Saber y Hacer
Revista de Ingeniería de la USIL
Vol. 1, Nº 1. Primer semestre 2014. pp. 41 - 54

Code obfuscation using pseudo-random number generators[1]

Ofuscación de código utilizando generadores de números pseudoaleatorios

John Aycock[2] ; Juan M. Gutiérrez[3] & Daniel M. Nunes[4]
Universidad San Ignacio de Loyola

[2] Department of Computer Science University of Calgary 2500 University Drive NW Calgary, Alberta, Canada T2N 1N4. E-mail: aycock@ucalgary.ca
[3] Independent Researcher Arequipa, Peru. E-mail: wits.gutierrez@gmail.com
[4] Department of Computer Science University of Calgary 2500 University Drive NW Calgary, Alberta, Canada T2N 1N4. E-mail: dmncastr@ucalgary.ca

Abstract

We describe a novel method for malicious code obfuscation that uses code already present in systems: a pseudo-random number generator. This can also be seen as an anti-disassembly and anti-debugging technique, depending on deploy- ment, because the actual code does not exist until run – it is generated dynamically by the pseudo-random number generator. A year's worth of experiments are used to demonstrate that this technique is a viable code obfuscation option for a malicious adversary with access to large amounts of computing power.

Key words: obfuscation, pseudo-random, generator.

Resumen

Se describe un nuevo método para la ofuscación de códigos maliciosos que utiliza códigos ya presentes en los sistemas: un generador de números pseudo-aleatorios. Esto también puede verse como una técnica anti-desmontaje y anti-depuración, dependiendo de su despliegue, debido a que el código real no existe hasta su ejecución - que se genera de forma dinámica por el generador de números pseudo-aleatorios. Se han usado xperimentos de todo un año para demostrar que esta técnica de ofuscación es viable para un adversario malicioso con acceso a una gran potencia computacional.

Palabras claves: ofuscación, pseudo-aleatorios, generador.

I. Introduction

What would a malicious adversary do with a million computers? It is reasonable to assume, given the bustling underground economy (Franklin et al, 2007), that whatever an adversary does is activity that is likely profit-driven. The activity may be direct, like stealing data to sell or mounting distributed denial-of- service attacks for extortion purposes. Alternately, the activity may be indirect in support of the adversary's continued operations, such as spamming out Trojan horses to compromise more computers.

Other indirect activities include measures the adversary takes to avoid detection of their malicious software on the computers they compromise. For example, server-side poly- morphism is an attempt to frequently change the appearance of malicious code (Szappanos, 2007). Also along these same lines are techniques the adversary can use to make (static or dynamic) analysis of malicious code more difficult; one of these techniques is code obfuscation.

Code obfuscation is an interesting area because it has legitimate applications discouraging reverse engineering (Collberg, Thomborson & Low, 1997) as well as malicious applications. We focus on the latter in this paper: code obfuscation performed by a malicious adversary who has illicit access to large amounts of computing resources. Once, it would have been silly to assume that an adversary had lots of computing power without also having government or military backing. However, botnets comprising huge numbers of computers are no longer a rarity.[5] One botnet reputedly had 1.5 million computers back in 2005 (Sterling, 2005). More recently Storm Worm was estimated variously to involve over a million machines (e.g., Larkin, 2007), although active measurement subsequently pared that number back to a lower estimate (Holz et al, 2008). Downadup/Conficker, which could arguably be classified as a botnet (Tiu, 2009), had size measurements that put it over 8 million strong (F-Secure, 2009). Another botnet was discovered with almost 2 million computers (Finjan, 2009). It is thus particularly timely to consider what may be done with a large number of compromised computers.

Tying this back into code obfuscation, an adversary would want to use a value that is expensive to compute or search for to obfuscate their code, thereby making use of their superior computing power and rendering the result hard to duplicate by security researchers. At the same time, the resulting obfuscated code must be cheap to deobfuscate and run, because that part of the process would occur on a single targeted computer. One possibility, we discovered, lies in the use of pseudo-random number generators. There are obviously other obfuscation techniques – for example, strong encryption could be used - but the technique we describe is novel, and novelty can be helpful in terms of the survivability of malicious software. For this reason, it is helpful for defensive purposes to consider this obfuscation technique before it is seen in the wild, allowing appropriate detection to be added to antivirus software if necessary.

In the next section we describe the overall idea of code obfuscation based on pseudo-random number generators (PRNGs). Section III details our proof-of-concept experiments. Finally, Sections IV and V contain related work and our conclusions, respectively.

II. PRNG – based obfuscation

We begin with a few definitions. C is the code that the adversary wants to obfuscate, and C is comprised of a sequence of bytes $b_1 b_2 b_3 \ldots b_n$. The different bytes need not be unique.

[5] Measuring the size of a botnet, or even deciding what "size" means, is a nontrivial problem (Rajab et al, 2007), but the numbers are large enough regardless that the measurement issue does not affect the viability of our work.

The basic idea is that the adversary tries to finds a seed s for a PRNG *G* so that *C* may be reconstructed using a series of consecutive calls to *G*, as shown in Figure 1. Once it has been completely regenerated, the code *C* may be run as usual. As a practical matter, the output of *G* is taken modulo m to bring it into the appropriate range of byte values; m is another parameter that the adversary must choose.

seed PRNG *G* with *s*
for i in 1...*n*:

    b*i* ← *G*() mod *m*

Figure 1. Basic regeneration of *C*
Source: Own elaboration

Obfuscation, then, is a matter of the adversary finding s and m, a problem which lends itself trivially to a brute-force search through all the different possibilities. Furthermore, the search can be partitioned so that each machine in a botnet can look through a portion of the search space independently. This meets the criteria in the last section: an expensive search used for obfuscation, and a cheap deobfuscation that requires only a handful of calls to a PRNG.

It is important to note that s and m may not exist for an arbitrary value of *C*. We are not claiming that this will work for any *C*, but when such values can be found, an adversary can use them to obfuscate code. We discuss in the next section variations on the basic scheme which make s and m easier to find.

The Achilles heel of this code obfuscation technique is knowledge of these two values, especially *s*. If a security researcher knows s, then the PRNG sequence from *G* can be generated and the logical possibilities for *m* can be tried until a plausible candidate for *C* emerges. However, if *m* is known, a security researcher would have no extra insight into *s*. We therefore argue that there the adversary would only try to hide s.

Clearly, *s* cannot be stored with the obfuscated version of *C*, or it would be trivial to deobfuscate the code. (This fact has not deterred past malicious code from "encrypting" itself and carrying its decryption key around with it, however.) Instead, we consider two possible locations for s; we call the computer where deobfuscation is occurring the *target*.

1. *s* may be located outside the target system. The deobfuscation code would fetch s from some external source, like an Internet web site. As Conficker has demonstrated, it is possible to have malicious software update itself in a robust fashion (Porras, Saidi, & Yegneswaran, 2009), and the same principles could be used to access *s*.

2. *s* may be computed using information located on on the target system; this is referred to as environmental key generation (Riordan, & Schneier, 1998). For example, the target system's domain name may be hashed into a single numeric value that, when XORed with a constant *x*, yields *s*. This would allow an adversary to create malicious software that only deobfuscates correctly on a single target system.

It is now apparent how our obfuscation technique functions as an anti-disassembly and anti-debugging technique. So long as s is unknown to a security researcher, then *C* cannot easily be regenerated – being unknown, *C* is thus resistant to static analysis (anti-disassembly) and dynamic analysis (anti-debugging). Of course, if s is discovered by a security researcher, then analysis is possible.

III. Experiments

To study the viability of PRNG-based code obfuscation, we have run a wide variety of proof-of-concept experiments. We selected four code samples of varying sizes to use for *C*:

PyHW

This is the bytecode for the simple Python function:

def foo(): print 'Hello, world!'

which is nine bytes long and contains several duplicate values, making it a useful first test case. In hexadecimal, it is 64 1 0 47 48 64 0 0 53.

EICAR

This is the standard EICAR anti-virus test file (EICAR, 2006) 68 bytes in length with 34 unique values.

EICARZ

The EICAR test file again, compressed with Zip. It is 184 bytes long with 64 unique values.

EICARZZ

EICAR yet again, doubly-Zipped, i.e., this is EICARZ compressed again with Zip. It is 308 bytes long and has 77 unique values.

In the remainder of this section, we summarize our experiments and their results.

A. Direct Generation.

The most straightforward approach is that shown in Figure 1, where the bytes of *C* are generated directly by the PRNG. Starting with PyHW, the shortest code sample, we fixed m to be 65_{16} to keep the generated values all within the range $0 \ldots 64_{16}$ that PyHW requires. The PRNG in the standard C library was used, seeding with srandom and generating a number with random.

All values of s were tried in the range 0 . . . 2147483646 and, for each value of *s*, the first 5000 values generated by the PRNG were searched for the PyHW code. We were unable to find all of PyHW; our best results yielded the first four bytes of PyHW right at the start of a PRNG sequence, meaning the first four calls to random immediately following a call to srandom. One of the 15 of these we found, for example, used the seed 827090594. This is an important point: if the adversary finds multiple usable results during their search, then they may choose any of them, not just the one that would be discovered first in a sequential brute-force search. This would potentially make the reverse engineering task more difficult for a security researcher attempting to duplicate the adversary's brute-force feat.

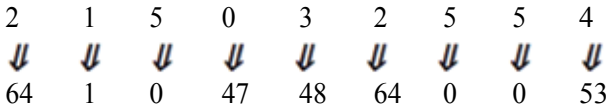
Looking deeper into the PRNG sequences, we found the first six bytes of PyHW in 12 locations. For instance, with $s = 288409801$, we found the six hex bytes 64 1 0 47 48 64 after 1374 calls to random.

Given the limited success even with PyHW, we decided that this approach is probably not feasible unless the adversary is extremely lucky, or has the luxury of altering *C* to match a given PRNG sequence.

B. 1:1 Interpretation.

It's said that there are really only two solutions in computer science: caching and indirection. We employ the latter for this next attempt. If *C* cannot be directly generated, then instead we look in the PRNG output for a sequence *C*' with bytes $b'_1 b'_2 b'_3$ such that b'_1 uniquely maps to $b_1 b'_2$ uniquely maps to b'_2, and so on. As before, the different bytes need not be distinct so long as the unique mapping exists.

With this change, PyHW was found very early on in the brute-force search. The modulus *m* was set to 6, as PyHW has six distinct values, and with $s = 5412$, we found the sequence 2 1 5 0 3 2 5 5 4 at the start of the PRNG sequence. This gives us the unique mapping



A slightly modified approach to deobfuscation is required, however – the pseudocode in Figure 1 will no longer suffice. Figure 2 shows one solution, where an array M is used to map byte values from *C'* to *C*.

```
seed PRNG G with s
for i in 1...n:

    bi ← M[G() mod m]
```

Figure 2. Regeneration of *C* using mapping table
Source: Own elaboration

Another solution is to view the bytes of C' as bytecode instructions for a virtual machine, instructions that require an interpreter to deobfuscate and run. Collberg et al. refer to this as 'table interpretation,' and characterize it as 'One of the most effective (and expensive) transformations' [3, page 13]. Ferrie lauds interpreted virtual machines similarly, calling them 'perhaps the ultimate in anti-dumping technology, because at no point is the directly executable code ever visible in memory.' [14, page 3]. Our approach goes further still, because as discussed in Section II, even the virtual machine code need not be in memory until it is generated by the PRNG. (Note that custom virtual machines are not simply academic: they are used in some anti-reverse engineering tools already, like Themida [Ferrie, 2010], Code Virtualizer [Oreans Technology, 2009], and VMProtect [VMPsoft, 2009]).

There are many different interpretation techniques (Debaere & Van Campenhout, 1990; Klint, 1981), and our work is not tied to a particular interpretation technique. Figure 3 shows some pseudocode for one interpreter for illustrative purposes. The variable pc acts as a program counter, and the byte-long instructions are extracted and executed by the interpreter one by one. Obviously there are some simplifications in the pseudocode: for example, there are likely to be constants and offsets that must be encoded into C'.

```
pc ← 0
while true:
    instruction ←
    switch (instruction)
    case :
            execute b1
        case :
          execute b2
        .⋱
        case b′ : execute bn
    pc ← pc + 1
```

Figure 3. Deobfuscating interpreter for C^t
Source: Own elaboration

In the worst case, each original instruction in C may be encoded as a different byte value, complete with constants, offsets, and other associated information. For example, Figure 4 shows the disassembly of the first few EICAR instructions, and how each could be remapped to a single-byte value. Other schemes are possible too, but a full discussion of instruction encoding is beyond the scope of this paper.

```
58      pop   ax       ⇒  b′1
354f21  xor   ax,214f  ⇒  b′2
50      push  ax       ⇒  b′3
254041  and   ax,4140  ⇒  b′4
```

Figure 4. Recoding C
Source: Own elaboration

Although we were successful in finding PyHW with this 1:1 interpretation, we were unable to find EICAR, EICARZ, or EICARZZ. A different approach was needed.

C. 1:N Interpretation.

Assuming an interpreter or mapping table is being used, there is no reason to limit ourselves to a 1:1 mapping from byte values in *C'* to byte values in *C*. Changing m to 256, so that we generate all 256 possible byte values, PyHW is found even more easily and frequently. For example, taking $s = 1$, the PRNG generates the hex values 67 c6 69 73 51 ff 4a ec 29, which map to PyHW as shown:

67	c6	69	73	51	ff	4a	ec	29
⇓	⇓	⇓	⇓	⇓	⇓	⇓	⇓	⇓
64	1	0	47	48	64	0	0	53

The duplicated values in *C*, 0 and 64, are now each represented using more than one value in *C'*.

We were finally able to find EICAR several places with this technique, such as with *s* = 60594, but EICARZ and EICARZZ still eluded us.

D. Dual-Stream 1:N Interpretation.

With the results for 1: *N* interpretation, we observed that both PyHW and EICAR were found without any byte values in *C'* being repeated. This observation allows the search constraints to be reduced considerably. We no longer have to look for PyHW or EICAR specifically; we need to look for a sequence of unique bytes output from the PRNG that is *long enough* to map into PyHW or EICAR. The search is still extensive, but the result – finding a unique PRNG sequence *n* bytes long – can be used to obfuscate any code sequence *n* bytes in length. This is a far more general approach.

```
seed PRNG G_0 with s_0 seed PRNG G_1 with s_1 for i in 1...n:
    x ← G_0() mod m
    if B[i] = 1:
        x ← G_1() mod m
    ← x
```

Figure 5. Deobfuscating a dual PRNG stream
Source: Own elaboration

We extend the approach further still. One PRNG will ultimately be limited in terms of how long a unique sequence it can produce, so we combine two PRNGs giving us two independently-produced streams of pseudo-random numbers. A bit vector stores one bit per byte of C' generated, telling the deobfuscation algorithm which PRNG value from the two streams to select.

Pseudocode for the deobfuscation algorithm is shown in Figure 5. B is the bit vector used to select the PRNG, and we now need to search for two PRNG seeds, s_0 and s_1. We used $m = 256$ and searched through all combinations of $0 \leq s_0 < 100000$ and $0 \leq s < 65536$. The PRNGs used were changed to a Mersenne Twister implementation (Kuenning, 2007) that supported multiple independent PRNGs. The search found a unique sequence 156

bytes long using this technique with $s_0 = 202$ and $s_1 = 54170$. Revisiting our code samples, this result is long enough for PyHW and over twice the length needed for EICAR, but 28 bytes short for EICARZ. (EICºARZZ's length exceeds 256 and cannot be obfuscated with this technique unless *m* is increased).

E. Dual-Stream 1:N Interpretation with NOPs.

To extend our results to longer unique sequences, we introduced NOPs, byte values that should be skipped during deobfuscation; ordinarily these NOP values would be repeated ones that would have caused the unique sequence to terminate.

With this change, we again took $m = 256$ and used two independent Mersenne Twister PRNGs to search through all seed combinations $0 \leq s_0 < 100000$ and $0 \leq s_1 < 65536$. We found 76030 values for s_0 and s_1 in that range that yielded unique sequences (with NOPs) of length 190 or above; this gives an adversary many seed values to choose from. Recall that EICARZ was only 184 bytes long, so it can be easily obfuscated with this technique. The longest sequence was 203, found at both $s_0 = 57547$, $s_1 = 5328$ and $s_0 = 4960$, $s_1 = 15041$.

The pseudocode for deobfuscation is in Figure 6. N and U are lists of NOP values and unique values, respectively. After this code executes, U contains the unique sequence that can be used for *C'*.

```
U = []
N = []
seed PRNG G0 with s0 seed PRNG G1 with s1 while |U|<n:
    x ← G0() mod m
    if x ∈ N:
        continue
    if x ∉ U:
        append x to U else:
        x ← G1() mod m
        if x ∈ U:
            delete x from U append x to N
        else:
            append x to U
```

Figure 6. Deobfuscating a dual PRNG stream with NOPs
Source: Own elaboration

F. Multiple-Stream Experiments.

The obvious extension is to use more than two independent PRNGs for generation. Initial experiments suggest that this gives us much more flexibility in the sequences we can generate: we were able to directly generate EICAR using 12 seeds, for example.

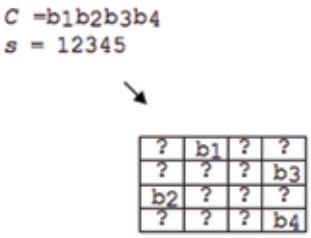
An interesting optimization aspect is introduced, because there is now a search for the smallest number of seeds required to produce a given sequence. An effective solution to this optimization problem would also have potential applications to improving the dual-stream techniques. The tradeoff to multiple-stream generation is that more seeds are required, more calls needed to PRNGs, and more bits per generated byte to select the PRNG value to use. It is an intriguing area for future work.

G. Other Experiments.

Another approach we explored is also based on PRNGs. While it foregoes the search aspect to obfuscation, it is able to obfuscate any given code sequence *C*. Obfuscation works in three steps:

1. A PRNG seed *s* is chosen. (Note that it is chosen, not searched for.)
2. Values generated by the PRNG are used to place the bytes of *C* into a table.
3. Unused table entries are filled with realistic, albeit random, values.

Figure 7 shows an example. Deobfuscation requires the table and the seed s in order to step through the table and extract the bytes of *C*. Given the table size and the absence



of a computationally intensive obfuscation process, we do not consider this technique to be as strong as the others we have described, even though it will work for any *C*.

Figure 7. Search-free PRNG obfuscation
Source: Own elaboration

IV. Related work

To the best of our knowledge, there is no prior work on this exact problem. Directly related, however, is Toyofuku et al. (Toyofuku, Tabata, & Sakurai, 2005), who explored the use of

random numbers to obfuscate the control flow in a program.

More generally, there has been some work over the last three years examining what adversaries can do given a large number of compromised computers; this affords an adversary vast computing resources and access to user data. The possibilities in terms of exploiting user data include sending more convincing spam (Aycock, & Friess, 2006) and establishing markets for efficient sales of purloined data (Friess, Aycock, & Vogt, 2008). In terms of using the computing power of compromised machines, the output from cryptographic hash functions has been searched for useful values for anti- disassembly (Aycock, De Graaf, & Jacobson, 2006), and the computing power required to forge SSL server certificates has been estimated (Hemmingsen, Aycock, & Jacobson, 2007).

This latter work brings up to date an earlier publication by White, looking at using computer viruses' application to distributed computing (White, 1989). White, in turn, was anticipated by the (non-malicious) Xerox worm experiments that also were being used as a distributed computing framework (Shoch, & Hupp, 1982). There are now numerous distributed computing projects where users voluntarily and knowingly apply their computers' computing power towards a goal, like SETI@home (Anderson et al, 2002).

Recent work by Sharif et al (2009) is able to take an interpreter and automatically reverse engineer it. However, their technique relies on being able to run the interpreter on the interpreted code. With our PRNG-based techniques, if the correct seed is unavailable, then the interpreted code will not be either.

V. Conclusion

This work has demonstrated that pseudo-random number generators can be used for code obfuscation. Our techniques require an extensive search for obfuscation, even though the deobfuscation is extremely cheap, making it suitable for an adversary who has access to large amounts of computing power such as that of a botnet. Our interpreted methods can be used to obfuscate arbitrary code sequences up to 203 bytes in length, and we conjecture that yet longer sequences are waiting to be discovered with different search parameters. Furthermore, it is stronger than interpreter-based obfuscation alone, in the sense that the code to be interpreted is not present until regenerated. While PRNG-based obfuscation is not perfect, it is novel, and novelty in malicious software is an advantage for the adversary.

Acknowledgment

The first author's work is supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada. Thanks to the anonymous referees for their helpful comments.

References

Franklin, J., Paxson, V., Perrig, A. & Savage, S. (2007). An inquiry into the nature and causes of the wealth of Internet miscreants. *14th ACM Conference on Computer and Communications Security* (pp. 375-388). New York, New York, United States of America. doi:10.1145/1315245.1315292

Szappanos, G. (2007). Exepacker blacklisting. *Virus Bulletin.* Available: http://www.virusbtn.com/virusbulletin/archive/2007/10/vb200710-exepacker-blacklisting

Collberg, C., Thomborson, C. & Low, D. (1997). A taxonomy of obfuscating transformations. *Technical Report 148*. Available: https://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf

Rajab, M., Zarfoss, J., Monrose, F. & Terzis, A. (2007). My botnet is bigger than yours (maybe, better than yours): why size estimates remain challenging. *1st Workshop on Hot Topics in Understanding Botnets (HotBots '07).* Available: https://www.usenix.org/legacy/event/hotbots07/tech/full_papers/rajab/rajab.pdf

Sterling, T. (2005). *Prosecutors say Dutch suspects hacked 1.5 million computers worldwide.* Associated Press. Available: http://www.foxnews.com/story/2005/10/20/dutch-hackers-infected-15-million-computers/

Larkin, E. (2007). Storm worm's virulence may change tactics. *Network World,* 2. Available: http://www.networkworld.com/news/2007/080207-black-hat-storm-worms-virulence.html

Holz, T., Steiner, M., Dahl, F., Biersack, E. & Freiling, F. (2008). Measurements and mitigation of peer-to-peer-based botnets: A case study on Storm Worm. *1st USENIX Workshop on Large-Scale Exploits and Emergent Threats.* Available: https://www.usenix.org/legacy/events/leet08/tech/full_papers/holz/holz.pdf

Tiu, V. (2009). Confounded Conficker. *Virus Bulletin,* March, pp. 7–11.

F-Secure (2009). Calculating the size of the Downadup outbreak. *Weblog: News from the Lab*, 16 January 2009. Available: http://www.f-secure.com/weblog/archives/00001584.html

Finjan (2009). *How a cybergang operates a network of 1.9 million infected computers.* Available: http://www.finjan.com/MCRCblog.aspx?EntryId=2237

Porras, P., Saidi, H. & Yegneswaran, V. (2009). An analysis of Conficker's logic and rendezvous points. *SRI International Technical Report.* Available: http://mtc.sri.com/Conficker

Riordan, J. & Schneier, B. (1998). Environmental key generation towards clueless agents. *Mobile Agents and Security,* 1419, pp. 15–24.

EICAR (2006). *The anti-virus or anti-malware test file.* Available: http://www.eicar.org/anti_virus_test_file.htm

Ferrie, P. (2010). Anti-unpacker tricks. *Virus Bulletin*, May, pp. 4–9.

Oreans Technology (2009). *Code Virtualizer.* Available: http://www.oreans.com/codevirtualizer.php

VMPsoft (2009). *VMProtect.* Available: http://vmpsoft.com/

Debaere, E. & Van Campenhout, J. (1990). Intrepretation and Instruction Path Coprocessing. *ACM SIGPLAN Notices*, 25(9), pp. 7-9.

Klint, P. (1981). Interpretation techniques. *Software – Practice and Experience*, 11(9), pp. 963–973.

Kuenning, G. (2007). *Mersenne Twist pseudorandom number generator package, version 1.20*. Available: http://www.cs.hmc.edu/~geoff/mtwist.html

Toyofuku, T., Tabata, T. & Sakurai, K. (2005). Program obfuscation scheme using random numbers to complicate control flow. *1st International Workshop on Security in Ubiquitous Computing Systems*, 3823, pp. 916-925.

Aycock, J. & Friess, N. (2006). Spam zombies from outer space. *15th Annual EICAR Conference,* pp. 164–179.

Friess, N., Aycock, J. & Vogt, R. (2008). *Black market botnets.* Department of Computer Science, University of Calgary 2500 University Drive N.W., Calgary, Alberta, Canada.

Aycock, J., De Graaf, R. & Jacobson, M. (2006). Anti-disassembly using cryptographic hash functions. *Journal in Computer Virology,* 2(1), pp. 79–85.

Hemmingsen, R., Aycock, J. & Jacobson, M. (2007). Spam, phishing, and the looming challenge of big botnets. *EU Spam Symposium.*

White, S. (1989). Covert distributed processing with computer viruses. *Advances in Cryptology – CRYPTO '89 Proceedings,* pp. 616– 619, LNCS 435.

Shoch, J. & Hupp, J. (1982). The "worm" programs – early experience with a distributed computation. *Communications of the ACM,* 25(3), pp. 172–180.

Anderson, D., Cobb, J., Korpela, E., Lebofsky, M. & Werthimer, D. (2002). SETI@home: An experiment in public-resource computing. *Communications of the ACM,* 45(11), pp. 56–61.

Sharif, M., Lanzi, A., Giffin, J. & Lee, W. (2009). *Automatic reverse engineering of malware emulators.* IEEE Symposium on Security and Privacy. Georgia Institute of Technology, USA.