

Component Similarity Based Methods for Automatic Analysis of Malicious Executables

Zhiyin Liang, Tao Wei, Yu Chen, Xinhui Han, Jianwei Zhuge,

Wei Zou(Corresponding Author)

Institute of Computer Science and Technology, Peking University, 100871
Phone: 8610-82529587 Fax: 8610-82529207
{Liangzhiyin, weitao, chenyu, hanxinhui, zhugejianwei, zouwei}@icst.pku.edu.cn

Abstract. In recent years with the popularity of source code sharing, the number of types of malware increases sharply; furthermore, malwares are also getting more and more sophisticated. These developments together propose a tremendous challenge to traditional ways of analyzing malware. One way to handle such challenge is to develop tools to automate the analyzing task, and in this paper we describe one such method for static analysis of malware. Inspired by the observation that a malicious executable usually consists of several components, each performing certain tasks, and that these components are often reused by others, our method employs techniques from reverse engineering and data clustering to component decomposing of an executable. For each component obtained, we then match it against a library of known malware components to identify it. For malicious programs, we further utilize the match result to classify them. We have built a prototype system called CompSim, and have applied it to analyze bot-like malicious executables. Initial results show that our approach has outperformed classical signature-based detection method in terms of false negative rate. Furthermore, comparing to dynamic analysis methods and model checking methods, our static method has the advantage of larger analytical coverage.

Keywords: Reverse Engineering, Malware Analysis, Component Extract

1. Introduction

It used to be a tough task for VXers to write malicious code, but things have changed due to the popularity of resource sharing on the Internet. Nowadays, malicious source code and virus creation tools can be easily downloaded from the Internet and all that one needs to do for creating a new malware variation is just modifying some parameters and then clicking his mouse. All of these reasons contribute to the dramatic increase of new malicious varieties. The majority of them are composed of existing malicious source code or functional components, whereas only a minority of them introduces novel technologies and new functions.

Take the development and transformation of bot programs for instance, bot

programs allow attackers to remotely control vulnerable computers and form virtual networks of zombies ^[Kim07]. Bot programs have been evolved for about ten years stealthily until the source of the first open source bots, SDbot ^[SDbot 02], and a well modularization designed bots, Agobot^[Agobot02] released on the internet, then bots caught the attackers' eyes and subsequently, other traditional malware technologies, such as propagate strategies from worms and virus, self-protecting technologies from rootkits, etc, also have been integrated into bots by the VXers. Gaobot ^[Gaobot04] and rbot ^[Rbot04] are both examples of them, made their infamous outbursts in 2004.

The overwhelming amount of new malware varieties emerging every year calls for more effective analysis tools. Since a malicious executable usually consists of several components, each performing certain tasks, such as infecting, attacking, information stealing, self-protecting etc. and that these components are often reused by other malware, it will make more sense to analyze specific malicious modules of a given malicious program than to analyze the whole program itself. In this paper we propose a novel method to automate malware analysis by capturing components from disassembled binary programs via reverse engineering^{[LZS05][BE82][DK93][PG02][SR00]}. In the process of capturing we utilize both domain knowledge and structural characteristics of binary malicious codes. For each component obtained, we then match it against a library of known malware components. In addition, we have developed an improved version of Call Graph similarity algorithm based on the method proposed by E. Carrera and G. Erdelyi ^[CE04], to calculate the similarity between components, yielding a more accurate comparison.

The benefits of our approach are three-folded: (1) Compared with classical malware analysis methods, this new approach can give more detailed information about instances than signature-based detection and whole-sample-based classification. In addition, constructing a component library is much easier than constructing a library for model checking. (2) By categorizing components into known and unknown ones, our approach accelerates the whole malware analysis process: If a component fails to find a match in the library, such component might contain novel malware strains, so that only unknown components become our focus. (3) Furthermore, we can apply the comparing result to classify malicious codes by tracking their origin and constructing their genealogy tree based on their component similarity rate.

2. Related works

In the books of [Szo05] [SZ03], the authors illustrate the state-of-the-art technology in malware creation and analysis. They classify malicious behaviors according to their purposes, and point out that a malicious executable program is often composed of different malicious modules.

Anti-virus researchers have taken great efforts to automate the analysis process of

a malicious executable, in order to discover technical details of the malicious behaviors and to classify the malware. The methods for doing such automation can be classified into two categories—dynamic and static.

Dynamic methods often refers to first running malicious executables in a secured environment such as a sandbox, or executing samples via an emulator^[BMKK06]^{[Sandbox][YGN+06]}, and then collecting the behaviors information^[LM06]. A dynamic method won't be hampered by obfuscation or other anti-disassemble tricks employed in malwares, and generally it can retrieve useful information. However, limited by the running environment, most of the malware execute paths might not be triggered or sometimes the subject could not even run in the controlled environment.

Static methods deal with the binary codes or disassembled ones directly. String salvaging tools applied on binary codes are often used for collecting clues of all aspects to aid the manual analysis^[Szo05]^[SZ03], and data-mining methods applied directly on binary codes can automatically distill malicious code signatures for virus detection^{[KA94][SEZS01]}. J. Bergeron *et al*^[BDEK99] proposed model checking methods^{[HD02][KKS05][SL03]} and semantics-aware malware detection^{[CJSSB05][KRV04]} to check whether certain malicious behaviors exist in given samples. Model checking method and semantics-aware method both can be applied to identify malicious behavior, but the quality of the result were sensitively to the quality of known malicious models(or templates), and it is always not a trivial task to create good models(or templates) with both low false negative and low false positive rates.

The classification of malware is another important issue for malware research. Data-mining and data-clustering methods have been applied to cluster malware samples, and API sequence^[SXC04], function calls graph^[CE04] and control flow graph, program dependence graph^[Ste93] and other criteria all have been used as clustering features. Clustering methods are helpful on grouping polymorphic varieties of known malware, but nearly all existing methods treat the sample as a whole, which makes it very hard to find out the intrinsic relationships between malware variations, which only adopted parts of existing malwares, i.e. the malicious components

3. Architecture of bot programs

It is hard to define a unified nomenclature for malicious programs^[Szo05]. Terms such as Worms, Backdoors (Trapdoors), Password-Stealing, Exploits, Injectors, Spammer Programs, Flooders, Keyloggers, Rootkits, etc are mainly based on typical malicious behaviors of a malware. With the evolvement of malware technology, the integration of malicious behaviors into one single malware has been a trend and bot program is a typical kind of integrated malware.

The term "botnet" can be used to refer to any group of bots, such as IRC bots. The

word is generally used to refer to a collection of compromised computers (a.k.a. zombie computers) running programs, usually referred to as worms, Trojan horses, or backdoors, under a common command and control infrastructure. A botnet's originator (a.k.a. "bot herder") controls the group remotely, usually through means such as IRC, and usually for nefarious purposes. Individual programs manifest as IRC "bots". A bot typically runs stealthy, and complies with the IRC standard. Generally, the perpetrator of the botnet has compromised a series of systems using various tools (exploits, buffer overflows, etc). New infected bots can automatically scan their environment and propagate themselves using vulnerabilities and weak passwords ^[WikiBot].

As described in [BY06]: a bot program is typically consist of several kinds of mechanisms, such as botnet control mechanism, host control mechanism, propagation mechanisms, exploit and attack mechanisms, malware delivery mechanisms, obfuscation mechanisms, and deception mechanisms. To implement these mechanisms, a bot program generally contains a command and control module and some optional functional modules, including propagation modules, host control modules, download/update modules, information theft modules and evading-detection/anti-analysis modules, as shown as in Fig. 1.

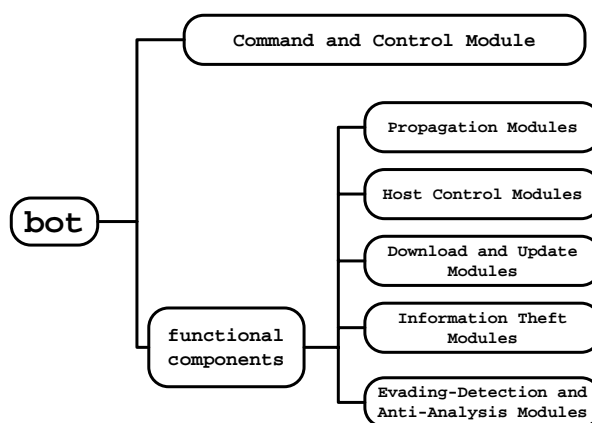


Fig.1. Architecture of bot programs

The modular architecture of bot programs makes it suitable to apply component-based approaches to analyze these malware.

4. Component similarity based automatic analysis strategy

This section presents the methodology of malicious code analysis based on component similarity.

4.1 Framework

In this paper, a component is defined as one module built up by a cluster of related functions in order to implement some specific tasks. The cluster has a header function, a.k.a. component header, which calls all other functions in the cluster directly or indirectly. Fig.2 shows an instance of such a component.

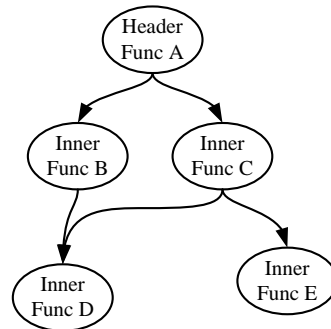


Fig.2. A component headed by Func A

The framework of our approach is shown in Fig.3.

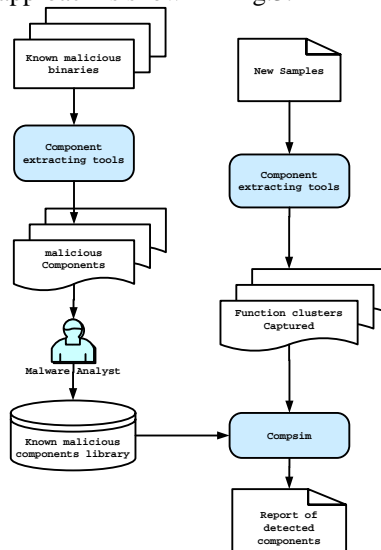


Fig.3. framework of our approach

Workflow:

- 1) An assistant tool extracts potential functional components from known malicious code samples;
- 2) Experts check these candidates and register valid ones, including their functional description and family information, into a malicious component library;
- 3) Given an unknown sample, another assistant tool detects the existence of

known malicious components in the new sample. If the result shows that the new sample contains many known malicious components, we can then tell that it is a malicious code, what it is supposed to do, and which family it belongs to.

In the workflow, the most important parts are component extraction and detection, and we discuss them in the following.

4.2 Component extraction

A software component is a system element offering a predefined service and able to communicate with other components. Characters of typical components may include multiple-use, non-context-specific, composable with other components, encapsulated i.e., non-investigable through its interfaces and to be a unit of independent deployment and versioning^[WikiCom]. Due to the development of the software component technology, as an effective means to acquire component, extraction of components from existing systems has become a research focus of both software reuse and program-comprehension for its lower cost and high efficiency^{[BE82][DK93][Kos02]}.

Typical techniques for extracting components are either knowledge-based^{[DK93][BMW93][SR00][PG02]} or structure-based^{[DMM99][CCK00][RRH+00]}. Knowledge-based methods use design knowledge, domain-specific knowledge, metric definitions, reengineering knowledge, etc. to split a software system into separate parts^[DK93], while structure-based methods split or cluster parts based on some abstract structures of the software system, such as class graph, call graph, module dependence graph^{[DMM99][CCK00][RRH+00]}, etc. .

For extracting components and implementing specific functions in malicious code, we mainly use knowledge-based methods, esp. domain-specific knowledge. Such components are usually small, and classic structured-based methods are not effective as usual.

In this paper, we extract potential components by identifying their header functions in three ways mainly based on domain-specific knowledge:

- 1) Dispatching based: identifying component headers that are invoked from the dispatching function;
- 2) Key APIs based: identifying component headers that call a set of key APIs directly or indirectly;
- 3) Multiply-invoking based: identifying component headers that are invoked multiple times from multiple sections of code^[DK93].

4.2.1 Dispatching

Dispatch is the process of mapping a message to a specific sequence of code (method) at runtime^[WikiDD]. Malicious codes, esp. bots, usually invoke functional components from a dispatching function according to the received commands.

After identifying the dispatching function, we can easily extract functional components in all the functions invoked from this function.

There are two typical kinds of dispatching implementation. The first kind is a straightforward one: the dispatching function parses the received messages and calls relative functions directly, such as `irc_parseline()` in `rbot` and `sdbot`. The second kind is a register-dispatch one: every component registers itself in a global dispatching table, and then the dispatcher invokes relative functions according to the result of parsing and the dispatching table, such as `g_cMainCtrl.m_cCommands` in `agobot`.

Both of these dispatching implementations have distinguishing characteristics, so experts could recognize them without too much effort in most cases.

4.2.2 Key APIs

A functional component in malicious codes is usually to perform some malicious tasks, such as killing antivirus processes, attacking a victim web site, logging keyboard activities, and so on. In order to perform these tasks, the component usually needs to invoke a set of key system APIs, either directly or indirectly. For example, to kill antivirus processes, it usually needs to invoke { `AdjustTokenPrivileges`, `CreateToolhelp32Snapshot`, `LookupPrivilegeValueA`, `Module32First`, `OpenProcess`, `Process32First`, `Process32Next`, `TerminateProcess` }; to log keyboard activities, it usually needs to invoke { `GetAsyncKeyState`, `GetForegroundWindow`, `GetKeyState` }.

Reversely, we can extract potential components by checking those functions which those key APIs converge into. The set of key APIs for specific task is important for this method, which could be defined by experts or could be collected in known components identified using other approaches.

4.2.3 Multiply-invoked functions

The last approach we use is proposed by M.F. Dunn and J.C. Knight in 1993^[DK93]. The only information needed is the set of invoked and invoking functions and the associated call graph. For example, if function A invokes functions B and C, and functions B and C invoke function D. Function D could be flagged as potentially reusable because of its multiple invocations. To avoid flagging the standard C library functions as reusable, a relation is included in the set of rules that identifies all of these functions, and any that are invoked are ignored.

4.3 Component detection

In this paper, we detect the existence of known malicious components by comparing the functions in the new sample with the functions in those known components in the library. If every function in a component has a matched version

in the new sample, we can tell that the new sample contains the component.

Here we transform the component detection problem into the classic function comparison problem.

There are two typical approaches for comparing functions^{[CE04][BMM06][BMM07]} in binary executables. The first one is proposed by E. Carrera and G. Erdelyi in 2004^[CE04], which is mainly based on call graph matching. The second one is proposed by D. Bruschi et al in 2006^{[BMM06][BMM07]}, which is based on control flow graph normalization and subgraph isomorphism. We prefer to adopt E. Carrera and G. Erdelyi's approach, because subgraph isomorphism can not handle code variation well and there is no mature inexact graph matching library available yet.

We propose a “weight-threshold” call graph inexact matching algorithm which is an improved version of the call graph matching algorithm proposed by E. Carrera and G. Erdelyi.

The algorithm has three steps:

- 1) Topological sort of call graph^{[CLRS01][Ste93]}: sort function nodes in a call graph to a sequence, such that in the sequence a function in the front never calls a function in the back unless there is a recursive call between them.
- 2) Node weighting: compute function nodes' weight along the sequence. If a function node doesn't call any functions, its weight is 1. If a function call some other functions (not including recursive call), its weight is 1 plus the sum of these functions' weight.
- 3) Compute the similarity between functions according to weight and threshold: compute function nodes' similarity along the sequences of the target sample and the component. When two functions are both APIs, the similarity is 1 if they are the same or 0 if they are not the same. When one function is an API while the other function is not an API, the similarity is 0. When two functions are not API, the similarity is the weighted sum of the similarity between the sets of functions they call. However, if the similarity is below than a predefined threshold (0.8 in our prototype), it is reset to 0. The pseudo code of computing similarity between two functions (not including API cases) is shown in Fig.4.

```

function sim_rate(func src, func dst)
{
    List cand_pair;
    Set src_old, dst_old;

    if(src.weight > dst.weight)
        total_weight = src.weight;
    else
        total_weight = dst.weight;

    // collect candidate matched pairs in functions called
    for_each(f, src.calls)
        for_each(fs, f.sim_funcs)
            if(fs.sim_func in dst.calls)
                cand_pair.append((fs.sim_rate,f, fs.sim_func));

    cand_pair->sort();

    total_match=1;
    while(cand_pair.len()!=0){
        // choose the most similar pair
        tuple max = cand_pair.pop();
        if( src_old->has(max[1]) || dst_old->has(max[2]) )
            continue;

        sim_max = max[0];
        if(sim_max==0) break;

        if(src.weight > dst.weight)
            total_match += max[1].weight*sim_max;
        else
            total_match += max[2].weight*sim_max;
        // prevent the matched pair being used again
        src_old.add(max[1]);
        dst_old.add(max[2]);
    }
    sim_rate= (total_match)/total_weight;
    // threshold
    if(sim_rate < SIM_RATE_THRESHOLD)
        return 0;
    else
        return sim_rate;
}

```

Fig.4. pseudo code of computing similarity between two functions

5. Experimental results

5.1 Component extracting

The selected instances for component extracting include: 1) two compiled binaries of source code release of rBot(v0.6.6a) and Agobot3(v0.2.1pre4); 2) one binary sample of W32.Ecup. We have applied all three approaches proposed in section 4.2 in extracting components from these samples. Here are the results:

- 1) rBot v0.6.6a (28 components in total):
AdvPortScanner, AdvScanner, DCCGetThread, DDOSThread, DownloadThread,

FindFileThread, HTTP_Server_Thread, ICMPFloodThread, IdentThread, KeyLoggerThread, Isass, RedirectThread, RlogindThread, SecureThread, ShowLogThread, SniffThread, Socks4ClientThread, Socks4Thread, SynFloodThread, TcpFloodThread, tftpserver, udp, kill_av, ping, ...

2) Agobot3 v0.2.1pre4 (37 components in total):
 CPolymorph_MapFile, ScanPort, encrypt_shellcode, CInstaller_RegStartAdd, CInstaller_Uninstall, CScannerDCOM2_Exploit, FtpWrite, GetCopies, CCommands_RegisterCommand, CScannerNetBios_GetUsers, FtpRead, FtpSendCmd, KillAV, RedirGRE, SYNflood, CBot_SysInfo, CIRC_NetInfo, RndNick, CInstaller_CopyToSysDir, CScannerDCOM_Exploit, CScannerNetBios_GetShares, CScannerWebDav_Exploit, ...

3) W32.Ecup (7 components in total):
 sub_40700B, sub_407321, sub_40735E, sub_407BE9, sub_4080D1, sub_408459, sub_4088DB

Fig.5. shows the function cluster of the component in rBot which exploits a hole in Isass.

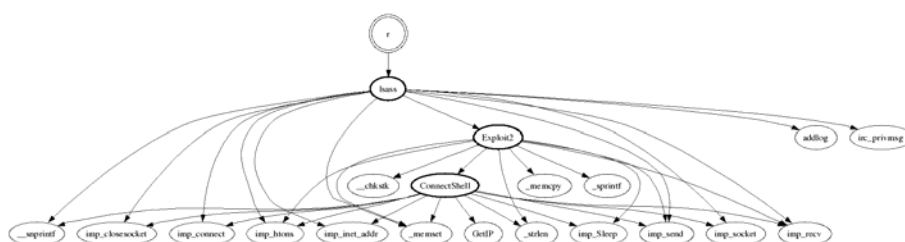


Fig.5. a component for Isass exploiting

5,2 Component detection

We choose 16 unpacked binaries captured by the Chinese Honeynet Project as instances for prototype testing, and then analyze these samples using BESTAR (Binary Executable Structurizer and Analyzer, our in-house decompiler and normalizer) and CompSim (a prototype of component similarity analyzer).

The results of component detection include component statistics and component mapping tables, which are shown as Table 1, Fig.6 and Table 2.

Table 1 and Fig.6 show the number of components detected in each instance. If a significant number of components of an instance are classified as certain malicious code, by common sense, that instance should belong to this malware family. For example, from the table we should conclude that among these 16 instances, 4 are rBot, 6 are Agobot, and 4 are W32.Ecup. Furthermore, the result shows that

different instances often contain various amounts of component.

Instance	hash (first 4 bit)	rBot (total 28)	Agobot (total 37)	Ecup (total 7)	Kaspersky	Trend	Symantec
1	086a	0	0	7	P2P-Worm.Win32.SpyBot.gz	unknown	W32.Ecup
2	1d93	0	1	0	Backdoor.Win32.Rbot.aea	WORM_SPYBOT.GEN	W32.Spybot.Worm
3	3dff	0	0	7	P2P-Worm.Win32.SpyBot.gx	WORM_Generic	W32.Ecup
4	4526	0	0	7	P2P-Worm.Win32.SpyBot.gz	unknown	W32.Ecup
5	4797	0	34	0	Backdoor.Win32.Agobot.gen	WORM_SDBOT.GEN-1	W32.HLLW.Gaobot.gen
6	576c	22	1	0	Backdoor.Win32.Rbot.gen	WORM_RBOT.GEN	W32.Spybot.Worm
7	58c2	4	1	0	Backdoor.Win32.Rbot.gen	WORM_RBOT.GEN-1	W32.Spybot.Worm
8	6f8c	0	0	7	P2P-Worm.Win32.SpyBot.gz	unknown	W32.Ecup
9	7198	0	21	0	Backdoor.Win32.Agobot.gen	WORM_SDBOT.GEN-1	W32.HLLW.Gaobot.gen
10	ae7a	0	0	0	Backdoor.Win32.SdBot.aad	WORM_RBOT.GEN	W32.Spybot.Worm
11	b437	0	34	0	Backdoor.Win32.Agobot.gen	WORM_SDBOT.GEN-1	W32.HLLW.Gaobot.gen
12	b4e3	0	37	0	Backdoor.Win32.Agobot.gen	WORM_AGOBOT.GEN	W32.HLLW.Gaobot
13	b65c	0	19	0	Backdoor.Win32.Agobot.gen	WORM_SDBOT.GEN-1	W32.HLLW.Gaobot.gen
14	d00a	22	1	0	Backdoor.Win32.Rbot.gen	WORM_RBOT.GEN	W32.Spybot.Worm
15	d804	0	34	0	Backdoor.Win32.Agobot.gen	WORM_SDBOT.GEN-1	W32.HLLW.Gaobot.gen
16	e200	21	1	0	Backdoor.Win32.Rbot.gen	WORM_RBOT.GEN	W32.Spybot.Worm

Table 1. Components detected

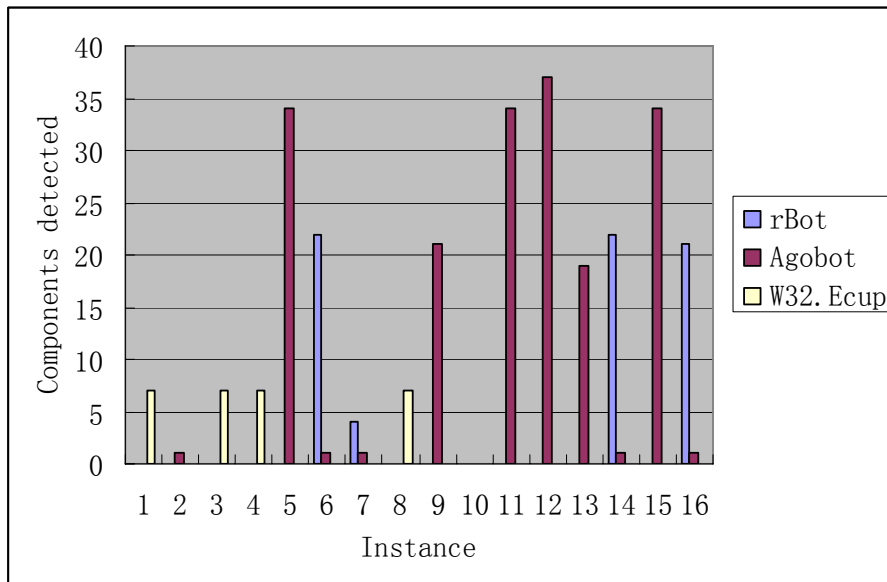


Fig.6. Components detected

Table 1 also shows the identification results of commercial anti-virus products. These results roughly validate our approaches. However, different anti-virus products produce different malware names: Symantec calls rBot as W32.Spybot.Worm, and Agobot as W32.HLLW.Gaobot; Kaspersky calls W32.Ecup as P2P-Worm.Win32.Spybot; Trend fails to recognize W32.Ecup and recognizes 3 Agobot samples as Sdbot by mistake. The naming mechanism of malware becomes more and more complex and confusing, and a single name can't tell the difference between different instances, or what a malware really does.

Furthermore, we manually double check these 16 instances. It turns out that our above conclusion (4, 6, and 4) is indeed correct, and the rest 2 instances—1d93 and ae7a—in fact don't belong to any of these three malware families. The result of these two cross validation shows that our component-based method is quite promising.

The table 2 shows the component mapping table which maps functions in the binary instance into a meaningful component name. This mapping table tells what an instance does in fact, and it helps expert skip these known parts, and therefore accelerate the analysis process.

Component (Agobot)	4797	7198	b4e3	b65c
CInstaller RegStartAdd	sub 406077	sub 40548F	sub 40AC3A	sub 4054A8
CInstaller CopyToSysDir	sub 405C69	sub 404A30	sub 40A82C	N/A
RndNick	sub 40A997	sub 409A02	sub 40F489	sub 409740
FtpWrite	sub 413FEC	sub 4127BA	sub 403FCC	sub 41239D
encrypt shellcode	sub 411426	N/A	sub 4012E0	N/A
KillAV	sub 40CE2B	N/A	sub 41186C	N/A
CRedirectSOCKS Thread	N/A	sub 4114B0	sub 4025FF	sub 4110BA
CScannerDCOM Exploit	sub 45AF60	N/A	sub 4596C4	N/A
CScannerWebDav Exploit	sub 4593D0	N/A	sub 458A10	N/A
CDDOSSynFlood	sub 41286A	sub 41112D	sub 458D0C	sub 410D3D
CDDOSUDPFlood	sub 412A97	sub 411458	sub 458E13	sub 41107E
...

Table 2. Component mapping table

6. Conclusion

Along with the popularity of sharing of malicious source code and virus creation tools, the majority of malware are consisted of different existent malicious component. Consequently, it is more reasonable to analyze the individual modules of a given malicious program than to analyze the program as a whole.

We propose a new approach based on the methodology of component analysis to analyze malwares. Compared with classical malware analysis methods, this new approach can give more detailed information about instances than signature-based detection and whole-sample-based classification. Another advantage of such approach is that building a component library is much easier than building a library for model checking.

Furthermore, we propose several effective methods to salvage potential components mainly based on domain-specific knowledge, and propose a "weight-threshold" call graph inexact matching algorithm, which is an improved version of the call graph matching algorithm proposed by E. Carrera and G. Erdelyi, to compute similarities between components.

We have applied these methods to 16 instances captured by Chinese HoneyNet Project. We have analyzed these instances using BESTAR (Binary Executable Structurizer and Analyzer, our in-house decompiler and normalizer) and CompSim (a prototype of component similarity analyzer). CompSim could effectively recognize components in these instances, and results are validated by commercial anti-virus products and experts. These initial experimental results show that this new approach is very promising in analysis of bot-like malwares.

Acknowledgement

* This research is supported by the National High Technology Research and Development Program of China (No.2006AA01Z445 and No. 2006AA01Z402).

* We would also like to express our thanks to Jing Luo and Jian Mao for their constructive comments and advice.

Reference

- [Agobot02] Sophos Inc, <http://www.sophos.de/support/disinfection/agobot.html>,
- [BDEK99] J. Bergeron, Mourad Debbabi, M. M. Erhioui, Béchir Ktari, "Static Analysis of Binary Code to Isolate Malicious Behaviors" (1999)
- [BE82] L. A. Belady and C. J. Evangelisti, "System Partitioning and its Measure", Journal of Systems and Software, vol. 2, pp. 23-29, 1982.
- [BMKK06] Ulrich Bayer, Andreas Moser, Christopher Kruegel, Engin Kirda, "Dynamic Analysis of Malicious Code", Journal in Computer Virology, Springer Paris, Volume 2, Number 1, Aug, 2006
- [BMM06] D. Bruschi, L. Martignoni, and M. Monga, "Using code normalization for fighting self-mutating malware", Proceedings of International Symposium on Secure Software Engineering, 2006.
- [BMM07] D. Bruschi, L. Martignoni, and M. Monga, "Code Normalization for Self-Mutating Malware", in IEEE SECURITY & PRIVACY, March/April 2007.
- [BMW93] T. J. Biggerstaff, B. G. Mitbender, and D. Webster, "The concept assignment problem in program understanding", Proceedings of Working Conference on Reverse Engineering 1993, pp. 27-43, 1993.
- [BY06] P. Barford and V. Yegneswaran, "An Inside Look at Botnets", Special Workshop on Malware Detection, Advances in Information Security, 2006.
- [CCK00] G. Canfora, J. Czeranski, and R. Koschke, "Revisiting the Delta IC Approach to Component Recovery", Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00), p. 140, 2000.
- [CE04] E. Carrera and G. Erdelyi, "Digital genome mapping: Advanced binary malware analysis", Proceedings of 15th Virus Bulletin International Conference (VB 2004), pp. 187-197, 2004.
- [CJSSB05] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, Randal E. Bryant, "Semantics-Aware Malware Detection", Proceedings of the

- 2005 IEEE Symposium on Security and Privacy, p.32-46, May 08-11, 2005
- [CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, Second ed., MIT Press, 2001.
- [DK93] M. F. Dunn and J. C. Knight, "Automating the detection of reusable parts in existing software", Proceedings of 15th international conference on Software Engineering, pp. 381-390, 1993.
- [DMM99] D. Doval, S. Mancoridis, and B. S. Mitchell, "Automatic clustering of software systems using a genetic algorithm", Proceedings of Software Technology and Engineering Practice, 1999.
- [Gaobot04] Symantec Inc, Gaobot Analysis. 2004. http://www.symantec.com/security_response/writeup.jsp?docid=2004-042914-1054-99
- [Kim07] Kim-Kwang Raymond Choo, "Zombies and botnets", Trends and issues in crime and criminal justice, no. 333, March 2007, <http://www.aic.gov.au/publications/tandi2/tandi333.html>
- [HD02] Chen H, Wagner D. MOPS: An infrastructure for examining security properties of software. In: Proc. of the 9th ACM Conf. on Computer and Communications Security (CCS). Washington: ACM Press, 2002. 235-244.
- [KA94] Jerrey O. Kephart, William C. Arnold, "Automatic Extraction of Computer Virus Signatures", 4th Virus Bulletin International Conference, 1994
- [KKS05] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, Helmut Veith: "Detecting malicious code by model checking", Proceedings of the 2nd International Conference on Intrusion and Malware Detection and Vulnerability Assessment (DIMVA), 2005
- [Kos02] R. Koschke, "Atomic architectural component recovery for program understanding and evolution", Proceedings of International Conference on Software Maintenance 2002, pp. 478-481, 2002.
- [LM06] Tony Lee, Jigar J. Mody, "Behavioral Classification", Proceedings of the EICAR Conference, May 2006
- [LZS05] Jing Luo, Lu Zhang, Jia-Su Sun, "A Survey of Techniques for Extracting Components from Existing Systems", COMPUTER SCIENCE, 2005.
- [PG02] M. Pinzger and H. Gall, "Pattern-supported architecture recovery", Proceedings of 10th International Workshop on Program Comprehension, pp. 53-61, 2002.
- [Rbot04] F-Secure Inc, F-Secure virus descriptions: rbot, 2004. <http://www.f-secure.com/v-descs/rbot.shtml>.
- [RRH+00] D. Rayside, S. Reuss, E. Hedges, and K. Kontogiannis, "The effect of call graph construction algorithms for object-oriented programs on automatic clustering", Proceedings of 8th International Workshop on Program Comprehension, pp. 191-200, 2000.
- [Sandbox] Norman sandbox. <http://sandbox.norman.no/>
- [SDBot02] Symantec Inc, Backdoor.Sdbot 2002. http://www.symantec.com/security_response/writeup.jsp?docid=2002-051312-3628-99&tabid=2
- [SEZS01] Matthew G. Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo, Data Mining Methods for Detection of New Malicious Executables, IEEE, 2001
- [SL03] Prabhat K Singh and Arun Lakhota, "Static Verification of Worm and Virus Behavior in Binary Executables using Model Checking", Proceedings of the

- 2003 IEEE Workshop on Information Assurance ,June 2003
- [SZ03] Ed Skoudis, Lenny Zeltser, "Malware: Fighting Malicious Code", Prentice Hall PTR; 1st edition. Nov. 2003
- [Szo05] P. Szor: The Art of Computer Virus Research and Defense, Addison-Wesley Professional, 2005
- [SR00] D. Spinellis and K. Raptis, "Component mining: A process and its pattern language", Information & Software Technology, vol. 42, pp. 609-617, 2000.
- [Ste93] B. Steensgaard, "Sequentializing program dependence graphs for irreducible programs", Techn. Rep. MSR-TR-93-14, Microsoft Research, Redmond, WA, 1993.
- [SXCM04] AH Sung, J Xu, P Chavez, S Mukkamala: Static Analyzer of Vicious Executables (SAVE)- Computer Security Applications Conference, 2004.
- [WikiDD] Dynamic dispatch, http://en.wikipedia.org/wiki/Dynamic_dispatch
- [WikiCom] Software componentry, http://en.wikipedia.org/wiki/Software_componentry
- [WikiBot] Botnet, <http://en.wikipedia.org/wiki/Botnet>
- [YGN+06] Yang Yu, Fanglu Guo, Susanta Nanda, Lap-chung Lam, Tzi-cker Chiueh, "A feather-weight virtual machine for windows applications", Proceedings of the second international conference on Virtual execution environments, 2006