is a need to adapt to a known whitelisting solution.

Another scenario involving whitelisting deception could be the following. Suppose a program creates large populations of clean programs that it distributes over the web. However, the clean programs are designed in such a way that their MD5 hash is identical to its own (malicious) self [23]. Because whitelisting applications collect executables from a variety of 'trusted' sources, the wide distribution of clean 'twin' programs could have the result of whitelisting the malicious version. Thus, such an attack can exploit both application whitelisting as well as application reputation systems, which score applications based on their popularity among users with good reputations. A simple biological example of such an attack is represented by the avoidance of bacterial DNA degradation via self-methylation. Bacteria tag their own DNA with a methyl group, and cut up any DNA inside the cell that does not carry such a group (the methylation is the analogue of an MD5 hash). Viruses have emerged that have learned to emulate these tags (whitelisting themselves in the process) and thus escape detection.

Virtual machine anti-emulation

It is conceivable that the size of a long function loop, or a function that calculates parameters to function calls on the fly, could be artificially increased by malware. In such a case, the resulting novel functions N might accidentally create an anti-emulation feature for a virtual machine by exhausting the emulator's preset upper limit iteration number to examine the program. This way, M will be capable of competing with an emulator, thus exploiting this feature for its own survival. In this case, the threat will be capable of circumventing not only

behavioural blockers that rely on emulators, but also a scanner's built-in heuristics analysers as well. Random sequences of APIs with parameter-check can also introduce such issues. This trick is often used by existing polymorphic threats as an anti-emulation feature. The W95/Drill family of viruses utilized random calls to a predefined set of APIs to confuse emulators. Later on, the Storm worm attacks (that created the Storm botnet) delivered thousands of copies of different malware executables that utilized layers such as W32/Tibs on them to confuse anti-virus products. The API sets used were changed by the attackers regularly to reduce the chance of detection based on the presence of such an API profile. A self-evolving threat would carry this feature by its own nature, as it can create new code and verification functions for them. We are not aware of a biological analogy for this trick.

Evasion by proxy

Even security products such as personal firewalls can be affected by self-evolving threats. Suppose threat M has a feature to proxy behaviour using another executable. While the threat may have several malicious features, it has the option either to execute them itself, or by using a running application as a proxy instead. As a result, when executing the communication via a trusted application, the threat can remain undetected when communicating to an outside location (such as the command control channel of a bot network). Alternatively, all malicious features might be separated among several processes as threads. Imagine that a set of threads from a set of processes together constitutes a malicious program. Yet, on its own, each feature might not be triggering the attention of heuristics or the behaviour-blocking engine, allowing the threat to escape attention.

Trust relationships can also introduce problems. For example, *Windows Vista* does not allow an executable to alter certain file permissions, even if an administrator executes the application. Still, using a script such permissions can be changed. For example, files such as kernel32.dll and their permissions can only be changed by so called Trusted Installers who have full control over these files (see Appendix A). If one tries to change the permission using CACLS.EXE from the Resource Kit, the attempt will fail because this executable cannot be used to take ownership of files, since it is not trusted. Yet, one can use XCACLS.VBS, and execute it with CSCRIPT.EXE. The script is trusted, since the script interpreter is a trusted application. This means that threats can potentially evolve by changing their representative forms (such as the particular language environment they use) to develop adaptive features to a security environment. What an executable is not able to do due to security restrictions might be achievable as soon as the threat feature is executed as a script. In addition, using a trusted proxy such as EXPLORER.EXE on *Vista*, a threat can take ownership of files that otherwise it would not be allowed to do (for example by using code injection techniques to become part of the process address space of *Explorer* in memory).

Environment-induced functional variation

New functionality can emerge as a result of environmental effects. Many websites, for example, change the code within HTML files on the client side arbitrarily by presenting extra functionality to the user. In particular, links to advertisement messages can be replaced with actual content. In other cases, scripts are inserted into pages, and changed on an ongoing basis (see Appendix B). Such effects could change actual malware code, rendering AV scanners incapable of identifying the threat.

We have seen several verified cases in the past where random corruptions to virus code resulted in a new variant of the virus that renders it undetected by security software. For example, a variation of the W95/CIH family exists that is the result of such a corruption: the replica escaped the attention of most scanners when the natural modification happened to a piece of its non-essential code, without altering the threat's replication functionality. In another case, an error in *Microsoft Word* resulted in the creation of thousands of macro virus variants via random corruptions. These macro-body corruptions were often ignored by the macro interpreter due to the existence of error handlers in the original virus code. As a consequence, these *Word*-induced variations could easily result in surviving mutations, as long as a single MacroCopy() command responsible for the successful replication of the threat was still present unchanged.

Malware code merging

A threat might be able to snatch code from another program in its environment. We have seen examples of a virus like Pinfi jumping on top of worms to replicate in new environments as a combination threat. Security products do not always recognize the worm once it is infected with a virus, and the combination helps the survival of both threats. Disinfected worm copies can also escape attention, as can copies of worm replicas that changed due to the transfer of code over network channels. It is conceivable that an evolutionary function in the malware could snatch clean or malicious code from other programs. It could integrate code from other programs by identifying function prologue and epilogue code. When this takes place, a function is safely inserted into the code base of the evolutionary virus as a new 'function' by running the newly acquired code as a new thread. Existing features might be replaced by the code, which could end up producing reliable output to a given input. (For example, as long as function X returns values greater than 0, it is accepted.) Even complete functionality might be snatched from another clean program, or another virus. As previously predicted [17], a cooperation protocol can enhance sharing of features between malicious executables as well. Code snatching is a tried and true function of almost all biological organisms. Bacteria exchange code in small segments called plasmids, while viruses routinely integrate bacterial code into their own. Often, viruses carry this piece of code to other bacteria, a phenomenon known as transduction.

A form of evolution was observed in macro viruses, which often merge their code base into a document. (Note that the integration of viral code into the host code is the default action for several biological viruses, e.g. HIV [24].) Often the file has a clean macro, and a virus with a set of macros. In addition, another virus may insert its set of macros at the same time, leading to viral macro code merging with both viral and clean macro code. In biology, this phenomenon is quite common, and