J Comput Virol
DOI 10.1007/s11416-006-0011-3

ORIGINAL PAPER

Anti-disassembly using cryptographic hash functions

John Aycock · Rennie deGraaf · Michael Jacobson Jr

Received: 13 January 2006 / Accepted: 26 March 2006

Abstract Computer viruses sometimes employ coding techniques intended to make analysis difficult for anti-virus researchers; techniques to obscure code to impair static code analysis are called *anti-disassembly* techniques. We present a new method of anti-disassembly based on cryptographic hash functions which is portable, hard to analyze, and can be used to target particular computers or users. Furthermore, the obscured code is not available in any analyzable form, even an encrypted form, until it successfully runs. The method's viability has been empirically confirmed. We look at possible countermeasures for the basic anti-disassembly scheme, as well as variants scaled to use massive computational power.

Keywords Code armoring · Reverse-engineering · Virus · Disassembly · Hash function

1 Introduction

Computer viruses whose code is designed to impede analysis by anti-virus researchers are referred to as *armored* viruses.[1] Armoring can take different forms, depending on the type of analysis being evaded: dynamic analysis as the viral code runs, or static analysis of the viral code. In this paper, we focus on static analysis.

Static analysis involves the tried-and-true method of studying the code's disassembled listing. *Anti-disassembly* techniques are ones that try to prevent disassembled code from being useful. Code using these techniques will be referred to as *disassembly-resistant code* or simply *resistant code*. Although we are only considering anti-disassembly in the context of computer viruses, some of these techniques have been in use as early as the 1980s to combat software piracy [8].

Ideally, resistant code will not be present in its final form until run time – what cannot be seen cannot be analyzed. This could involve self-modifying code, which presents problems for static analysis [9]. It could also involve dynamic code generation, such as that performed by a just-in-time compiler [2].

In this paper, we present a new method of anti-disassembly based on dynamic code generation, which has the following properties:

- It can be targeted, so that the resistant code will only run under specific circumstances. We use the current username as a key for our running example, but any value available to the resistant code (or combinations thereof) with a large domain is suitable, like a machine's domain name. Because this key is derived from the target environment, and is not stored in the

J. Aycock (✉) · R. deGraaf · M. Jacobson Jr
Department of Computer Science,
University of Calgary,
Calgary, AB,
Canada
e-mail: aycock@cpsc.ucalgary.ca

R. deGraaf
e-mail: degraaf@cpsc.ucalgary.ca

M. Jacobson Jr
e-mail: jacobs@cpsc.ucalgary.ca

[1] The techniques we describe can be used by any malicious software, or *malware*, so we use the term "computer virus" in this paper without loss of generality.

virus, our method may be thought of as environmental key generation [11].
- The dynamically generated code is not available in any form, even an encrypted one, where it can be subjected to analysis until the resistant code runs on the targeted machine. Other encryption-based anti-disassembly methods require that the resistant code be available in encrypted form (e.g., [5]), in which case it may be subject to analysis.
- Even if the dynamically generated code were somehow known or guessed, the exact key used by the resistant code is not revealed.
- It does not rely on architecture-specific trickery and is portable to any platform.

Below, we begin by explaining our anti-disassembly technique and presenting some empirical results. We then look at how the technique might be countered, along with some more entrepreneurial means of deployment.

2 The idea

A cryptographic hash function is one that maps each input to a fixed-length output value, such that it is not computationally feasible to reverse the process, nor is it easy to locate two inputs with the same output [13]. Like regular hash functions, a cryptographic hash function is many-to-one.

Our idea for anti-disassembly is to combine a key – here, we use the current username for concreteness – with a "salt" value, and feed the result as input into a cryptographic hash function. The hash function produces a sequence of bytes, from which we extract a subsequence between bytes *lb* and *ub*, and interpret that subsequence as machine code. We will refer to this subsequence as a *run*. The salt value, for this application, is a sequence of bytes chosen by the virus writer to ensure that the desired run appears in the hash function's output when the correct key is used.

Our anti-disassembly scheme is illustrated in Figure 1, and its application to code armoring is shown in Figure 2. The latter diagram shows how a virus writer can choose arbitrary instructions, and replace them with the machinery to reproduce those instructions using a cryptographic hash function.

A pseudocode example of this idea is shown in Figure 3; from a high-level point of view, this is what an analyst would be confronted with. The code for a cryptographic hash function is assumed to be available, likely in a system library, and `run` is the code sequence that the virus writer is trying to hide. The task of the analyst is to determine precisely what this code does when executed (the value of `run`) and what the target is (the correct value of `key`).

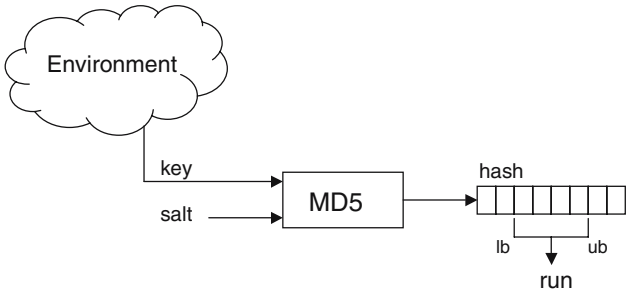

Fig. 1 Conceptual overview of anti-disassembly, using MD5

This pseudocode uses the username as a key, and MD5 as the cryptographic hash function [12]; + is the concatenation operator. MD5 is now known to be vulnerable to collisions [15], i.e., finding two inputs with the same MD5 hash value, but this is irrelevant to our technique. Why? Even if `run` is known by the analyst, the ability to find collisions does not help the analyst identify the exact key that produces a particular hash value containing `run`. If `run` is not known, being able to find two keys that yield the same hash value does not identify either a key that triggers a malicious value of `run` nor the value of `run` itself. In any case, our anti-disassembly technique can be used with any cryptographic hash function, so a different/stronger one can be chosen if necessary.

There are three issues to consider:

1. Having the wrong key. Obviously, if the wrong key value is used, then the run is unlikely to consist of useful code. The resistant code could simply try to run it anyway, and possibly crash; this behavior is not out of the question for viruses. Another approach would be to catch all of the exceptions that might be raised by a bad run, so that an obvious crash is averted. A more sophisticated scheme could check the run's validity using a checksum (or re-using the cryptographic hash function), but this would give extra information to a code analyst.
2. Choosing the salt. This is the most critical aspect; we suggest a straightforward brute-force search through possible salt values. Normally, conducting a brute-force attack against a cryptographic hash function to find an input that has a particular hash value would be out of the question, because the hash functions are designed to make this computationally prohibitive. However, we are only interested in finding a subsequence of bytes in the hash value, so our task is easier. An analysis of the expected computational

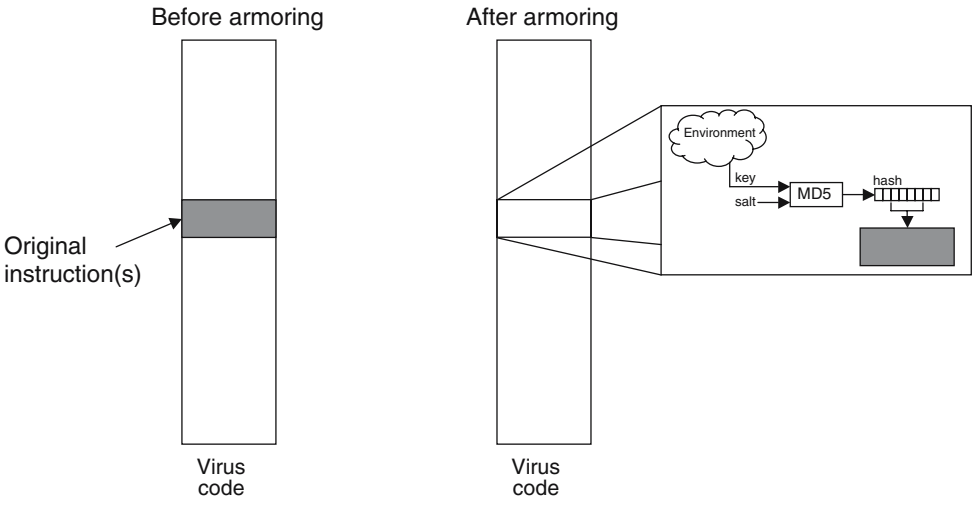

Fig. 2 Using cryptographic anti-disassembly for armoring

```
key ← getusername()
hash ← md5(key + salt)
run ← hash_lb...ub
goto run
```

Fig. 3 Anti-disassembly pseudocode

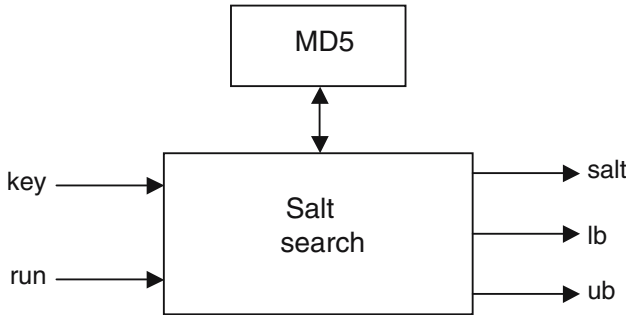

Fig. 4 Salt search using MD5

effort required to find the required salt is presented in the next section.

3. Choosing *lb* and *ub*. These values are derived directly from the hash value, once the desired salt is found.

The salt search (Figure 4) is by far the most time-consuming operation, but this need only be done once, prior to the release of the resistant code. The search time can be further reduced in three ways. First, faster machines can be used. Second, the search can be easily distributed across multiple machines, each machine checking a separate part of the search space. Third, the search can be extended to equivalent code sequences, which can either be supplied manually or generated automatically [7,10]; since multiple patterns can be searched for in linear time [1], this does not add to the overall time complexity of the salt search.

3 Analysis

In order to find a salt value, we simply compute the cryptographic hash of

`key + salt`

for all possible salt values until the hash output contains the required byte sequence (run). The pseudocode for this using MD5 is shown in Figure 5. In order to speed up the search, we allow the run to begin in any position in the hash output.

Approximately half of the output bits of a cryptographic hash function change with each bit changed in the input [13]; effectively, we may consider the hash function's output to change randomly as the salt is changed. Given that, the probability of finding a particular b-bit run in a fixed position of an n-bit output is the ratio of the bits not in the run to the total number of bits: $2^{n-b}/2^n$, or $1/2^b$. The expected number of attempts would then be 2^{b-1}. Furthermore, because only the salt is being changed in the brute-force search, this implies that we would need $b-1$ bits of salt for a b-bit run.

```
let key be the desired key
let run be the desired run
let N be the number of salts to try

for salt in 0...N:
    hash ← md5(key + salt)
    if run in hash:
        lb ← start position of run in hash
        ub ← end position of run in hash
        output salt
        output lb
        output ub
        halt
output "no salt found"
```

Fig. 5 Salt search pseudocode

If we allow *lb*, the starting position of the run, to vary, the expected number of attempts will be reduced by a factor equal to the number of possible values of *lb*. If we index the starting position at the byte level, then there are $m = (n - b)/8$ possible starting positions. The probability of finding the b-bit run increases to $m/2^b$, and the expected number of attempts becomes $2^{b-1}/m$. Similarly, if we index at the bit level, there are $n - b$ starting positions and the expected number of attempts reduces further to $2^{b-1}/(n - b)$.

Notice that the computational effort depends primarily on the length of the run, not the length of the hash function output. The length of the hash function only comes into play in reducing the expected number of attempts because the number of possible values for *lb*, the starting point of the run, depends on it.

We only discuss the case of single runs here, but this technique trivially extends to multiple runs, each with their own salt value. Because the salt computation for each run is independent of the others, the total effort required for multiple-run computation scales linearly. If the computational effort to compute the salt for one run is X, then the effort for one hundred runs is $100X$.

As an example of salt computation, suppose we want our run to consist of a single Intel x86 relative jump instruction. This instruction can be encoded in 5 bytes, so we need to find a salt that, when concatenated to the key, yields a hash value containing this 5-byte run starting in any position. The MD5 hash function has 128-bit outputs, so if we index the run at the byte level, there are 11 possible values for *lb*. The expected number of attempts to find the run is therefore

$$2^{39}/11 < 2^{36}.$$

If instead we index at the bit level, there are 88 possible values for *lb* and the expected number of attempts is

$$2^{39}/88 < 2^{33}.$$

Using a 160-bit hash function such as SHA-1 yields $2^{39}/15$ and $2^{39}/120$ when indexing *lb* at the byte and bit levels, respectively. In all cases, the computation can be done in only a few hours on a single modern desktop computer.

It is feasible to use this method to find runs slightly longer than 5 bytes, but the computational effort adds up very quickly. For example, to find an 8-byte run using SHA-1 and indexing *lb* at the bit level would require roughly $2^{63}/120 > 2^{56}$ attempts. A special-purpose, massively parallel machine would likely be required to find the run in this case, as the computational effort involved is roughly equivalent to that required to break the DES block cipher, for which such hardware was also required [6].

4 Empirical results

To demonstrate the feasibility of this anti-disassembly technique, we searched for the run (in base 16)

```
e9 74 56 34 12.
```

These 5 bytes correspond on the Intel x86 to a relative jump to the address 12345678_{16}, assuming the jump instruction starts at address zero.

The search was run on an AMD AthlonXP 2600+ with 1 GB RAM, running Fedora Core 4 Linux. We tested five different keys with 1- to 5-byte salts, sequentially searching through the possible salt values. [2] Table 1 shows the results for three cryptographic hash functions: MD5, SHA-1, and SHA-256. For example, the salt "07e9717a09," when concatenated onto the key "aycock," yields the SHA-1 hash value

```
ef 6d f4 ed 3b a1 ba 66 27 fe
e9 74 56 34 12 a2 d0 4f 48 91.
```

Numbering the hash value's bytes starting at zero, our target run is present with $lb = 10$ and $ub = 14$. The run is highlighted in gray above.

For our purposes, it is sufficient to demonstrate that it is possible to find salt values that produce a given run. To put the search times in Table 1 into context, however,

Table 1 Brute-force salt search for a specific 5-byte run

Algorithm	Key	Salt	# Salts tested	Search Time (s)
MD5 (128 bits)	aycock	55b7d9ea16	96915712675	80262
	degraaf	a1ddfc1910	68082987191	58356
	foo	e6500e0214	84599106230	73206
	jacobs	9ac1848109	40201885669	34557
	ucalgary.ca	4d21abe205	24899771642	23059
	Average		62939892681	53888
SHA-1 (160 bits)	aycock	07e9717a09	40084590622	38795
	degraaf	0d928a260e	59834611693	57907
	foo	2bc680de1e	130536957733	125537
	jacobs	ca638d5e06	26937346972	26314
	ucalgary.ca	585cc614	344525998	339
	Average		51547606603	49778
SHA-256 (256 bits)	aycock	7cad4d4807	30796664539	46360
	degraaf	dd72e2380a	43225788191	64625
	foo	c17a8c3629	174262804678	260641
	jacobs	effa7fc07	33787185089	51744
	ucalgary.ca	48343fa40f	66147214782	101823
	Average		69643931455	105039

[2] For implementation reasons, we iterated over salt values with their bytes reversed, and did not permit 0 bytes in the salts.

Table 2 gives benchmark results for the three cryptographic hash functions we used. The times shown are the total user and system time for 10,000,000 hash computations, using different input lengths to the hash function. At these input lengths, the input size has little effect on the results. SHA-1 hashes took about 28% longer to compute than MD5 hashes, and SHA-256 hashes took about 125% longer.

Another question is whether or not every possible run can be produced. Using the key "aycock," we were able to produce all possible 3-byte runs with 3 bytes of salt, but could only produce 6% of 4-byte runs with a 3-byte salt. With a 4-byte salt, we were able to generate 4-byte runs which covered between 99.999 and 100% of the possible combinations – this was checked with five different keys and three different cryptographic hash functions. (Our test system did not have sufficient memory to record coverage data for 5-byte runs in a reasonable amount of time.) The 4-byte run data are shown in Table 3.

These results tend to confirm our probability estimate from section 3: b-bit runs need $b - 1$ bits of salt.

Table 2 Cryptographic hash function benchmark results (times are in seconds)

	Input length (bytes)		
Algorithm	8	12	16
MD5	5.22	5.05	5.08
SHA-1	6.66	6.49	6.50
SHA-256	11.72	11.54	11.39

Table 3 Generation of possible four-byte runs using a four-byte salt

Algorithm	Key	Runs found	Runs not found
MD5 (128 bits)	aycock	4294936915	30381
	degraaf	4294937044	30252
	foo	4294936921	30375
	jacobs	4294937188	30108
	ucalgary.ca	4294936946	30350
	Average	4294937003	30293
SHA-1 (160 bits)	aycock	4294966707	589
	degraaf	4294966733	563
	foo	4294966660	636
	jacobs	4294966726	570
	ucalgary.ca	4294966769	527
	Average	4294966719	577
SHA-256 (256 bits)	aycock	4294967296	0
	degraaf	4294967296	0
	foo	4294967296	0
	jacobs	4294967296	0
	ucalgary.ca	4294967296	0
	Average	4294967296	0

Four-byte runs are of particular interest for portability reasons, because RISC instruction sets typically use instructions that are 4 bytes long; this means that at least one RISC instruction can be generated using our technique. One instruction may not seem significant, but it is sufficient to perform a jump anywhere in the address space, perform an arithmetic or logical operation, or load a constant value – potentially critical information that could be denied to an analyst.

5 Countermeasures

An analyst who finds some resistant code has several pieces of information immediately available. The salt, the values of *lb* and *ub*, and the key's domain (although not its value) are not hidden. The exact cryptographic hash function used can be assumed to be known to the analyst, too – in fact, resistant code could easily use cryptographic hash functions already present on most machines.

There are two pieces of information denied to an analyst:

1. The key's value. Unless the key has been chosen from a small domain of values, then this information may not be deducible. The result is that an analyst may know that a computer virus using this anti-disassembly technique targets someone or something, but would not be able to uncover specifics.
2. The run. If the run is simply being used to obscure the control flow of the resistant code, then an analyst may be able to hazard an educated guess about the run's content. Other cases would be much more difficult to guess: the run may initialize a decryption key to decrypt a larger block of code; the entire run may be a "red herring" and only contain various NOP instructions.

Note that even if the run is somehow known to an analyst, the cryptographic hash function cannot be reversed to get the original key. At best, the analyst could perform their own brute-force search to determine a set of possible keys (recall that the hash function is many-to-one). However, the analyst also knows the salt and the domain of the key, so given the run, the analyst can find the key by exhaustively testing every possible value. This underscores the point that the key domain must be sufficiently large to preclude such a brute-force analysis – our example in section 4 of using usernames as keys would likely not prevent this.

Whether or not every last detail of the resistant code can be found out is a separate issue from whether or not

a computer virus using resistant code can be detected. In fact, malware already exists that can automatically update itself via the Internet, such as Hybris [4], so complete analysis of all malware is already impossible.

Fortunately for anti-virus software, computer viruses using the technique we describe would present a relatively large profile which could be detected with traditional defenses, including signature-based methods and heuristics [14]. Precise detection does not require full understanding.

6 Enter the botnet

What if the computing power available for a brute-force salt search were increased by five orders of magnitude over the computer we used for our experiments? Few organizations have that much computing power at their fingertips, but a few individuals do. A *botnet* is a network of malware-controlled, "zombie" machines that executes commands issued via Internet Relay Chat (IRC) channels [3]. These have been used for sending spam and distributed denial-of-service attacks [3], but they may also be viewed as very large-scale distributed computing frameworks which can be used for malicious purposes.

If a virus writer wants to armor a virus using the anti-disassembly technique described here, especially for long runs with many instructions, a botnet may be used for salt computation. A naïve salt computation on a botnet would involve partitioning the salt search space between machines, and the key and desired run would be available to each machine. Using the earlier Intel x86 relative jump example, for instance, four zombie machines in a botnet could each be given the desired key (e.g., "aycock") and run (e974563412) and a 4-byte salt search could be divided like so:

Zombie 1	00000000 . . . 3fffffff
Zombie 2	40000000 . . . 7fffffff
Zombie 3	80000000 . . . bfffffff
Zombie 4	c0000000 . . . ffffffff

Having the virus writer's desired key and run on each zombie machine would not be a bad thing from an analyst's point of view, because locating any machine in the botnet would reveal all the information needed for analysis.

A more sophisticated botnet search would do three things:

1. Obscure the key. A new key, key', could be used, where key' is the cryptographic hash of the original key. The deployed resistant code would obviously need to use key' too.
2. Supply disinformation. The virus writer may choose lb and ub to be larger than necessary, to mislead an analyst. Unneeded bytes in the run could be NOP instructions, or random bytes if the code is unreachable. (In general, ub need not be revealed by the virus writer at all, if the run is executed by jumping directly to the address of $\texttt{hash}_{lb}$.)
3. Hide the discovery of the desired run. Instead of looking for the exact run, the botnet could simply be used to narrow the search space. A weak checksum could be computed for *all* sequences of the desired length in the hash function's output, and the associated salts forwarded to the virus writer for verification if some criterion is met. For example, the discovery of our 5-byte run in section 4 could be obliquely noted by watching for 5-byte sequences whose sum is 505.

This leaves open two countermeasures to an analyst. First, record the key' value in an observed botnet in case the salt is collected later, after the virus writer computes and deploys it – this would reveal the run, but not the original key. Second, the analyst could subvert the botnet, and flood the virus writer with false matches to verify. The latter countermeasure could itself be countered quickly by the virus writer, however, by verifying the weak checksum or filtering out duplicate submissions; in any case, verification is a cheap operation for the virus writer.

7 Related work and conclusion

There are few examples of strong cryptographic methods being used for computer viruses – this is probably a good thing. Young and Yung discuss cryptoviruses, which use strong cryptography in a virus' payload for extortion purposes [16]. Riordan and Schneier mention the possibility of targeting computer viruses [11], as does Filiol [5].

Filiol's work is most related to ours: it uses environmental key generation to decrypt viral code which is strongly-encrypted. Neither his technique nor ours stores a decryption key in the virus, finding instead the key on the infected machine. A virus like the one Filiol proposes hides its code with strong encryption, carrying the encrypted code around with the virus. In our case, however, the code run never exists in an encrypted form; it is simply an interpretation of a cryptographic hash function's output. Our technique is different in the sense that the ciphertext is not available for analysis.

The dearth of strong cryptography in computer viruses is unlikely to last forever, and preparing for such threats is a prudent precaution. In this particular case of anti-disassembly, traditional defenses will still hold in terms of detection, but full analysis of a computer virus may be a luxury of the past. For more sophisticated virus writers employing botnets to find salt values and longer runs, proactive intelligence gathering is the recommended defense strategy.

Acknowledgements The first and third authors' research is supported in part by grants from the Natural Sciences and Engineering Research Council of Canada. Karel Bergmann and Eric Filiol made helpful comments on early versions of this paper, as did the anonymous referees for EICAR 2006.

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun ACM **18**(6), 333–340 (1975)
2. Aycock, J.: A brief history of just-in-time. ACM Comput Surv **35**(2), 97–113 (2003)
3. Cooke, E., Jahanian, F., McPherson, D.: The zombie roundup: understanding, detecting, and disrupting botnets. In: USENIX SRUTI Workshop, 2005
4. Secure, F.: F-Secure virus descriptions: Hybris, 2001. http://www.f-secure.com/v-descs/hybris.shtml
5. Filiol, E.: Strong cryptography armoured computer viruses forbidding code analysis: The Bradley virus. In: Proceedings of the 14th Annual EICAR Conference, pp. 216–227 (2005)
6. Electronic Frontier Foundation. Cracking DES: secrets of encryption research, wiretap politics, and chip design. O'Reilly, 1998
7. Joshi, R., Nelson, G., Randall, K.: Denali: a goal-directed superoptimizer. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, pp. 304–314, 2002
8. Krakowicz. Krakowicz's kracking korner: The basics of kracking II, c. 1983. http://www.skepticfiles.org/cowtext/100/krckwczt.htm
9. Lo R.W., Levitt, K.N., Olsson, R.A..: MCF: a malicious code filter. Comput Security **14**, 541–566 (1995)
10. Massalin, H.: Superoptimizer: a look at the smallest program. In: Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems, pp. 122–126, 1987
11. Riordan, J., Schneier, B.: Environmental key generation towards clueless agents. In: Mobile Agents and Security (LNCS 1419), pp. 15–24, 1998
12. Rivest, R.: The MD5 message-digest algorithm. RFC 1321, 1992
13. Schneier, B.: Applied cryptography, 2nd edn. Wiley, New York, 1996
14. Szor, P.: The art of computer virus research and defense. Addison-Wesley, Reading, 2005
15. Wang, X., Feng, D., Lai, X., Yu, H.: Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199, 2004. http://eprint.iacr.org/
16. Young, A., Yung, M.: Cryptovirology: extortion-based security threats and countermeasures. In: IEEE Symposium on Security and Privacy, pp. 129–141, 1996